Programming Language and Tools for Automated Testing

Roy Patrick Tan

Doctoral Dissertation
submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy
in
Computer Science and Applications

Dr. Stephen H. Edwards. Chair
Dr. James D. Arthur
Dr. Manuel Pérez-Quiñones
Dr. Naren Ramakrishnan
Dr. David Tegarden

August 8, 2007
Blacksburg, Virginia
Acknowledgments

I write this with a grateful heart to the Divine Providence that is the foundation for all human endeavor. I would also acknowledge my parents, Lucas and Heddy Tan, whose loving support can never be repaid by any son. Thanks also to Tom, Emil, Martha, and Marion, my brothers and sisters who have kept me in their hearts throughout this endeavor. Thanks to Aaron Tokhy for implementing earlier versions of the test case generator; and to Robert, Jess, Allan, Bibins, Joan, Melanie, and the rest of the Filipino community past and present, who have become my family in Blacksburg. Finally, great gratitude goes to my advisor, Dr. Stephen Edwards, without whose guidance this dissertation could not have been written.
Abstract

Software testing is a necessary and integral part of the software quality process. It is estimated that inadequate testing infrastructure cost the US economy between $22.2 and $59.5 billion. We present Sulu, a programming language designed with automated unit testing specifically in mind, as a demonstration of how software testing may be more integrated and automated into the software development process. Sulu’s runtime and tools support automated testing from end to end; automating the generation, execution, and evaluation of test suites using both code coverage and mutation analysis. Sulu is also designed to fully integrate automatically generated tests with manually written test suites. Sulu’s tools incorporate pluggable test case generators, which enables the software developer to employ different test case generation algorithms.

To show the effectiveness of this integrated approach, we designed an experiment to evaluate a family of test suites generated using one test case generation algorithm, which exhaustively enumerates every sequence of method calls within a certain bound. The results show over 80% code coverage and high mutation coverage for the most comprehensive test suite generated.
## Contents

1 Motivation 1
   1.1 Introduction ................................................. 1
   1.2 Contributions ............................................... 4
   1.3 Organization ................................................... 5

2 Background 6
   2.1 The Testing Process .......................................... 6
   2.2 Unit Testing ...................................................... 8
   2.3 Generation and Evaluation of Test Cases ......................... 8
      2.3.1 Test Case Generation ................................. 10
      2.3.2 Test Case Evaluation .................................. 11
   2.4 The Oracle Problem ............................................ 11
   2.5 An Integrated Approach to Automated Testing ................. 13

3 The Sulu Programming Language 15
   3.1 A Simple Program .............................................. 16
   3.2 Specifying a Stack Component ............................... 20
   3.3 Implementing the Stack ....................................... 23
   3.4 Using and Testing the LinkedList Stack .................. 27
   3.5 Putting It Together: A Scenario ............................ 30

4 The Sulu Runtime and Tools 38
List of Figures

1.1 A vision for the integration of automated testing in the software development lifecycle. .................................................. 2

3.1 A program to display the fibonacci sequence . ........................................... 17

3.2 A Stack concept .................................................................................. 21

3.3 A LinkedList realization of the Stack concept ........................................... 24

3.4 An abstraction function for the LinkedList realization. ................................ 26

3.5 Using a Stack component ...................................................................... 27

3.6 Testing a LinkedList Stack component ...................................................... 29

3.7 Pedro’s specify-code-test cycle .................................................................. 31

3.8 A test execution tool runs tests, and the programmer browses the results in a viewer. ................................................................. 32

3.9 Test case failures require the programmer to fix the bugs that resulted in the failure. ............................................................................. 33

3.10 If the built-in measures of test thoroughness is inadequate, the developer may choose to build his own. ................................................................. 34

3.11 Even when all tests pass, these tests may not be thorough enough, and thus require more test cases. ........................................................................ 35

3.12 The programmer is done with unit-testing when he achieves his chosen adequacy criteria. ................................................................................. 36

4.1 The Sulu tools for automated unit testing ...................................................... 39

4.2 A “flow graph” for a stack component ............................................................ 43

4.3 A GUI for running tests and reporting code coverage ........................................ 45
## List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.1</td>
<td>Mutant detection JML-JUnit vs randomly executing two methods in sequence</td>
<td>51</td>
</tr>
<tr>
<td></td>
<td>as reported in [70]</td>
<td></td>
</tr>
<tr>
<td>5.2</td>
<td>Sulu reference components used for evaluation</td>
<td>54</td>
</tr>
<tr>
<td>5.3</td>
<td>Number of tests generated for each software component</td>
<td>56</td>
</tr>
<tr>
<td>5.4</td>
<td>Percent of invalid test cases in each test suite</td>
<td>57</td>
</tr>
<tr>
<td>5.5</td>
<td>Code Coverage information for Sulu components</td>
<td>59</td>
</tr>
<tr>
<td>5.6</td>
<td>Number of mutants generated for each component and mutation operator</td>
<td>60</td>
</tr>
<tr>
<td>5.7</td>
<td>Mutation coverage information for all singles with one parameter</td>
<td>64</td>
</tr>
<tr>
<td>5.8</td>
<td>Percent of code covered for each test suite</td>
<td>65</td>
</tr>
<tr>
<td>5.9</td>
<td>Aggregate mutation coverage information for each test suite</td>
<td>65</td>
</tr>
<tr>
<td>5.10</td>
<td>Tukey’s test results for each comparison metric; numbers on right of tables</td>
<td>71</td>
</tr>
<tr>
<td></td>
<td>are different at $\alpha = 0.05$</td>
<td></td>
</tr>
<tr>
<td>5.11</td>
<td>Percent covered: Singles$^2$ versus Pairs$^1$ and Triples$^1$; starred values</td>
<td>72</td>
</tr>
<tr>
<td></td>
<td>indicate significant difference against Singles$^2$ at $\alpha = 0.05$</td>
<td></td>
</tr>
<tr>
<td>5.12</td>
<td>Percent covered: Triples$^1$ versus Pairs$^2$; starred values indicate</td>
<td>73</td>
</tr>
<tr>
<td></td>
<td>measures where Pairs$^2$ is significantly better at $\alpha = 0.05$</td>
<td></td>
</tr>
<tr>
<td>5.13</td>
<td>P-values for effects of test suite, size, and their interaction; values less</td>
<td>74</td>
</tr>
<tr>
<td></td>
<td>than 0.05 are considered significant</td>
<td></td>
</tr>
</tbody>
</table>
Chapter 1

Motivation

1.1 Introduction

Testing is an integral part of the modern programming discipline. While testing will not guarantee that a program is correct, it can detect bugs that are not typically detectable by a compiler. Unit testing, especially, is recognized as a useful and pragmatic tool that increases the quality of code.

However, writing and executing tests can be a tedious as well as expensive task. A 2002 NIST report puts the annual cost of inadequate software testing infrastructure as between $22.2 to $59.5 billion [2]. On a per project basis, the cost of software testing is often estimated to be between 30 and 50 percent of the total cost of building new software [14, 44]. Clearly, more automation and better tools for software testing can lower the cost of software development, increase the reliability of software, and reduce the negative economic impact of defective software.

Many parts of the testing process can be automated, but the job of writing test cases often remains in the hands of the software development professional. We assert that testing, particularly unit testing, can be more fully automated; that the work of generating, executing,
and evaluation of unit test cases can be integrated and automated into the programming task, leaving the programmer to write test cases only for the most difficult-to-detect bugs.

Thus, this dissertation asks: can the use of automated testing that is fully integrated into a programming language and its runtime be effective in increasing the quality of code and reducing bugs? Can we add an effective layer of automated error detection beyond that of a traditional compiler?

Automated testing is sometimes taken to mean that there is a program which when given test cases, can run all the tests at the push of a button—that is, test execution automation [15]. However, in this dissertation, the term “automated testing” encompasses the whole process of automatically generating test cases, the execution of these test cases, the classification of software units having passed or failed their test cases, and the evaluation of the thoroughness of the tests.

We envision the automatic generation, execution, and evaluation of baseline test case as becoming an integral part of the software development lifecycle. Figure 1.1 illustrates this vision. To help answer the research question, we built a suite of software tools that include:

- A programming language called Sulu that supports design-by-contract style specifications.
• An interpreter and automated unit testing system for Sulu with a pluggable architecture where automated test case generators, software components, and mutation analysis operators can be separately developed and plugged in.

• A fully integrated approach to automated (and manual) testing, including a framework for evaluating the strength of test cases.

• A test case generator that can exhaustively generate all sequences of \( n \) methods, with \( p \) parameter value inputs.

• An extensible mutation analysis tool with 6 common mutation operators implemented.

These tools constitute a proof-of-concept system that integrates automated testing into the software development process. Using these tools (and a text editor) a set of ten reference software components were developed. These components represent some of the most common data structures and algorithms, three of which correspond to java.util classes.

An automatic test case generator was also implemented; and an experiment to evaluate the effectiveness of test suites produced by the test-cases generator was conducted, evaluating them by running the test suites against our reference components, and using various metrics that estimate the defect detecting capability of these test case generators.

In a paper on the future of software engineering, Bertolino [16] outlines the dreams and challenges of research in software testing. The research in this document is squarely in her category of a dream of 100% automated testing. Although this research does not seek to replace manual testing, we attempt to augment the test efforts of the human tester with automatic tests. This research touches on several of Bertolino’s key challenges: determining test effectiveness, the use of test oracles, and the generation of test inputs. We apply our approach to all three challenges into an integrated, and automated unit testing system.

In this document, we hope to present to the reader a unified view of software unit testing: where automation is possible at all stages—from generation, to execution, to the evaluation
of the test effort; where automated testing is a complementary activity to manual testing; and where the effect on unit testing is an integral part of language design. We believe that the answer to our motivating problem is in the affirmative: that we can create a system of automated unit testing and provide an additional, effective layer of error-checking.

1.2 Contributions

Imagine a programmer in the not so distant future: what is the process by which he writes software such that it is of the highest quality? How will software testing, this venerable and basic (but essential!) step in ensuring software quality, figure in this process? This dissertation offers one vision of this process. It is a vision where it is essential to have an integrated programming system, where the programming language and its tools work seamlessly to provide a platform for automated testing.

We demonstrate the practical effectiveness of this vision by designing a programming language, and implementing its runtime and tools. We show through an experiment that by using this language and its tools to write software in the manner that we envision, we can achieve good confidence in the quality of our code. To reiterate, this research:

- Provides a vision of automated unit testing, where testing is integrated into the programming language—its design, runtime, and tools.
- Implements a proof-of-concept system with a programming language, interpreter, and tools for the automated testing system.
- Employs a plug-in architecture for key parts of the automated testing system.
- Performed an experimental evaluation of several test suites from our automated test-case generation strategy.
- Shows the effectiveness of our approach.
1.3 Organization

The rest of this dissertation is organized as follows: Chapter 2 surveys the landscape of software testing research, and where this particular dissertation lies. Chapter 3 presents the Sulu programming language, and a scenario of how we envision Sulu is to be used. More details of the interpreter and the automated testing tools is shown in Chapter 4. Chapter 5 describes how we evaluated the automated testing strategy in Sulu, but also presents an earlier experiment with JMLUnit. We lay the foundations in Chapter 6 for extending Sulu’s specification language to handle specifying method sequences. Finally, we end in Chapter 7 with concluding remarks.
Chapter 2

Background

Software construction is an error-prone activity. Thus, in the process of software development, there are activities that focus mainly on the construction of the software, and activities that focus on ensuring the acceptable quality of the software being constructed. These quality related activities are often called the quality process.

The quality process is intertwined into the whole software development lifecycle, from the gathering of requirements to product deployment and maintenance. Software testing is part of this quality process. Its aim is twofold: detect defects in the software being developed; and when no defects are revealed, increase confidence in the quality of the software under test. In software testing, faults are revealed by comparing the behavior of the software in a test case against a specified expected behavior.

2.1 The Testing Process

Software testing in various stages of the development lifecycle constitute three parts: selection or generation of specific test cases, execution of these test cases, and evaluation of not only
the quality of the software under test but also of the test cases themselves. That is, the test effort also needs to be evaluated for its thoroughness.

Test case generation involves selecting a particular set of test cases (a test suite) within an often practically infinite domain of program execution (In 1987, Dijkstra estimated that it would take 10,000 years to test integer multiplication exhaustively). Because the domain of all possible test cases is practically infinite, judicious selection of test cases is important. Various mechanisms for systematically generating test cases with different selection criteria have been proposed, but test case generation is still often left to the programmer.

It is no surprise that test case execution, being the most amenable to automation, has the most sophisticated automation tools available. JUnit [12] is perhaps the most popular example of a test case execution tool. If test cases are to be executed for software modules that have dependencies on other modules however, it is often necessary to develop scaffolding to simulate the environment in which the software under test is running.

When test cases are executed, it should achieve the twofold goal of finding defects, and increasing confidence in the quality of the software under test. To detect defects, it must be possible to compare the state of the computation after a test case is run with a specified expected state. Often this comparison is done by consulting an oracle—a software artifact that decides whether a test case has passed or failed. Oracles themselves are often either manually constructed, or automatically derived from a software system’s specifications.

Even if no defects were found during testing, however, no guarantee can be made that the software under test is defect-free. That testing can show the presence of bugs but not their absence was famously observed by Dijsktra [35, 36]. However, we can have some metrics that give a sense of the defect revealing capabilities of our test suite. Various program coverage metrics have been traditionally used for this. Statement coverage, for example measures the number of statements executed in the software under test by running the test suite. Statement and branch coverage have been used to estimate the thoroughness of tests since at least the late 60’s [75].
2.2 Unit Testing

Software testing happens at several conceptual levels; at the innermost layer is unit testing. Unit testing, the subject of this research, concerns itself with testing a single software unit; the software unit could be a function or procedure, a module, a collection of modules, or an individual program [1]. The software unit is typically part of a larger system. In object-oriented languages, there is a general consensus that the ideal software unit for testing is a class [17, 45], since it is a natural encapsulation boundary. In this research, we employ this notion of the class (or its equivalent in Sulu; for historical reasons and to avoid ambiguity—see Section 3.3—we will in this dissertation call this a “component,” or a “software unit”) as the unit to be tested.

Unit testing is currently being popularized by Test-Driven Development (TDD) [12, 13], a process advocated by Extreme Programming and other Agile software development methodologies. In Test-Driven Development, unit tests are written before the software unit (although Beizer [14] cites Gelperin and Hetzel [48] advocating a “test, then code” approach as far back as 1987). By writing tests first, the unit tests also document use cases, guiding the design of the software component, and often serve as a replacement for its formal and informal specification.

Other levels of testing such as regression testing, integration testing, and system testing, while worthy subjects of study, are not the focus of this research; although we note that other levels of testing often rely on running unit tests as an underlying process.

2.3 Generation and Evaluation of Test Cases

For any particular software unit, the question of what tests we should generate, and how we should evaluate the thoroughness of the test cases generated are of key importance. Different test adequacy criteria are used to answer both questions. Zhu et al. [92] differentiates
three notions of adequacy criteria: as a stopping rule, a measurement of test quality, and a
generator for test cases.

When test cases are generated manually, a single test adequacy criterion can be used to
guide both the selection of the test cases, and evaluation of the test effort. For example, the
statement coverage criteria may be used to show that certain statements are not executed,
and thus a test case must be written to exercise those statements. A test suite is then deemed
sufficiently thorough when all statements are covered.

When test cases are automatically generated, adequacy criteria can similarly be used to
generate test cases. Often, as in the case of the test case generators used in this research,
test cases are generated to exhaustively satisfy the criteria used. However, the question
remains of whether the generated test cases are thorough enough to reveal real bugs. One
way to measure the quality of automatically generated test cases is to collect real bugs from
production software and count the number of bugs found. However, given the difficulty of
finding bugs “in the wild” surrogate measures are often used.

From a practical standpoint, analyzing the power of a test suite in the above manner is only
a retrospective measure; that is it can tell a tester how many of previously known bugs were
discovered, but may not be indicative of how much of the unknown bugs the test suite has
missed. Thus, automatically generated (and manually written) test cases are often measured
against other adequacy criteria such a code coverage and mutation coverage.

The generation and evaluation of unit tests can generally be categorized into two classes:
black-box testing, and white-box testing. Black-box testing treats the software unit as the
eponymous “black box” where the internal implementation of the software unit is ignored,
and what is evaluated is instead its specified behavior; thus this kind of testing is often called
behavioral, or functional testing. White box or structural testing, on the other hand takes
into consideration the structure of the software under test: its statements, conditionals, etc.\[53\]. In this research, we implemented the generation and classification (whether the test
case passed or failed) of tests as black-box based; and the evaluation of the test effort (how thorough the test suites are) are white-box based.

2.3.1 Test Case Generation

The testing tools detailed in this dissertation includes an extensible framework to add different automated test case generators. Various different automated test case generation strategies have been proposed in the literature. While traditionally viewed as inferior to systematic techniques of test case generation, variations on random testing [27, 62] are still a popular research approach to generating test cases. Clever hybrid approaches such as DART [49] and RANDOOP [73] augment simple random input selection with model-based approaches, and appear to succeed in creating effective test cases.

One alternative to random testing is model-based testing, where test case generation is directed by a formal model of the software component. This mechanism of taking formal specifications and generating test cases from them is used by the ASTOOT [37] and DAISTS [46], approaches to automated testing. The Korat [18] system exhaustively explores every configuration of an object’s state variables (within a bound) and uses those as the basis for tests. Cheon and Leavens’ JMLUnit [29] call every method of an object, passing in the cross product of parameter examples. Evolutionary algorithms have also been applied [28, 83, 84] to explore the space of possible test cases.

The strategy used in the generators provided in Sulu adopts the model-based approach by performing a systematic enumeration of paths in an object’s flowgraph model, derived from the mechanism proposed by Edwards [39, 40] which in turn was developed from the method described by Zweben and Heym [93]. Instead of exhaustively exploring an object’s state variables, we explore every sequence of method calls, providing for parameter values similar to JMLUnit. We shall explore this mechanism more thoroughly in Section 4.2. A family of such generators is provided by the author and used for evaluation.
2.3.2 Test Case Evaluation

Once a test suite is executed, it is imperative that the tester evaluate the thoroughness of the test suite. The traditional test-adequacy criteria for this evaluation is code coverage. Indeed, the IEEE standard for software unit testing [1] specifies complete statement coverage as a minimum requirement for unit tests. Statement and branch (i.e., decision) coverage has been in popular use since at least the early 70’s [67, 75]. It is quite common to have a rule of thumb of between 80-90% code coverage for testing of commercial-quality software [32, 89]; A conversation with a testing engineer revealed that Microsoft has a company-wide bar of 75%, with higher code coverage bars for some individual testing groups. Sulu profiles three different code coverage metrics: statement, decision, and condition-decision [54] coverage.

Mutation analysis, where defects are automatically seeded in otherwise acceptable code can also serve as an approximation of the defect-detecting capabilities of a set of test cases [23]. The earliest language system that used mutation testing was the Mothra [33] project, which implemented mutation analysis tools for Fortran77.

Offut and his colleagues [71] have identified a minimal set of mutation operators that may be sufficient to gauge the quality of test suites. A recent study [6] suggests that mutation coverage using these mutants are a good predictor of a test suite’s strength in detecting real bugs. Sulu’s mutation analysis tool includes six mutation operators, and can be extended by adding new mutation operator plug-ins. We will describe these evaluation criteria in more detail in Section 5.2.3.

2.4 The Oracle Problem

Test cases are useful only if we can execute them, and compare them to the software component’s specified expected behavior. With manually written tests, it may be sufficient for the tester to also manually assert the expected behavior of a component.
As an example, we may want to test a stack component. A possible test case would be to push an item into an empty stack. If we want to know that the stack component behaved correctly, we have to verify that the stack size is one, and the top of the stack is indeed the item we pushed into it. This is a fairly standard unit testing convention: create an object, call one or more of its public methods, and programmatically assert that the object is in the correct state. This is in fact what is typically done in a JUnit test case, and is supported by the Sulu programming language as well.

However, manually writing asserts is infeasible for automatically generated test cases when the number of test cases are large. For example, we generated nearly 70,000 test cases from one of the components implemented in this work. Clearly there is a need for a better mechanism to determine the correct behavior of the software component beyond writing assertions manually for every test case.

The solution is to somehow consult an *oracle*, a software construct that tells you whether the program is in the correct state or not. This problem of how to determine whether the software under test behaved correctly under a test case is often called the *oracle problem* [47].

Sulu uses the runtime checking of its embedded specification language as the test oracle. For software that has an executable specification (i.e. a design-by-contract [65, 66] specification), the specification itself can be used as the oracle. That is, a method’s postcondition is already a specification of what the software component’s state should be after the method is called. Thus a postcondition failure also corresponds to a test failure.

Design-by-contract specifications can also be used as a *test filter*. That is, if a test case causes a precondition of the software under test, it means that the test case exercised the unit in a manner that is not allowed by the specification. Thus a test case is considered *invalid* if it causes a precondition failure. Because of its central role as test oracle, and a test filter, a well-specified software component is an important part of the approach advocated by this thesis.

One of the earliest embedded assertion languages is Anna [64], a specification language for
ADA, that was also used as a test oracle. More recently, the use of DBC specifications as test oracles have been advanced by automated testing work in Eiffel [30, 62], and JML. The Java Modeling Language [60] is a specification language for Java that incorporates runtime checking of specifications; this runtime checking has also been used as a test oracle [27, 29]. JML assertions are often written in relation to the object’s abstract state, rather than its concrete state variables, thus necessitating the implementation of abstracting methods (called model methods in JML) that translates the concrete state of the object into an abstract representation of the object. Jia [55] proposed to use a similar mechanism of abstractors as a mechanism to translate the state of a software component (C++ in his case) to its abstract specification (e.g., \( Z \)), for use as a test oracle. Baresi and Young [9] provide an extensive survey of test oracle mechanisms.

2.5 An Integrated Approach to Automated Testing

Software testing therefore consists of generating test cases, execution of these test cases and comparing it with the expected behavior (e.g. consulting an oracle), and evaluation of these test cases using certain adequacy criteria.

Much of the literature and available tools concern themselves with automating only one or two aspects of this process. With Java, for example, there are disparate tools for test case execution (e.g. JUnit), code coverage metrics (Clover [3]), mutation analysis (Jester [69]), and design-by-contract mechanisms for use as oracles (iContract [57], JML).

JML, the Java Modeling Language, whose specifications can be used as an oracle for test cases, and it includes a tool, JMLUnit [29], which integrates an automated test case generator with JML as an oracle and JUnit as a test case execution mechanism. We will describe an evaluation of JMLUnit in Section 5.1.

Subsequent research has enhanced the tools available for automated testing in JML [27]. However, even with these tools, the programmer does not have the adequate infrastructure
to evaluate the effectiveness of the automated testing mechanism without access to other tools. That is, given two automatically generated test suites, how can we decide which one is more effective? For example, in Cheon’s work, evaluation of their testing strategy involved hand-seeding mutants, instead of automatically generating them.

Autotest \cite{30, 62} is a similar random testing tool for Eiffel, which appears to collect code coverage (but not mutation coverage) information in addition to automatically generating and executing test cases. In that work, the authors emphasize integration between user-generated tests and automatically generated ones. This is also a central theme to the research contained herein—that the integration of manual tests with automated testing can provide a more thorough evaluation of the software under test.

The research described in this document employs a higher level of integration of automated testing tools, providing an end-to-end integrated approach to automated unit testing. Testing support is designed into the programming language and its runtime system; and tools are available for automatic generation, execution, and evaluation of test cases.
Chapter 3

The Sulu Programming Language

To realize the vision of an integrated platform for automated testing, we designed a programming language called Sulu, implemented an interpreter for it, and constructed automated testing tools that can generate tests automatically, run these tests, and evaluate these tests via code coverage and mutation analysis metrics.

The Sulu programming language is an object-oriented language influenced by the Resolve programming language [80], a project that this author and his adviser have participated in. The goal in the development of Sulu was to facilitate the integration of automated unit testing tools for programs written in this language. In this sense, the Sulu programming language provides a novel outlook, we developed this language with a constant eye towards its effect on unit testing. For example, since we need to have a test oracle to determine whether a test suite found a bug or not, we implemented a design-by-contract specification language as an integral part of the language.

We also considered how best to provide language features that make it possible to implement software components that are easy to test. We found that this goal is not dissimilar to the goals of the Resolve project of making software components that are simple to reason about, both formally and informally. This is not by chance; every test case is also a use case of
that software component—it is often the case that a software component that is easy to use is also easy to test.

Thus, Sulu attempts to follow the principles espoused by the Resolve discipline, including those listed by Weide [85]:

- Separation of specifications from implementation
- Allowing for multiple interchangeable implementations of a single specification
- Having a standard set of “horizontal”, general-purpose, domain-independent components such as lists, trees, maps, etc.
- Having templates as a useful composition mechanism
- Having value semantics even for user-defined types
- Minimizing reasoning about programs that use pointers/references

The key features of the Sulu programming language include syntactic slots for design-by-contract specifications, generic templates, alternative data movement operators (swapping, one-way transfer) instead of assignment, and self-type support. The most directly related to software testing among these features is the design-by-contract support. As discussed in Section 2.4, we use the specifications as an oracle for determining whether a program is in the correct state or not. This is therefore the one feature required by our approach.

To better understand the Sulu programming language, let us look at some examples.

### 3.1 A Simple Program

Figure 3.1 is a simple example of a Sulu program that prints out the first 10 numbers of the Fibonacci sequence. At first glance this is fairly standard notation; the goal is to have
//This program prints the first 10 fibonacci numbers.
var current: Int;
var previous: Int;
var index: Int;

var c: Console;

current := 1;
previous := 0;

while( index < 10 ) {
    c.println(current);
    c.print(" ");

    var tmp: Int;
    tmp := current + previous;
    previous := current.clone();
    current := tmp.clone();
    index := index + 1;
}
a familiar syntax at the statement level. Variable declaration is Pascal-like, as well as the assignment operator; the while loop is C-like.

There are some details we must note—one would observe, for example, that the while loop on line 11 of Figure 3.1 should be a for loop instead. However, special syntax for counter-controlled loops are not included in the current version of Sulu, which was decision to cut down work on the implementation of the interpreter. This is a minor detail, however, since every for loop can be expressed as a while loop; and there is no barrier to adding this syntactic feature in future versions of Sulu.

Line 18 is a more fundamental departure from current programming practice:

```plaintext
previous := current.clone();
```

One would normally think this should simply be previous := current; however, in Sulu, direct variable assignment is disallowed.

Today’s popular object-oriented systems languages such as Java and C# differentiate between scalar variables, and object variables. Scalar variables represent built-in types like integers and characters, and have value semantics. Assigning one scalar variable to another creates a copy of the first variable’s value. Object variables represent user-defined types, and have pointer semantics. Assigning one object variable to another creates an alias of to the object referred to by the first variable.

In Sulu, there is no such dichotomy. Every variable represents an object, but have value semantics. Assignment from one variable to another is explicitly disallowed. Instead, assignment is only permitted if one variable is assigned to the result of a method call. Methods are also required to return new objects. Thus, two variables are always two different objects, and never aliases of each other (in Chapter 5 we will relax this rule when we recreate some java.util classes, but these should not be considered idiomatic to Sulu).

Aliasing, and reference or pointer semantics that cause aliasing has been a long-standing
problem with both the writing of formal specifications and in the informal reasoning about programs. Hogg et al. considered this problem in [52]. Weide and Heym [86] look specifically at the difficulty of formally specifying software in the face of references. Kulczycki et al. [59] considered this problem in reference to repeated arguments. In fact, as early as 1973, Hoare commented on references: “their introduction into high-level languages has been a step backward from which we may never recover” [51]. Thus to avoid the problems of references and the aliasing that the can cause, Sulu provides three alternative mechanisms for data transfer: deep copy, swapping, and clearing transfer.

If we cannot assign one variable to another directly via pointer copying, how can we give one variable the same value that is previously held by another? In the Fibonacci example, we have one solution: create a clone of the object by calling a clone method. Since Sulu methods must return new objects, clones have to be deep copies. However, making deep copies can be an expensive operation, especially for complex objects.

Sulu therefore supports two data-movement operators as alternatives to aliasing assignment: a swap (:=;) operator, and a clearing transfer (<<) operator. Harms and Weide originally proposed swapping as an efficient non-aliasing alternative to assignment [50, 88]. A swap statement like this:

```
a :=: b;
```

means that after the statement is executed, a gets the old value of b, while b gets the old value of a. The swap statement can be implemented as a constant time operation using pointers internally, and this is how it is done in the Sulu interpreter.

Minsky’s [68] unshareable pointers and Baker’s [7] linear type introduced a different alternative, a destructive read operation that whenever a variable is read, that variable takes on a null value. Sulu’s clearing transfer operation is similar except that for a clearing transfer operator like this:

```
a << b;
```
After the statement, a gets the old value of b, but b is set to a valid initial state. This data movement operator can also be efficiently implemented if either the initial state of every object is easy to construct, or lazy initialization is applied such that b is only actually initialized when b is next used (we should note that Sulu does not currently employ lazy initialization). Swapping and clearing transfer are also the main data movement operators of the Tako programming language [58], which is part of a project to support value semantics in Java.

Thus far, we have only seen a one-off main program to compute a fibonacci sequence in Figure 3.1. While it is possible to write one long main program in Sulu, it is rather designed for creating reusable and automatically testable software components. How do we create a reusable and testable software component? First, we define a formal specification for it.

### 3.2 Specifying a Stack Component

In this section we look at how we can specify a software component. In Sulu, as with Resolve, there is a separate module used to define the specification of a software component as opposed to its implementation. Sulu specifications are placed in a syntactic unit called a concept. A Sulu concept is similar to a Java interface in that they define the public methods that all implementations of the concept or interface must realize. However, concepts also have additional syntactic slots to put in a design-by-contract formal specification.

Figure 3.2 is the Sulu code for a Stack concept, slightly modified for clarity and brevity. A stack is a simple component with three publicly accessible methods: push, pop, and length, with the usual semantics, although pop is slightly unusual in that the top of the stack is removed and placed in the element parameter. This actually exposes one other feature of Sulu: all parameters are in/out.

An English description is not enough, however, to specify the Stack concept—our test execution framework needs a formal artifact to use as a test oracle. Sulu uses a style of
concept Stack( Item ) {

    initially this.getSequence().length() == 0;

    method push( element: Item )
        ensures this.getSequence() ==
            old( this.getSequence().insertFirst(element.getMathObject()) );

    method pop( element: Item )
        requires this.getSequence().length() > 0
        ensures old( this.getSequence() ) ==
            this.getSequence().insertFirst( element.getMathObject() );

    method length(): Int
        ensures length.equals( this.getSequence().length() );

    model method getSequence(): concept Sequence(concept MathObject() );

}

Figure 3.2: A Stack concept
specification that is influenced by both Resolve and JML. Particularly, it adopts JML’s runtime assertion checking mechanism to allow it to execute the specifications.

The specification for the stack here is fairly standard model-based one. We model a stack component as a mathematical sequence of items, with (the math model of) the top of the stack as the first element of the sequence. The initially section on line 3 provides the initial state of the model: an empty sequence.

After every method signature, one can put in design-by-contract preconditions in a requires clause, and postconditions in an ensures clause. The contract for the pop method on line 9, for example, requires that the sequence is not empty when the method is called, and ensures that the sequence in the pre-state (the old sequence) is the same as the sequence in the post-state with the popped element added to the beginning.

The old(...) construct is used to denote that the expression encapsulated between the parentheses is evaluated in the state before the method is called. Note that this is not a function call. All procedures in Sulu are attached to objects as methods. Although not shown in the figure, Sulu also supports class invariants with an invariant section.

A property of these specifications is that they can be executed. When we model a stack as a sequence of items, it is possible for us to actually construct a programmatic sequence of items such that we can check, for example that the pop method actually decreases the length of the sequence by one, and produces the right object.

Of course, every implementation of Stack may have a different way of constructing the sequence needed by the pre- and postconditions. A stack implemented using arrays, for example, would not construct this sequence of items the same way as a stack implemented using a linked list. Thus, we provide instead a model method called getSequence. Like push, pop, and length, programmers are required to realize that method for every implementation of the Stack concept. The model keyword indicates that this method is going to be used for specification purposes only.
The model method `getSequence` is an example of an abstraction function, it converts the specific internal representation of a data type into an abstract mathematical (but also executable) model that the specifications can manipulate. The sequence returned by `getSequence` represents an abstract state of the `Stack` component.

Sulu provides several “math” components such as `Set`, `MultiSet` and `Sequence`, to serve as the basic types used to represent the abstract state of a software unit. Developers may also build their own mathematical components for more complicated abstract states if they so wish. Math components are intended for use only in specifications, and do not get used to actually implement the software unit.

### 3.3 Implementing the Stack

With a `Stack` concept at hand, we are now ready to implement a realization; a realization in Sulu is analogous to a Java class that implements an interface. However, in Sulu, all software components other than the main program must have both a concept, and at least one realization. Figure 3.3 is one such realization for the `Stack` concept.

A stack may be implemented in several different ways, probably the two most common implementations are: using an array which is resized as the stack grows; or as in the example in Figure 3.3 using a singly-linked list, with a pointer to the top of the list.

This implementation uses a couple of other components that might be unfamiliar to the reader. One is the `Pair` component, which, as its name implies, is simply a generic component that is a pair of two objects; each object can be accessed by the `swapFirst` and the `swapSecond` methods. The `class` notation also deserves special mention—in Sulu, a class is an entity that is fully concrete and actualized. That is, it is a unit where we know the concept and the realization of the software component, and where all the template parameters have actual types. On line 5 in Figure 3.3 we created a `Node` class which has the public specification of the `Pair` concept, and realized by the `Obvious` realization. Furthermore, the
realization LinkedList() implements Stack(Item) {

    // A linked-list node is a pair containing the item to store and a
    // pointer to another node.
    class Node extends
        concept Pair(Item,
            concept ChainPointer(Node) realization Builtin()
        )
    realization Obvious();

    var top: concept ChainPointer(Node) realization Builtin();
    var count: Int;

    method push(element: Item) {
        var newNode: Node;
        newNode.swapFirst(element);
        newNode.swapSecond(top);

        var next: concept ChainPointer(Node) realization Builtin();
        next.swapEntry(newNode);

        top := next.clone();
        count := count + 1;
    }

    method pop(element: Item) {
        var topNode: Node;
        top.swapEntry(topNode);

        topNode.swapFirst(element);
        topNode.swapSecond(top);

        count := count - 1;
    }

    method length(): Int {
        length := count.clone();
    }

    ...

Figure 3.3: A LinkedList realization of the Stack concept
types of the two objects stored in the **Node** class is the item that is to be stored in the stack, and a pointer to the next node.

The **ChainPointer** concept might also be unfamiliar; it is basically a pointer component that does not allow cycles. This pointer component allows developers to create data structures such as linked lists, tail sharing lists, and other directed acyclic structures.

While some of these implementation details might be unfamiliar to the reader, the implementation is actually fairly standard. The **push** method creates a new node, with the old top node as its next node, and sets the top pointer to the new head of the list. The **pop** method sets the **element** parameter to the value stored in the head node, and sets the top pointer to the next node. Both **push** and **pop** maintain the count of the number of elements in the stack, which can be accessed via the **length** method.

One method that is not shown in Figure 3.3 is the abstraction function **getSequence**. For reference, **getSequence** is listed as Figure 3.4. The **getSequence** method walks through the nodes in the linked list, calls the abstraction function **getMathObject** of each of the values in the list, and constructs a **Sequence** component that represents a mathematical sequence of items. Because our **Stack** concept in Figure 3.2 is specified in terms of the **getSequence** abstract value, the abstraction function is a crucial part of our testing framework. A programmatic representation of a sequence of items allows us to execute operations on the sequence to tell us whether the object is in the correct abstract state when a method is called.

The reader might be concerned that creating a stack component seems quite involved for what seems to be simple operations. We should note that a pointer component is a fairly low-level construct in Sulu. The philosophy in Sulu— inherited from Resolve—is that we should layer software components on top of each other in such a way that we rarely need these low-level constructs. As you might imagine, creating a linked-list based stack component can be complex, but using it should be simple.
model method getSequence(): concept Sequence(concept MathObject())
{
    var sequence: concept Sequence( concept MathObject() )
        realization LinkedList();

    var nodePtr: concept ChainPointer(Node) realization Builtin();
    var nextPtr: concept ChainPointer(Node) realization Builtin();

    var tmpNode: Node;
    var tmp: Item;

    var i: Int;
    i := 0;

    nodePtr := top.clone();

    while(i < count) {
        nodePtr.swapEntry(tmpNode);
        tmpNode.swapFirst(tmp);
        tmpNode.swapSecond(nextPtr);

        sequence.mutInsert( tmp.getMathObject(), sequence.length() );

        tmpNode.swapFirst(tmp);
        tmpNode.swapSecond( nextPtr.clone() );
        nodePtr.swapEntry(tmpNode);

        nodePtr :=: nextPtr;
        i := i + 1;
    }
    getSequence << sequence;
}

Figure 3.4: An abstraction function for the LinkedList realization.
```java
class StringStack extends concept Stack(String)
    realization LinkedList();

var stack: StringStack;

stack.push("Hello");
stack.push("World!");

var c: Console;
var str: String;

stack.pop( str );
c.println( str ); //should print "World!"

stack.pop( str );
c.println( str ); //should print "Hello"

stack.pop( str ); //this is a precondition violation
```

Figure 3.5: Using a Stack component

### 3.4 Using and Testing the LinkedList Stack

Once we have the Stack concept, and a proper realization, we are now ready to use it. Figure 3.5 is a version of the “hello world” program that uses a stack of strings. The author hopes that the figure illustrates that once the stack component is implemented, it is easy to use.

The last line is a pop call on an empty stack. This is a precondition failure, as the Stack concept requires that the stack must not be empty before a pop call. With runtime assertion checking turned on (by default), this will cause a precondition failure: the program terminates, and the precondition that failed is printed out on the console. If assertion checking is turned off, the precondition violation will not be detected, and the component will behave in an arbitrary unspecified manner. The contract with the caller is broken, and thus anything
may ensue (in this case, a Java null pointer exception will be caught by the Sulu interpreter, and the program will terminate anyway).

If a postcondition failure occurs at any time in the execution of the program, this means that the stack (or one of the component it uses) contains a bug and did not fulfill the contract specified in the postcondition. When running a program normally, a postcondition failure in Sulu results in the termination of the program. However, a testing tool can take advantage of the result of postconditions to determine whether a component behaved correctly or not.

While Sulu can generate tests automatically, it allows users to write their own tests and run it separately or along with the automated tests. This is part of the integrated approach taken in this thesis: automated test case generation can systematically generate test cases, often more test cases than a programmer can manually write. This provides a baseline level of error checking. In contrast to an automated test generator, a programmer has a limited capacity for writing tests, but has more knowledge of the problem domain, and so he should be given the task of crafting test cases that test the often complex behavior not covered by the test generator.

Figure 3.6 is an example test for our linked-list stack component. This component looks very much like a JUnit test class. In fact, this realization works nearly identically to a JUnit test class. First, we note that our realization implements ComponentTester. All components that implement ComponentTester are treated specially by the Sulu tools. Similar to JUnit, a Sulu test runner will run every method that begins with test, and will call setup before every test method is run, and tearDown after every method is executed.

The testPush method differs from normal JUnit methods in that it does not have any assertions that provide the expected results. Instead, we rely on the automatic specification checking mechanism as a test oracle. If a postcondition failure occurs, then there is a bug. A postcondition failure means that there is an inconsistency between the specification of a software unit, and its implementation. The bug could be in either one, or both (this is also
realization Stack_LinkedList_Test() implements ComponentTester() {

    class ComponentUnderTest extends
    concept Stack(String)
        realization LinkedList();

    method setup() { /* do nothing */ }

    method tearDown() { /* do nothing */ }

    method testPush() {
        var stack: ComponentUnderTest;
        //we rely on runtime specification checking
        //to detect bugs here:
        stack.push("hello");
    }

    method testPushThenPop() {
        var stack: ComponentUnderTest;
        var s: String;

        stack.push("hello");
        stack.pop(s);

        var asserter: Asserter;
        asserter.isTrue( s == "hello");
        asserter.isTrue( stack.length() == 0);
    }
}

Figure 3.6: Testing a LinkedList Stack component
a problem with manually written assertions, the bug may lie with the software under test, or it may lie with the incorrect expected value of the assertion).

However, manually placing asserts is also supported by Sulu. A special Asserter component is used to assert the correctness of the program, and cause an assertion failure if it is not correct. Lines 25 to 27 of Figure 3.6 shows how the Asserter is used. Allowing testers to use manual asserts lets the programmer write test cases for components that do not have specifications. It also allows programmers to test under-specified components more thoroughly.

When a test for a fully-specified component causes an assertion failure without causing a postcondition failure, it indicates that the postcondition specifies that a certain illegal post-state is allowed; which in all likelihood means that the postcondition is not strong enough.

We provide a command line tool, as well as a GUI tool that can run manually written tests, these are discussed more thoroughly in section 4.4. Automatically generated test cases also follow the same format as user-written tests—i.e., they are methods that begin with test in realizations that implement the ComponentTester concept.

3.5 Putting It Together: A Scenario

How might a programmer with access to a programming language such as Sulu and its automated testing tool develop software differently? In this section we show one scenario of how a programmer might develop a hypothetical software component using Sulu. Imagine a programmer (let’s call him Pedro) who wants to build a hypothetical software component called a Sprocket.

Figure 3.7 illustrates how Pedro codes his Sprocket component. This cycle is analogous to the usual test-and-code cycle, but with the additional specification step. By specification, we mean here the writing of formal, design-by-contract specifications for the Sprocket. In
this specify-code-test cycle, Pedro may begin from any point. He may start by writing a full formal specification first; or, from a mental model of how things are supposed to work, start by coding first; or if Pedro is used to Test-driven-development, he may start by writing tests first, representing use cases for his software component. One imagines Pedro working incrementally, writing some specs, some tests, and some code, and then going back through the cycle until the software unit is completed.

In creating unit tests, Pedro can write his own tests, but also choose to generate unit tests from an automated test case generator. There are a couple of test case generators implemented already in Sulu based on the flowgraph model (we shall discuss this in the next chapter); but we imagine that there can be a whole ecology of test cases generators, each with their strengths and weaknesses. Pedro can choose one of these generators and use the test cases generated from them as a baseline set of test cases. In fact, the automation can be pushed further in that the testing system might be running test suites generated by various test generators and reporting to Pedro the test suites that are most effective. If Pedro is
unhappy with his choice of test generators, he also has the option of writing his own test case generator—perhaps one that is suited for objects such as Sprockets.

At some point in Pedro’s development cycle, he will be ready to run his first test. With the current Sulu system, this means Pedro invokes a test execution tool; but in the scenario where the testing system executes tests in the background as the programmer develops his software component, the tests may have already been executed for Pedro. In this case, Pedro merely has to open a viewer for the tests that have already been executed. Using this viewer, Pedro can gather two critical pieces of information: a measure of how well his components stood up to the tests—i.e., how many tests passed or failed; and a measure of how well his tests exercised his software component—measures of code and mutation coverage. Figure 3.8 illustrates this, using a test execution tool already available with Sulu.

Figure 3.9 shows Pedro’s viewer after running the initial tests. It shows that there is one test case that failed. If there are failed test cases, Pedro must return to his code, specifications, or tests to identify where the bug may lie. The bug may be in one of the three, or perhaps in
Figure 3.9: Test case failures require the programmer to fix the bugs that resulted in the failure.
all of them. Even if our intrepid programmer fixes all the bugs such that all his tests pass, he may not be done. That is, Pedro also needs to look at the thoroughness of his tests.

Sulu provides several measures of thoroughness. Currently, we can measure thoroughness on three different code coverage criteria, and six mutation coverage criteria. However, if Pedro feels that these measures are inadequate, he can also develop his own mutation operation, which can then be seamlessly integrated into the programming tools (Figure 3.10).

To be able to say he is done, Pedro needs to achieve a certain adequacy coverage bar. This may be 100% statement coverage, or some other criteria that Pedro set for himself, or that his organization requires. Let us imagine that Pedro sets himself a test adequacy bar of 100% code coverage. Figure 3.11 depicts a scenario where all Pedro’s tests pass, but that code coverage is inadequate—only half of all decisions are covered.

Because Pedro’s tests are not enough to satisfy his code coverage criteria, Pedro must go back and generate more tests. He may generate more tests by writing the extra test cases required to cover the gaps in his existing test suites, or he may choose to use a different test
Figure 3.11: Even when all tests pass, these tests may not be thorough enough, and thus require more test cases.
Figure 3.12: The programmer is done with unit-testing when he achieves his chosen adequacy
criteria.

case generator to generate test cases and augment the test cases he already has. Figure 3.12
shows that by generating more tests, Pedro achieves 100% code coverage. Since Pedro’s test
suites satisfies the criteria he set of himself, Pedro can now say he is done.

This scenario highlights several things which differ from current programming practice. One
is the essential nature of formal specifications. Because we use these specifications as oracles
to determine the correctness of the software component, an automated system along the
lines we advocate in Sulu will be severely hampered. This may pose an additional burden
to programmers, but we note that programmers often are required to provide very specific, if natural-language, descriptions of their software components anyway.

We also see how both manual test cases and automated test cases can work together to provide a more thorough test of the component-under-test. That is, the software testing engineer fills in the gaps and exercises the parts of the software under test where the automatically generated testing system failed to exercise. To be able to determine, however, which parts of the software is under-tested, the programmer needs ready access to a measure of test thoroughness. Thus, manual test case creation, automated test case generation, test execution, and adequacy evaluation need to work hand-in-hand to provide a powerful mechanism to ensure the high quality of software components. With this scenario in mind, we describe in the next chapter more specific details of the Sulu interpreter, and its software testing tools.
Chapter 4

The Sulu Runtime and Tools

Programs written in the Sulu programming language are executed by an interpreter. The interpreter was implemented using Java and the ANTLR parser generator (http://www.antlr.org). Execution of a Sulu program using the interpreter is in three phases: parse, typecheck, and execute. In the first phase, the source file is parsed using a parser generated by ANTLR, resulting in an abstract syntax tree (AST); then, the AST is traversed using an ANTLR tree parser for type checking. During the typechecking phase, the interpreter constructs objects that represent the global environment of the system (concepts, realizations, etc.). The interested reader might consult Section B.3 for more details on how this is done. Finally, during the execution phase, these objects in the runtime environment and the syntax tree is used by the interpreter to execute the actual statements in the program.

In addition to the interpreter, a set of unit testing tools were implemented: a test case generator, a test-case execution tool, and a mutation analysis tool. Because they are tightly integrated into the interpreter, these were also implemented in Java. Figure 4.1 is a diagram of the overall architecture of the Sulu tools for automated unit testing. The only input required by these tools is the component under test, which contains an embedded design-by-contract specification of its own behavioral requirements. Pluggable automatic test case generators then take this information and produce one or more test suites.
Figure 4.1: The Sulu tools for automated unit testing

The test suites generated by the test generation tool are in the same format as manually generated test suites; and thus the test execution tool can execute the automatically generated test suites in exactly the same way it runs the manually written ones. The Sulu interpreter includes a code coverage profiler that collects three code coverage measures: statement, condition, and condition/decision coverage. In addition, a mutation analysis tool is also included. Using the mutation analysis tool, we can generate a set of software components each of which differs from the original in exactly one way (for example, having a plus operator changed to a minus operator). Thus, a test suite may be run against a mutant, if the test suite fails, the mutation is detected by the test suite, and the mutant is said to be killed.

Our mutation analysis tool generates mutants, runs each mutant against a test suite, and reports the number of mutants killed versus the total number of mutants. The mutation analysis tool also supports pluggable mutant generators, so that the tester may add different kinds of mutant generators as needed.

Thus, the Sulu tools for automated testing provides all the necessary infrastructure for
generating, executing, and evaluating test suites, given the specification and implementation of a software component. We believe that taken together, the Sulu programming language and its testing tools allows us to have an effective layer of automated software testing that can reduce bugs and increase the reliability of software.

The tools and the interpreter for the Sulu programming language are available as of this writing at http://sourceforge.net/projects/sulu-lang. The version used in this research will also be available as part of Virginia Tech’s Electronic Thesis and Dissertation submission of this document.

4.1 Test Case Generation

During the typechecking phase, the Sulu interpreter builds a collection of objects representing the global environment of the system, such as objects that represent concepts and realizations. These representations of concepts and realizations contain the concepts’ method interfaces, and the realizations’ method bodies. This information can then be passed on to a test-case generator.

Section 5.3.1 presents evidence that the automated test case generation strategy implemented for our research (outlined in the next section) results in good coverage for many of the test adequacy criteria devised in this research. However, we have to consider the cases where our strategy is inadequate for certain software components, or where better alternative test case generation algorithms are available.

There are of course other algorithms for automatically generating tests. Sulu provides support for implementing these other strategies by having a plug-in architecture. In practice, a person who might want to build a different test case generator needs to implement a simple Java interface called `sulu.generator.TestGenerator`. A command line tool allows the tester to select from different implementations of this interface for generating test suites. The `TestGenerator` implementation that is selected is given access to all the information
available to the interpreter just after the typechecking phase. That is, it has access to the internal representations of concepts, realizations, methods, etc., but also access to the raw AST, if the test case generator wishes.

Given this information, the test case generator is tasked to create a test suite, a file in the format of Figure 3.6. The generated test suite can then be used in the future as the input to our test case execution tool.

The idea of pluggable test case generators is that in practice, different test case generation strategies may be better for different kinds of software components. As an example, we present in this document a test generation strategy that can work well for collection components. But if the programmer is interested in testing, for example, GUI components instead, a different test generation strategy may prove to be better. By providing a plug-in mechanism, the programmer may choose or implement the test generation algorithm he believes is best.

4.2 Using Black-box Flowgraphs to Generate Test Cases

The Sulu architecture supports plugging in different generators that construct test cases of varying strategies. However, the ones implemented for use in this research implements a strategy for black-box testing using flowgraphs.

Edwards [39] presents a strategy of generating test cases using flow graphs which in turn is based on the methodology described by Zweben and Heym [93]. Given a specified component, we build a graph whose paths represent every 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 $v_2$. [39]
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 \( v_1 \). Our automated testing algorithm constructs a modified flowgraph that relaxes this requirement, it has an edge between two vertices if it cannot statically determine that one method call cannot be followed by another. Thus, our test cases generator constructs a flowgraph in this way:

Let \( \text{init} \) be a vertex representing the \text{initialize} operation, and \( \text{finalize} \) be a vertex representing the \text{finalize} operation. Let \( M = \{m_1, m_2, \ldots, m_n\} \) be a set of vertices such that each vertex represents a method in a component \( C \) that is neither \text{initialize} nor \text{finalize}. The flowgraph \( F(C) \) is a directed graph with vertices \( V = M \cup \{\text{init}\} \cup \{\text{finalize}\} \), and edges \( E \) such that: for every vertex \( m \in M \) there is an edge from \( \text{init} \) to \( m \), an edge from \( m \) to \( \text{finalize} \), and an edge from \( m \) to every vertex in \( M \) (including \( m \) itself)\(^1\); and additionally, there is an edge going from \( \text{begin} \) to \( \text{end} \). Figure 4.2, for example, is a flowgraph for a stack component.

Thus, a walk from the \( \text{init} \) vertex to the \( \text{finalize} \) vertex represents a sequence of method calls from object initialization to object finalization, i.e. a possible object lifetime. However, some of these paths may be infeasible. For example, the sequence of method calls represented by \( \text{init} \rightarrow \text{push} \rightarrow \text{pop} \rightarrow \text{pop} \rightarrow \text{finalize} \) for a stack component may be infeasible, because the last pop call violates the method’s precondition.

In theory, the graph in Figure 4.2 should not have an edge from \( \text{push} \) to \( \text{pop} \). This is because there is no object lifetime for which \( \text{pop} \) can be called immediately after it is created. However, from the standpoint of automatically generating flowgraphs, the test case generator does not have sufficient capability to determine this fact. Thus the test case generator assumes that some object value allows the edge to be traversed; we will rely on the runtime execution of the specification to filter out invalid sequences of method calls. We will present a refinement of the flowgraph model of object lifetimes in Section 6.

\(^1\)that is, if we take the subgraph containing the vertices in \( M \) and every edge that connects two vertices in \( M \), we get a complete directed graph (with self-loops)
Figure 4.2: A “flow graph” for a stack component

By defining the graph, three coverage criteria immediately comes to mind: all nodes, all (feasible) edges, and all feasible paths. The all-nodes criterion means that every method is called at least once. All edges implies that for every two methods, if a call to one can follow a call to the other, then that sequence of calls must be executed. All paths means every object lifetime must be tested. While the all paths criterion is obviously strongest, it is also clear that it is infinitely large, and thus impossible to cover.

There are actually three test case generators that use the flowgraph strategy in the current version of Sulu. Two early Sulu automatic test case generators implemented the all-nodes (every method is called) criterion, and the all-edges (every pair of methods is called). A third, more general test case generator is implemented that can, given an integer $n$, generate test cases for all $n$-sequences of method calls, representing a walk through the flowgraph with $n$ interior nodes.

One key issue not addressed by the flowgraph model is input parameter selection. An all edges strategy tells us that \texttt{push} then \texttt{length} must be called, but what does one pass into
the call to push? The currently implemented generators use one of two strategies: use the default value; and let the programmer provide values.

Every Sulu object starts at some valid initial state. Because of this, it is possible to generate valid parameter values for methods by simply creating a variable that is of the same type as the parameter, and passing that. This might be enough for container objects, like stacks and lists, but it might not be for more complex objects whose method parameters have a direct impact on its behavior, like maps, and sorting machines.

Another simple strategy for parameter values is to let the programmer provide examples for every parameter type encountered. We use a limited version of this strategy for some of our experiments, although this does not preclude the implementation of more complex strategies for generating objects for parameter inputs. We can imagine taking, for example, parameter selection strategies proposed by Marinov et al. [18], and Cheon [27], and adapting these for use in Sulu.

### 4.3 Test Case Execution

Once test-cases are generated, there must be a way to execute these test cases and determine whether the component under test behaved correctly. Figure 4.3 is a screenshot of the GUI test runner tool implemented for Sulu. This test execution tool allows the human tester to select one or more test suites—components that realize the ComponentTester concept (a similar command-line tool is also available).

Given a test execution tool, the test engineer can then run the test suites, and the tool will place each test case into one of three categories: pass, fail, or invalid. Recall that a test case is essentially a sequence of method calls. During the execution of the test-cases, the interpreter executes preconditions and invariants before every method call. The test case execution tool considers as invalid a precondition or invariant check that fails from a test-
Figure 4.3: A GUI for running tests and reporting code coverage
case calling a method; that is, the test case did not exercise the component under test in a way that is allowed by its specification.

However, if it is not the test-case that triggers the precondition failure, but rather the software under test itself—that is, the software under test executes a method that causes a precondition failure—this indicates a failure of the software under test to abide by the contract of its underlying components. Test cases that expose this behavior are tagged as having failed. Similarly, all test cases that cause postcondition failures are marked as failed. Test cases that do not cause any assertion failures are marked as passed.

As test cases are executed, the Sulu interpreter also collects information about which parts of each software component is being executed. This execution profile of the software under test is one measure of the thoroughness of the test effort.

### 4.4 Evaluating the Test Effort

When executing unit tests, determining whether or not the test cases revealed buggy behavior is not enough. One must also ask: how thorough is one’s test suite? What is the level of its bug-revealing capabilities? That is, we need to also determine how effective the test suites are at finding bugs. The Sulu tools provide two of the most common measures of the strength of the test effort: code coverage and mutation analysis.

Code coverage is a white-box test adequacy criteria that takes advantage of the structure of the software under test. The simplest code coverage criteria is statement coverage, where a count is made of the number of statements of the component under test executed by the test suite. Aside from statement coverage, the Sulu code coverage profiler also collects decision coverage, and condition/decision coverage. Figure 4.3 shows how code coverage information can be displayed by a GUI tool.

We implement code coverage profiling by setting coverage flags on AST nodes. The Sulu
interpreter executes statements by traversing an abstract syntax tree that has special nodes that represent statements, conditionals, and boolean operators. Statement nodes can be flagged as executed once the interpreter encounters that node. Nodes that represent conditionals and boolean operations are given a score:

0 for unevaluated operations;

1 if the operation evaluated to either true only, or false only;

2 if the operation evaluated to true at one point and false at another point in time

We can then give a code coverage score by simply summing the scores attached to the AST nodes.

Sulu provides a second mechanism to assess the thoroughness of test suites: mutation analysis. Mutation analysis involves seeding bugs into otherwise acceptable code. Typically, this is done by transforming a software component via a certain mutation operator. A mutation operator produces a set of programs that is identical to the original aside from a single change. A mutation operator may change a plus to a minus, for example. If you had a realization that contained three additions in its methods, the mutant generator would generate three “mutant” realizations, one for each addition operation. Each mutated version would have its corresponding addition operation converted into a subtraction operation.

The Sulu mutation analysis tool runs each test suite against every mutant. If the test suite detected the bug (a test case failed), the mutant is said to be killed. If all test cases passed, the mutant is said to have survived. Every mutation operator may be capable of producing a large number of mutants, and thus a judicious selection of mutation operators must be used to make mutation analysis feasible. Offutt and his colleagues provide a set of five “sufficient” operators that have the property of corresponding to the result of a larger set of mutation operators [71]. Sulu currently provides close variants of the five operators in Offutt’s paper, plus one mutation operator used by Andrews and his colleagues [6].
As of this writing, a command-line tool is available to run mutation analysis on a realization. As with test case generators, Sulu provides a simple plug-in mechanism for adding more mutants, if the programmer so desires. In this case, a programmer who wishes to create a different mutation operator needs to extend `sulu.tools.mutator.Mutator`. This is an iterator-like abstract class that lets the mutation analysis tool iterate over every generated mutant.

By having both a framework where test case generators can be plugged into existing tools, and also an architecture where evaluation tools are pluggable into the system, we make it possible for Sulu to grow a suite of benchmark components, tests, automated test-generators, and evaluation criteria. This will let us make better evaluation and comparisons between two different test case generators, mutation operators, etc..

In the next chapter, we discuss the experimental setup of such an evaluation, comparing six different automatically generated test suites against evaluation criteria including the three code coverage criteria, and mutation analysis with six different mutation operators.
Chapter 5

Assessing Automated Test Generation

Once the software tools described in the previous chapter were constructed, we were now able to perform evaluations of automated unit test generation. Our strategy for evaluating test case generation involves these steps:

1. Implement a test generation algorithm
2. Select a set of reference software components
3. Generate test suites for every software unit
4. Execute test suites and gather code coverage information
5. Select (and implement, if necessary) a set of mutation operators
6. Run every relevant test suite against each mutant and gather mutation coverage

Before we describe in more detail the experimental setup used in evaluating the automated testing in Sulu, we discuss an early assessment by this author of a related automated testing mechanism, that of JMLUnit [29]. Sulu has adopted many of the ideas used by JMLUnit, notably the use of runtime checking of formal specifications as both a test oracle and a test filter.
5.1 Evaluating JMLUnit: An Early Experience

Cheon and Leavens [29] have proposed a way to semi-automatically generate JUnit-style test cases for Java classes that have JML specifications, and have written a tool for this purpose. The JML-JUnit tool is included with the open-source distribution of JML (http://jmlspecs.org).

The JMLUnit tool takes advantage of the JML assertion checker that automatically generate run-time checks for code with JML specifications. As such, it provides an attractive choice for Java developers who write formal specifications. Yet little was known about the effectiveness of testing using JMLUnit.

We presented an assessment of the effectiveness of JMLUnit testing and the lessons learned from this experience [82]. The experimental assessment indicated that JMLUnit, while providing useful insights into the use of formal specifications in automated testing, was not a strong technique for detecting implementation bugs.

Adapting the experiments that Edwards [39] and Mungara [70] have done for other testing strategies, we followed the following steps for our unit testing experiment:

1. choose classes to test;
2. generate the mutants—- buggy versions of correctly implemented classes;
3. generate JML-JUnit tests for each of the classes;
4. run the JML-JUnit tests on each of the mutants, and count the number of mutants for which the tests failed.

We used a modified version of the mutation testing tool Jester [69] to generate mutants for the classes we tested. In an earlier experiment, Mungara [70] selected a representative set of classes from java.util, chosen to cover the spectrum of number of methods, average method size, and significant lines of code. From Mungara’s collection, we selected
<table>
<thead>
<tr>
<th>Class</th>
<th>JML-JUnit</th>
<th>Random</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>% found</td>
<td># tests</td>
</tr>
<tr>
<td>BitSet</td>
<td>11.5</td>
<td>715</td>
</tr>
<tr>
<td>Hashtable</td>
<td>10.0</td>
<td>224</td>
</tr>
<tr>
<td>LinkedList</td>
<td>8.1</td>
<td>203</td>
</tr>
<tr>
<td>Observable</td>
<td>0.0</td>
<td>16</td>
</tr>
<tr>
<td>TreeSet</td>
<td>15.3</td>
<td>71</td>
</tr>
<tr>
<td>Vector</td>
<td>19.7</td>
<td>678</td>
</tr>
</tbody>
</table>

Table 5.1: Mutant detection JML-JUnit vs randomly executing two methods in sequence as reported in [70]

six java.util classes for which JML specifications were available: BitSet, Hashtable, LinkedList, Observable, TreeSet, and Vector.

The mutation operators we selected were:

- Change numerical constants: change the digit 0 to 1, 5 to 6, 9 to 0
- Flip boolean variables: changes true to false and vice versa
- Changes conditions of if statements to always evaluate to true (or false)
- Mutate ++ to -- and vice versa
- Mutate != to == and vice versa

After generating and compiling the mutants for each of these classes, we then used JML-JUnit tools on the correctly implemented components to generate the JUnit test classes we needed to run. Figure 5.1 shows the results of the experiment.

We can compare this result with those reported by Mungara. One of the methods Mungara used to generate test cases is to randomly generate a pair of methods to be called in sequence; parameters and assertions are then manually added. Figure 5.1 compares the results for the two test-generation strategies.
We should note that while our procedure for running the experiments were the same, Mungara used additional mutants not found in this experiment. Nevertheless, the numbers suggest that compared to the strategy of randomly calling two methods in sequence, the JML-JUnit testing strategy is not very effective.

The experiment suggests that the effectiveness of JMLUnit with regards to detecting bugs is somewhat disappointing—there is no test suite for which more than 20% of the generated mutants were killed. It is worth noting, however, that running JMLUnit on the correctly implemented component has revealed several bugs in the specifications of almost all of the classes tested. These specification bugs can be separated into three general categories: (a) incorrect specification when dealing with null values, (b) not specifying behavior for throwing exceptions, and (c) not differentiating between object identity (==) and the equals method. Since this work on assessment of JMLUnit, researchers have done more work [27] that alleviates some of the problems reported here, which may increase test effectiveness.

Experience in carrying out this experiment has highlighted specific issues. One thing we noted with the JMLUnit test cases was that while the cross-product of possible method parameters were provided, every test case had only one method call. We speculated that a sequence of method calls would expose more of the object’s state, and thus catch more bugs.

We also became convinced that code coverage information is also important to collect aside from mutants killed. Code coverage data can complement counting the number of mutants killed, by providing additional insight into how “hard” a test suite exercises a given piece of code. That is, a test suite that covers 10% of the code and reveals 10% of the bugs is different from a test suite that reveals the same amount of bugs, but covers 100% of the code.

Similarly, other summary statistics on the code being tested may also be helpful in interpreting results. Data such as the number of methods, number of statements, and so on can provide a useful perspective when looking at differences in the performance data. Such data
provides a richer context for discussing potential differences in performance and effectiveness of the various subject software components, and testing strategies.

5.2 Assessing Exhaustive Enumeration of Method Sequences: Experimental Setup

The experiments performed for assessing the testing strategy used in Sulu broadly follows the same methodology in the previous section, but also applied the lessons learned from the earlier experiment. That is, we select a set of software components, we generate a set of tests for each component, and finally gather coverage information. In this case we gather both code coverage, as well as mutation coverage (the number of mutants killed versus the total number of mutants generated). Additionally, for mutation analysis, we selected a relevant set of mutation operators.

5.2.1 Component Selection and Implementation

For the purposes of this research, a set of ten software components were selected for implementation. These will serve as the reference components against which the automated test case generation strategies are evaluated against; Table 5.2 lists these components.

These software components are collection abstract data types, meant to represent some of the most common data structures and algorithms used by programmers. These components can broadly be separated into two sets, components derived from the Resolve family of components, and software units derived from java.util classes. The binary tree, list, stack and sorting machines come from the Resolve heritage.

The Sorter components recasts the sorting algorithm into a sorting machine that makes it more component-based, following loosely the work by Weide et al. [87]. Typical usage of these components would be calling the insert method for every item in the list to be sorted,
then calling `sort`, then finally calling `remove` repeatedly, where `remove` removes the smallest element in the list.

There are four different sorting algorithms implemented, representing different strategies of where the work of the sorting is distributed. **BubbleSort** keeps the objects in arbitrary order when inserted, and does all the work on the call to `sort`. **ListBased** keeps the list in sorted order on insert (essentially an insertion sort). The **MinFinding** realization keeps the list in arbitrary order, and locates the smallest item on every call to `remove`. And finally **HeapBased** keeps the objects in a heap structure, thus the work is distributed between the `insert` and the `remove` methods.

The remaining three components, the Hashtable based map and the two Vector components are based on the `java.util` library. The `Vector` components implement most of the Java `List` interface, while the Hashtable implements most of the Java `Map` interface. The `ArrayBased` implementation of Vector is based closely on the 1.3 version of the GNU Classpath library implementation, while the other two are based on the Java source code for JDK 1.5.

While the author attempted to create a very close implementation, these are not exactly the same, due to certain features that are lacking in Sulu. Namely: the lack of iterator

<table>
<thead>
<tr>
<th>Concept</th>
<th>Realization</th>
</tr>
</thead>
<tbody>
<tr>
<td>BinaryTree</td>
<td>Standard</td>
</tr>
<tr>
<td>List</td>
<td>TwoStacks</td>
</tr>
<tr>
<td>Map</td>
<td>Hashtable</td>
</tr>
<tr>
<td>Stack</td>
<td>LinkedList</td>
</tr>
<tr>
<td>Sorter</td>
<td>BubbleSort</td>
</tr>
<tr>
<td>Sorter</td>
<td>HeapBased</td>
</tr>
<tr>
<td>Sorter</td>
<td>ListBased</td>
</tr>
<tr>
<td>Sorter</td>
<td>MinFinding</td>
</tr>
<tr>
<td>Vector</td>
<td>ArrayBased</td>
</tr>
<tr>
<td>Vector</td>
<td>LinkedList</td>
</tr>
</tbody>
</table>

Table 5.2: Sulu reference components used for evaluation
components, the lack of exceptions, and the lack of selection and looping structures other than `while` and `if` statements. Aliasing pointers are a core part of Java, and implementing these Java-like components required the creation of a `Pointer` concept that allows aliasing. Using this aliasing pointer component should not be taken as the norm for Sulu components.

### 5.2.2 Test Case Generation

The Sulu interpreter allows for different implementations of test case generators. Our implementation uses the method described in Section 4.2 for automatically generating test cases, which is essentially generate a sequence of method calls. Our adequacy criteria corresponds to all nodes (call every method), all pairs of methods, and all triples of methods. All pairs of methods is essentially equivalent to all edges, although there is no explicit test for creating an object and not calling a method on it. Since there are no explicit finalizers for Sulu components, the sequence `initialize→finalize` is implicitly tested by the other test methods.

Our test case generation tool can generate test cases exhaustively for any sequence of \( n \) method calls. However, because of exponential explosion, the effectiveness of all quadruple method calls and beyond were not considered in this research.

In addition to the sequence of method calls, a test case needs to provide inputs to methods that have parameters. Since every Sulu variable has a default value, we can automatically provide one value for every parameter of most Sulu method calls. It is often the case, however, that different parameter values may result in different behavior, and thus uncover more bugs. Our test case generator thus can generate test cases that enumerate all sequences of \( n \) method calls for \( p \) possible different parameter values for every formal parameter in a method call. That is, if there are \( k \) formal parameters in a sequence of method calls, and we want to test \( p \) different parameter values for each formal parameter, we generate \( p^k \) test cases for that sequence. Because of this combinatorial explosion problem, we can only generate test suites that are of practical use when \( n \) and \( p \) are low.
We also encounter the problem of generating parameter values other than the default value for every parameter. Currently, our test case generator generates stubs where the testing professional can fill in additional parameter values for every parameter type encountered in the software component.

We generated six different test suites from our test case generator. We generated test suites corresponding to each test case being a single method call, a pair of method calls, and three method calls; and for each of those, a test suite for one parameter value, and a test suite for two parameter values for each formal parameter. Table 5.3 shows the number of test cases generated for each test suite. Because we are generating test suites from method signatures, test suites generated for different realizations of the same concept are essentially identical.

When generating test suites where two possible parameter values are entered for every parameter, the programmer must fill in two example values per parameter type. For arbitrary objects, we use integers as the actual type, and the set of parameter values are either 0 or 1. Some methods of the components we tested also took \textit{selftype} parameters \cite{19, 21}; that is, parameter values that have the same type as the object from which the method is called.

Since these are all collection objects, for the selftype values, we use one of either the default value (an empty collection), or the default value with 0 inserted.

Figure 5.1 shows the relationships between the generated test cases. A test suite generates with every sequences of \( n \) method calls and \( p \) parameter values is a superset of every test
suite with fewer method calls or fewer parameter values. Thus our most comprehensive test suite, all triples with 3 parameter values is a superset of all the other generated test suites.

Note the exponential nature of the strategy. A sequence of three methods is at the edge of what is practical for our test generation mechanism. However, we should emphasize that Sulu supports the construction of more sophisticated test case generators.

Once test cases were generated, each test suite was executed, code coverage information

<table>
<thead>
<tr>
<th>Component</th>
<th>One Parameter</th>
<th>Two Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>singles</td>
<td>pairs</td>
</tr>
<tr>
<td>BinaryTree</td>
<td>Standard</td>
<td>33.3</td>
</tr>
<tr>
<td>List</td>
<td>TwoStacks</td>
<td>20.0</td>
</tr>
<tr>
<td>Sorter</td>
<td>BubbleSort</td>
<td>20.0</td>
</tr>
<tr>
<td>Sorter</td>
<td>HeapBased</td>
<td>20.0</td>
</tr>
<tr>
<td>Sorter</td>
<td>ListBased</td>
<td>20.0</td>
</tr>
<tr>
<td>Sorter</td>
<td>MinFinding</td>
<td>20.0</td>
</tr>
<tr>
<td>Stack</td>
<td>LinkedList</td>
<td>33.3</td>
</tr>
</tbody>
</table>

Table 5.4: Percent of invalid test cases in each test suite
was gathered (Table 5.5), as well as the number of invalid test cases. Table 5.4 shows the percentage of test cases that are invalid for each component. The two Vector components, and the one Map component are not included in Table 5.4, since none of their methods have preconditions, and so all the test cases are considered valid. The slight discrepancies in the different Sorter components come from whether or not \texttt{remove} can be called immediately after \texttt{insert}.

5.2.3 Evaluation Criteria

Sulu provides tools to evaluate the effectiveness of the automated testing tools using both code coverage and mutation analysis. There are three different code coverage criteria that are measured: statement coverage, decision coverage, and condition/decision coverage [54]. Statement coverage is simply a count of the number of statements executed versus the total number of statements in the component under test. Decision coverage means that it counts whether for every boolean statement in an \texttt{if} statement and \texttt{while} loop, it evaluates to \texttt{true} at least once, and \texttt{false} at least once. That is, every \texttt{while} loop or \texttt{if} statement can be given a maximum of 2 points:

- 0 for unevaluated conditionals;
- 1 if evaluated to either true only, or false only;
- 2 if evaluated to true at one point and false at another point in time

Condition/decision extends the idea of condition coverage by considering complex boolean expressions that are composed of several boolean variable inputs. The condition/decision coverage criteria states that not only does the decision have to be \texttt{true} at some point and \texttt{false} at another, every input to the decision should also be \texttt{true} at least once and \texttt{false} at least once. Thus, for example if the conditional is \texttt{A \&\& B || C}, our condition/decision coverage counts 8 possible points, a maximum of two points for every input variable, and another
### Table 5.5: Code Coverage information for Sulu components

<table>
<thead>
<tr>
<th>Concept</th>
<th>Realization</th>
<th>methods</th>
<th>statements</th>
<th>decision count</th>
<th>c/d count</th>
</tr>
</thead>
<tbody>
<tr>
<td>BinaryTree</td>
<td>Standard</td>
<td>3</td>
<td>19</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>List</td>
<td>TwoStacks</td>
<td>5</td>
<td>15</td>
<td>6</td>
<td>12</td>
</tr>
<tr>
<td>Map</td>
<td>Hashtable</td>
<td>9</td>
<td>133</td>
<td>44</td>
<td>116</td>
</tr>
<tr>
<td>Sorter</td>
<td>BubbleSort</td>
<td>3</td>
<td>29</td>
<td>10</td>
<td>20</td>
</tr>
<tr>
<td>Sorter</td>
<td>HeapBased</td>
<td>5</td>
<td>63</td>
<td>22</td>
<td>46</td>
</tr>
<tr>
<td>Sorter</td>
<td>ListBased</td>
<td>5</td>
<td>18</td>
<td>6</td>
<td>14</td>
</tr>
<tr>
<td>Sorter</td>
<td>MinFinding</td>
<td>5</td>
<td>13</td>
<td>4</td>
<td>8</td>
</tr>
<tr>
<td>Stack</td>
<td>LinkedList</td>
<td>5</td>
<td>14</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Vector</td>
<td>ArrayBased</td>
<td>18</td>
<td>139</td>
<td>60</td>
<td>138</td>
</tr>
<tr>
<td>Vector</td>
<td>LinkedList</td>
<td>18</td>
<td>157</td>
<td>50</td>
<td>120</td>
</tr>
<tr>
<td>TOTAL</td>
<td></td>
<td>76</td>
<td>600</td>
<td>204</td>
<td>478</td>
</tr>
</tbody>
</table>

two points for the decision itself. Thus, condition/decisision subsumes decision coverage in that 100% condition/decision coverage implies 100% decision coverage. Complete decision coverage also implies complete statement coverage. Table 5.5 shows the method, statement, decision, and condition/decision count for each of the components.

While method coverage can also be measured, we did not do so in our experiments because all the test case generators have complete method coverage; and we expect that any reasonable test case generator should also do so as well.

In addition to code coverage, Sulu also provides a mutation analysis tool. The tool allows programmers to add mutation operators at will, but we implemented the following:

1. Change an integer constant to one of 0, 1, −1, and its negation
2. Change an arithmetic operator to another arithmetic operator
3. Change a comparison operator to another comparison operator
4. Change a boolean operator to another boolean operator
5. Force an if statement or while loop to evaluate to either true or false
<table>
<thead>
<tr>
<th>Component</th>
<th>Realization</th>
<th>del stmt</th>
<th>chg arith</th>
<th>chg comp</th>
<th>chg bool</th>
<th>chg const</th>
<th>force t/f</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sorter</td>
<td>MinFinding</td>
<td>13</td>
<td>4</td>
<td>10</td>
<td>0</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>Stack</td>
<td>LinkedList</td>
<td>13</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>6</td>
<td>0</td>
</tr>
<tr>
<td>List</td>
<td>TwoStacks</td>
<td>15</td>
<td>4</td>
<td>15</td>
<td>0</td>
<td>0</td>
<td>6</td>
</tr>
<tr>
<td>Sorter</td>
<td>ListBased</td>
<td>18</td>
<td>4</td>
<td>10</td>
<td>3</td>
<td>2</td>
<td>6</td>
</tr>
<tr>
<td>BinaryTree</td>
<td>Standard</td>
<td>19</td>
<td>8</td>
<td>5</td>
<td>0</td>
<td>8</td>
<td>2</td>
</tr>
<tr>
<td>Sorter</td>
<td>BubbleSort</td>
<td>28</td>
<td>8</td>
<td>20</td>
<td>1</td>
<td>13</td>
<td>10</td>
</tr>
<tr>
<td>Sorter</td>
<td>HeapBased</td>
<td>63</td>
<td>8</td>
<td>60</td>
<td>1</td>
<td>17</td>
<td>22</td>
</tr>
<tr>
<td>Map</td>
<td>Hashtable</td>
<td>133</td>
<td>60</td>
<td>85</td>
<td>23</td>
<td>103</td>
<td>44</td>
</tr>
<tr>
<td>Vector</td>
<td>ArrayBased</td>
<td>139</td>
<td>88</td>
<td>140</td>
<td>13</td>
<td>100</td>
<td>60</td>
</tr>
<tr>
<td>Vector</td>
<td>LinkedList</td>
<td>156</td>
<td>52</td>
<td>115</td>
<td>15</td>
<td>57</td>
<td>50</td>
</tr>
<tr>
<td>TOTAL</td>
<td></td>
<td>597</td>
<td>236</td>
<td>460</td>
<td>56</td>
<td>308</td>
<td>204</td>
</tr>
</tbody>
</table>

Table 5.6: Number of mutants generated for each component and mutation operator

6. Delete a statement

The first five correspond to Offutt’s set of sufficient mutation operators [71], the last one is taken from Andrews [6], where the “delete statement” operator is added to the set. Table 5.6 show the number of mutants generated per component for every mutation operator.

Every generated test suite is run against every mutant. When the tests all pass the mutant is said to have survived, meaning the test suite is not strong enough to catch the mutant. If one of the tests fail, the testing is terminated, and the mutant is deemed killed. We then gather the number of mutants killed versus the total number of mutants generated.

5.2.4 Issues With Mutation Analysis

Time Constraints

Any researcher running mutation analysis comes up with several common problems. The first is the length of time needed to run it. Because mutation testing can generate a large number of mutants, running the tests can take a long time. Indeed, mutation testing for one
of the components (the linked-list based Vector) using the most comprehensive test suite took about twelve hours. Running mutation testing for that amount of time is still feasible to do for the purposes of this research, but may be prohibitive for practical purposes. Additional mutants or longer running test suites can quickly turn this kind of comprehensive mutation testing infeasible.

A number of things can help lower the running time, aside from choosing a small set of mutation operators. One is batch-processing the tests on a server farm, as part of e.g., a weekly build-and-test procedure. Mutation testing might still be used in the normal test-and-code test driven development cycle by taking a random sample of the mutants generated, instead of running everything at once. Also, while beyond the scope of this work, we note that our unit tests are inherently massively parallelizable. Running one test suite against a mutant is independent of running the same test suite against a different mutant. Indeed, even individual test cases can be independently executed. We discuss some more ways to manage long-running tests in Section 5.4.

Equivalent Mutants

A second problem are behaviorally equivalent mutants. A mutation operator may change the program to something that is behaviorally equivalent to the original. An example of this is the ArrayBased realization for the Vector component. Figure 5.2 shows the init method of the ArrayBased realization of Vector; init is the Sulu equivalent of a constructor.

The ArrayBased realization is initially set up to have an array with a capacity of ten, and a size of zero. When applying the “change an integer value” mutator to the component, we would at one point change the count of the items in the Vector to one (line 5) and at another point change the initial capacity of the array to one (line 6).

Changing the initial count would obviously create a buggy program, calling size on an empty vector would return one. However, changing the initial capacity would not cause it
```swift
var elementData: ItemArray;
var elementCount: Int;

method init() {
    elementCount := 0;
    elementData.setSize(10);
}
```

Figure 5.2: Initializing the ArrayBased realization of Vector

to be buggy. That is, the program will behave correctly, because add will simply adjust the size of the array if the array is filled (e.g., on the second add call).

Thus, even if changing the initial size of the array to one creates a different program, it conforms to the specification, and behaviorally equivalent to the original. Therefore, this is not a buggy component, and no correct test suite will detect it. However, determining whether two programs are equivalent is generally undecidable, and the more specific case of showing a mutant to be equivalent to its original has also been shown to be undecidable [22]. Some mechanism for allowing the developer to annotate programs to prevent certain mutants from being generated will be useful, though not currently implemented here. This means that some mutants generated (including the one changing the initial capacity of the array) in this research are equivalent to the original and a 100% kill ratio will not be possible.

We manually examined the mutants generated by the “change comparison” mutation operator, and determined that at least 36 of the 460 mutants generated were behaviorally equivalent to the original program. The kill ratio for detectable mutants for the most comprehensive test suite of all triples with two parameter values is at 71.9%. However, we note that 36 of 460 mutants is less than 10% of all generated mutants for the “change comparison” operator.

From a practical standpoint, however, it may not be prudent to ask the tester to inspect every mutant for equivalence to the original software component, since there may be a very
large number of mutants. Although generally undecidable, Offutt and Pan \cite{72} recognize that never reaching 100% coverage may mean the loss of confidence of the programmer in the metric; they argue that some static analysis techniques may be able to detect a good number of equivalent mutants.

**Infinite Loops**

A final problem is infinite loops. It is quite common for a mutation operator to create mutants that go in an infinite loop—for example, by deleting the statement that increments the loop counter, or making the loop counter decrement instead of increment. However, again, the problem of whether a program will terminate or not is generally undecidable. The mutation analysis tool manages this problem by setting a timeout value. When the experiment was run, the time out was configured to be ten seconds. The timeout value is configured in such that any test that runs over the time limit is stopped, and test suites that time out are reported as killed.

### 5.3 Results of the Experiment

Recapping our experimental setup, we have six different test suites that were generated automatically: using all-singles, all-pairs, and all-triples sequences of methods. For each of those, either one value is evaluated per parameter, or two.

We also have ten reference software components to run each of the test cases against, representing common data structures and algorithms. Finally, we have nine criteria for evaluating the thoroughness of the test suites: three code coverage criteria (statement, decision, and condition/decision); and six mutation coverage criteria (delete statement, change arithmetic operators, change comparison operators, change integer constant, force true/false, and change boolean operators).
We ran every test suite against every component, and we gathered information about the code coverage. We also ran every test suite against every mutant generated for each relevant software unit, and counted the number of mutants killed. Table 5.7 gives a flavor of the detail of information gathered from the experiments; similar information is available for code coverage. For brevity we will only report aggregate data in the rest of this section, although the complete set of data is shown in Appendix A of this document.

Table 5.8 shows aggregate code coverage information for each test suite. The results show what we expected: All pairs of method calls is superior to all single method calls, and all triples cover more than all pairs. Also, a test suite that takes in two parameters does better than the corresponding test suite that only takes one parameter value. Also notable is the fact that all triples with two parameters have 90% statement coverage, and close to 85% decision and condition/decision coverage.

Aggregate data for mutation kill ratio is shown in Table 5.9. The data also confirms that all-triples kill more mutants than all-pairs, and all-pairs in turn kill more mutants than all-singles. Furthermore, two parameters values have better kill ratios than one. The all-triples with two parameter values test suite achieved 75% or nearly 75% kill ratios on four of the

<table>
<thead>
<tr>
<th>Concept</th>
<th>Realization</th>
<th>del stmt</th>
<th>chg arith</th>
<th>chg comp</th>
<th>chg const</th>
<th>force t/f</th>
</tr>
</thead>
<tbody>
<tr>
<td>BinaryTree</td>
<td>Standard</td>
<td>2 (10.5)</td>
<td>0 (0.0)</td>
<td>0 (0.0)</td>
<td>0 (0.0)</td>
<td>0 (0.0)</td>
</tr>
<tr>
<td>List</td>
<td>TwoStacks</td>
<td>2 (13.3)</td>
<td>2 (50.0)</td>
<td>3 (20.0)</td>
<td>N/A</td>
<td>1 (16.7)</td>
</tr>
<tr>
<td>Map</td>
<td>Hashtable</td>
<td>35 (26.3)</td>
<td>36 (60.0)</td>
<td>12 (14.1)</td>
<td>26 (25.2)</td>
<td>12 (27.3)</td>
</tr>
<tr>
<td>Sorter</td>
<td>BubbleSort</td>
<td>7 (25.0)</td>
<td>2 (25.0)</td>
<td>6 (30.0)</td>
<td>5 (38.5)</td>
<td>3 (30.0)</td>
</tr>
<tr>
<td>Sorter</td>
<td>HeapBased</td>
<td>6 (9.5)</td>
<td>0 (0.0)</td>
<td>3 (5.0)</td>
<td>2 (11.8)</td>
<td>1 (4.5)</td>
</tr>
<tr>
<td>Sorter</td>
<td>ListBased</td>
<td>4 (22.2)</td>
<td>2 (50.0)</td>
<td>3 (30.0)</td>
<td>2 (100.0)</td>
<td>2 (33.3)</td>
</tr>
<tr>
<td>Sorter</td>
<td>MinFinding</td>
<td>2 (15.4)</td>
<td>2 (50.0)</td>
<td>0 (0.0)</td>
<td>0 (0.0)</td>
<td>0 (0.0)</td>
</tr>
<tr>
<td>Stack</td>
<td>LinkedList</td>
<td>5 (38.5)</td>
<td>N/A</td>
<td>N/A</td>
<td>3 (50.0)</td>
<td>N/A</td>
</tr>
<tr>
<td>Vector</td>
<td>ArrayBased</td>
<td>29 (20.9)</td>
<td>8 (9.1)</td>
<td>24 (17.1)</td>
<td>14 (14.0)</td>
<td>9 (15.0)</td>
</tr>
<tr>
<td>Vector</td>
<td>LinkedList</td>
<td>53 (34.0)</td>
<td>2 (3.8)</td>
<td>16 (13.9)</td>
<td>7 (12.3)</td>
<td>11 (22.0)</td>
</tr>
<tr>
<td>Total</td>
<td></td>
<td>145 (24.3)</td>
<td>54 (22.9)</td>
<td>67 (14.6)</td>
<td>59 (19.2)</td>
<td>39 (19.1)</td>
</tr>
</tbody>
</table>

Table 5.7: Mutation coverage information for all singles with one parameter
Table 5.8: Percent of code covered for each test suite

<table>
<thead>
<tr>
<th>Test Suite</th>
<th>% Code Coverage</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>method calls</td>
</tr>
<tr>
<td>Singles</td>
<td>1</td>
</tr>
<tr>
<td>Pairs</td>
<td>1</td>
</tr>
<tr>
<td>Triples</td>
<td>1</td>
</tr>
<tr>
<td>Singles</td>
<td>2</td>
</tr>
<tr>
<td>Pairs</td>
<td>2</td>
</tr>
<tr>
<td>Triples</td>
<td>2</td>
</tr>
</tbody>
</table>

Table 5.9: Aggregate mutation coverage information for each test suite

<table>
<thead>
<tr>
<th>Test Suite</th>
<th>% Mutation Coverage (Kill Ratio)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>method calls</td>
</tr>
<tr>
<td>Singles</td>
<td>1</td>
</tr>
<tr>
<td>Pairs</td>
<td>1</td>
</tr>
<tr>
<td>Triples</td>
<td>1</td>
</tr>
<tr>
<td>Singles</td>
<td>2</td>
</tr>
<tr>
<td>Pairs</td>
<td>2</td>
</tr>
<tr>
<td>Triples</td>
<td>2</td>
</tr>
</tbody>
</table>

six mutation operators, with lower numbers for the “change comparison operators” and the “change integer constant” mutators.

Of particular interest is statement coverage versus the kill ratio of the “delete statement” mutation operator. Recall that a statement is considered covered if that statement was executed at least once. The delete statement mutator creates a mutant by deleting one from the component under test. Delete statement is thus a stronger measure of statement coverage in that it tells us whether deleting that statement will cause our test cases to fail. A similar comparison can be made between decision coverage and the “force conditionals to true or false” mutant kill ratio.

Figure 5.3 shows the statement coverage versus delete statement kill ratio. The kill ratios for the delete statement mutation operator is much lower than the statement coverage. It
5.3.1 The Effectiveness of Test Suites

The experimental results show that for the most comprehensive automatically generated test suite (all triples with two parameters), we achieve 90% statement coverage and nearly 80% decision and condition/decision coverage with the reference components. Mutation analysis also shows fairly high coverage, with kill ratios of above or nearly 75% for four of the six mutation operators.

In contrast, a recent conversation with a Microsoft employee revealed to the author that Microsoft’s code coverage bar for production-level software is 75% coverage (it is unclear whether this was statement coverage or decision coverage). Of course, our simple components are much more easily tested than, for example, an operating system which may need special hardware to test certain sections of the code. However, the high coverage rate for these
components means that the programmer is given a baseline test that frees him to attack the parts of the code that is not covered by the automatically generated tests.

Figure 5.4 shows a radar graph of the statement coverage of the all-triples with two parameter values test suite. The axes represent a size metric—the number of statements of each component. The dark gray area shows the statements covered by the test suite; the light area shows the total statements for each component, and the dashed line shows the 75% code coverage bar. The graph shows one weakness of this study: the number of statements is heavily weighted by the three largest components. Figure 5.5 is a normalized radar graph, where the radial axis is scaled to the percent of statements covered per software component. The light gray area represents the all-triples with two parameters test suite, the darker area shows all-pairs with two parameters, which is the test suite with the next best coverage. The dotted line is the 75% code coverage level. Figure 5.4 and Figure 5.5 show that our most

Figure 5.4: Statement coverage for all-triples with two parameter values
effective test suite achieved the 75% code coverage bar for each one of the components in our reference set. The relatively low (but still over 75%) statement coverage on the Hashtable component comes from the fairly complex rehashing code not covered by the short length of the method sequences in our test suite.

The all-pairs test suite with two parameters, while having less coverage than the all-triples method still has 85% statement coverage and close to 70% decision and condition/decision coverage. All-pairs with two parameter values might still be useful as a baseline set of test cases, especially if time is a strong constraint.

Figure 5.6 is a parallel coordinate graph of the aggregate coverage information among all nine measures. Each vertical axis represents the coverage over their respective size measures. That is, the percent of statements, decisions, condition/decision cases for code coverage, and percent of mutants killed for each mutation operator. Every line in the graph associates the coverage achieved by a test suite across each of the measures. The dashed grey line over the
Figure 5.6: Aggregate coverage graph of experimental results

75% gridline is the coverage bar where our most comprehensive test suite is at or nearly at
for seven out of the nine measures of the test effort. From this graph in Figure 5.6, we can
clearly see the strengths of the different suites; and clearly the Triples2 suite is best, while
Singles1 is worst.

That more method calls have better adequacy coverage than fewer method calls, and that
two parameter values are better than one is evident from the graph in Figure 5.6 and the
subset relationships between the test suites. Figure 5.7 shows again the subset lattice of
the test suites; for any path in the lattice, every node has a greater bug-detecting capability
than every preceding node in the path.

We performed statistical analyses to answer the question: are there differences in code cov-
erage among test suites? To determine at a 95% confidence level that the differences in
coverage is not from random chance, we performed a one-way analysis of variance. At
$\alpha = 0.05$, the analysis of variance found significant differences exist between the test suites
on all our coverage measures. We also performed a Tukey’s test as a post-hoc analysis on
Figure 5.7: Each succeeding node in every path has stronger bug-detecting capabilities
every pair of test suites, for every coverage measure; this allows us to determine which specific
pairs of test suites are significantly different.

Table 5.10 shows the results to the Tukey test. Each sub-table represents Tukey’s test
performed on all pairs of test suites on a certain measure of thoroughness. The numbers to
the right of each table is the cumulative percent of coverage. Each column of letters represent
cluster of means where we cannot reject the null hypothesis. That is, for each column, the
test suites that are associated with a letter cannot be said to be statistically significant to
each other. Pairs that do not share membership on any cluster are significant at $\alpha = 0.05$.

Although using Tukey’s test, *Triples2* is not significantly better than *Pairs2* in any of the
measures, recall from Figure 5.7 that *Triples2* contains all of the test cases of *Pairs2*, and
thus any additional coverage is known to be beyond what can be covered *Pairs*. We are
therefore particularly interested in the specific pairs of test suites where there is no subsump-
tion relationship, and thus where their relative strengths are unknown. By examining the
lattice structure shown in Figure 5.7 we determine that these three pairs do not have this
subsumption relationship: (*Pairs1, Singles2*), (*Singles2, Triples1*) and (*Pairs2, Triples1*).
| statement | | | decision | | | condition/decision | | |
|---|---|---|---|---|---|---|---|
| Triples2 | A | 90.3 | Triples2 | A | 85.8 | Triples2 | A | 85.1 |
| Pairs2 | A | 85.0 | Pairs2 | A B | 77.5 | Pairs2 | A B | 78.0 |
| Triples1 | A | 82.7 | Triples1 | A B | 69.6 | Triples1 | A B | 68.8 |
| Pairs1 | B | 77.8 | Pairs1 | B C | 62.7 | Pairs1 | B C | 62.9 |
| Singles2 | B | 59.7 | Singles2 | C D | 42.2 | Singles2 | C D | 43.5 |
| Singles1 | B | 54.7 | Singles1 | D | 32.5 | Singles1 | D | 33.7 |

| delete statement | | chg. arithmetic op | | | | force t/f | | chg. boolean op | |
|---|---|---|---|---|---|---|---|---|
| Triples2 | A | 78.6 | Triples2 | A | 74.6 | Triples2 | A | 75.5 |
| Pairs2 | A | 69.2 | Pairs2 | A B | 66.9 | Pairs2 | A B | 65.7 |
| Triples1 | B | 46.6 | Triples1 | B C | 48.7 | Triples1 | B | 43.1 |
| Pairs1 | B | 45.6 | Pairs1 | C D | 42.4 | Pairs1 | B | 39.7 |
| Singles2 | B C | 31.2 | Singles2 | D E | 28.4 | Singles2 | B C | 25.5 |
| Singles1 | B C | 24.3 | Singles1 | E | 15.7 | Singles1 | C | 19.1 |

<table>
<thead>
<tr>
<th>chg. comparison op</th>
<th></th>
<th>chg. integer constant</th>
<th></th>
<th></th>
<th></th>
<th>chg. boolean op</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Triples2</td>
<td>A</td>
<td>66.3</td>
<td>Triples2</td>
<td>A B</td>
<td>65.3</td>
<td>Triples2</td>
<td>A</td>
</tr>
<tr>
<td>Pairs2</td>
<td>A</td>
<td>55.4</td>
<td>Pairs2</td>
<td>A B</td>
<td>53.6</td>
<td>Pairs2</td>
<td>A B</td>
</tr>
<tr>
<td>Triples1</td>
<td>B</td>
<td>35.2</td>
<td>Triples1</td>
<td>A B</td>
<td>45.8</td>
<td>Triples1</td>
<td>A B C</td>
</tr>
<tr>
<td>Pairs1</td>
<td>B C</td>
<td>31.7</td>
<td>Pairs1</td>
<td>A B C</td>
<td>39.0</td>
<td>Pairs1</td>
<td>A B C</td>
</tr>
<tr>
<td>Singles2</td>
<td>C D</td>
<td>18.3</td>
<td>Singles2</td>
<td>B C</td>
<td>25.6</td>
<td>Singles2</td>
<td>B C</td>
</tr>
<tr>
<td>Singles1</td>
<td>D</td>
<td>14.6</td>
<td>Singles1</td>
<td>C</td>
<td>19.2</td>
<td>Singles1</td>
<td>C</td>
</tr>
</tbody>
</table>

Table 5.10: Tukey’s test results for each comparison metric; numbers on right of tables are percent covered; test suites not connected by the same letter are significantly different at $\alpha = 0.05$. 
Table 5.11: Percent covered: Singles2 versus Pairs1 and Triples1; starred values indicate significant difference against Singles2 at $\alpha = 0.05$

At $\alpha = 0.05$, Tukey’s test shows that Triples1 is significantly better than Singles2 in all code coverage measures, and two mutation coverage measures: “change arithmetic operators”, and “change comparison operators”. Pairs1 is significantly better than Singles2 with the statement coverage measure. Table 5.11 summarizes the result of comparing Singles2 against Triples1, and Singles2 against Pairs1.

This lends some additional evidence to our speculation in Section 5.1 that not changing the state of the component under test is an important factor in the poor performance of JMLUnit in our earlier experiment.

For Triples1 and Pairs2, there is no significant difference between these test suites with the code coverage metrics; but Pairs2 is significantly better than Triples1 in three of the mutation metrics: delete statement, change comparison operators, and force conditionals to true or false. This is indicative of the balance that must be struck between the length of the method call sequences and the number of parameter values that is passed into the methods. We summarize the comparison of Pairs2 against Triples1 in Table 5.12.
Table 5.12: Percent covered: Triples1 versus Pairs2; starred values indicate measures where Pairs2 is significantly better at $\alpha = 0.05$

5.3.2 The Size Factor

In the above analysis, we assume that test suite is the only factor that affects the coverage results. We can ask additionally: does size matter? Recall that among our reference components, seven are derived from the Resolve family of components, these are typically characterized by a small number of methods and much fewer lines of code. The other three—the two Vector components, and the Map component—are Sulu versions of Java classes, typified by a large number of methods, and lines of code.

We wish to determine whether the size of the components has an effect on the adequacy coverage results, and whether the interaction between size and test suite is also a significant factor. We thus performed a two-way analysis of variance, with test suite as one factor, and size as another factor; size having two possible values: “large” for the Java-like components and “small” for the Resolve-like components. We have the following null hypotheses:

- $H_{0\text{suite}}$: The test suite has no significant effect on coverage
- $H_{0\text{size}}$: The size of the component has no significant effect on coverage
- $H_{0\text{suite} \times \text{size}}$: There is no significant interaction between test suite and the component’s size
Table 5.13: P-values for effects of test suite, size, and their interaction; values less than 0.05 are considered significant

<table>
<thead>
<tr>
<th>Coverage Criteria</th>
<th>Test Suite</th>
<th>Size</th>
<th>Test Suite × Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>statement</td>
<td>&lt; 0.0001</td>
<td>0.0031</td>
<td>0.0986</td>
</tr>
<tr>
<td>decision</td>
<td>&lt; 0.0001</td>
<td>&lt; 0.0001</td>
<td>0.1315</td>
</tr>
<tr>
<td>condition/decision</td>
<td>&lt; 0.0001</td>
<td>&lt; 0.0001</td>
<td>0.1771</td>
</tr>
<tr>
<td>delete statement</td>
<td>&lt; 0.0001</td>
<td>&lt; 0.0034</td>
<td>0.2565</td>
</tr>
<tr>
<td>force T/F</td>
<td>&lt; 0.0001</td>
<td>&lt; 0.0001</td>
<td>0.0371</td>
</tr>
<tr>
<td>change arithmetic op</td>
<td>&lt; 0.0001</td>
<td>0.0589</td>
<td>0.9009</td>
</tr>
<tr>
<td>change comparison op</td>
<td>&lt; 0.0001</td>
<td>0.0001</td>
<td>0.0249</td>
</tr>
<tr>
<td>change int constant</td>
<td>&lt; 0.0001</td>
<td>&lt; 0.0003</td>
<td>0.3715</td>
</tr>
<tr>
<td>change boolean op</td>
<td>0.6016</td>
<td>0.0998</td>
<td>0.9291</td>
</tr>
</tbody>
</table>

5.3.3 Relationships Between Coverage Measures

Two of our code coverage metrics have direct analogues to the implemented mutation coverage measures. These are: statement coverage and the “delete statement” mutator, and decision coverage and the “force conditionals to T/F” mutator. We expected the mutation coverage to be a finer grained measure of actual coverage of that particular statement or decision than the corresponding code coverage measure.

Indeed matched pairs t-tests of statement coverage and decision coverage versus their corresponding mutation operators on each of the test suites show that code coverage significantly reports higher coverage. That is, they report more coverage than their corresponding mutation kill ratios (at $\alpha = 0.05$). This is further confirmation that high code coverage measures...
may not mean high bug detecting capabilities. However, there is also a correlation between the code coverage metrics and the mutation operators—for statement coverage versus delete statement, correlation is at 0.827 and p-value < 0.0001 and for condition coverage versus force-conditional, correlation is at 0.746, with a p-value < 0.001—suggesting that higher code coverage means higher bug detection.

Thus, code coverage can still be a practical way of making a quick assessment of the effectiveness of test suites. That a section of code is covered by no means implies that the section has no bugs. However, if a section of code is not covered, this implies that bugs in that section will never be found. Thus code coverage can provide a baseline metric: how much of the code is not being tested by a test suite.

The main appeal of using code coverage to quantify the thoroughness of testing is that it can be gathered fairly quickly, especially in comparison to mutation testing. For example, decision coverage can be gathered in one execution of the test suite, no matter how many decisions there are. However, if there are \( m \) decisions in the software unit, the test suite has to be run on the order of \( m \) times. If the mutation tester flags the mutant killed on the first test case failure, as is the case in Sulu, assuming the mutant for which test cases fail are evenly distributed, the execution time is only cut by half (it is still \( O(m) \)).

In a software development environment, one can imagine that code coverage metrics are used in regular testing tasks, up to a certain point. That is, up to a certain code coverage bar, after which mutation testing can be periodically used to identify areas which code coverage tools report as covered but the mutation tool reports as not having been tested.

If this strategy is employed, we should ask if it is possible to create code coverage analogues to our other mutation operators, so that these can be used as quick proximate measures of the mutation kill ratio. Thus far, statement coverage is a stand-in for “delete statement,” and decision coverage is a stand-in for “force conditionals to T/F”. We have 4 other mutation operators to consider.
Three of these: change arithmetic operator to another, change a boolean operator to another, and change a comparison operator to another have straightforward analogues: arithmetic operator coverage, boolean operator coverage, and comparison operator coverage. That is, we keep track of every arithmetic, boolean, and comparison operator, and flag them as covered when they are called. In Sulu, we can store this information on the AST node of each operator, similar to what is already implemented for decision and condition/decision coverage.

A code coverage analogue for the “change integer constant” is a little more complex; similar to all the others, in Sulu we can hang information on an integer constant’s abstract syntax tree, and count that constant as covered when it is used, i.e., when it is passed as a parameter to a method, or used as part of an arithmetic computation. However, if the integer constant is assigned to a variable, we must ensure that the variable is subsequently used, and not immediately reassigned another value.

5.3.4 Threats to Validity

No experiment is perfect; it is the responsibility of the researcher to identify the threats to validity of their research. In this section we explore the internal, external, and construction threats to the validity of our experiment. These threats relate mainly to the size, number, and composition of the main inputs to the study.

Internal validity is threatened by the small sample size. With just 10 components we could only conclude that the all-triples with 2 parameter values test suite is better than every other test suite in 3 of the 9 metrics. And we found no significant differences among the test suites for the “change boolean operator” mutant kill ratio. A larger experiment may be used as further confirmation of our conclusions.

External validity is affected by the type and size of the software components we used. The set of reference components used here are collection classes; which suggests that the auto-
mated test case generations strategy works well for software components similar to those in
our reference set, but it may not be representative of other kinds of software components
(e.g., components that deal with I/O). The reference components themselves are small, with
relatively few lines of code. Our conclusions on the effectiveness of the test suites may not
be applicable to much larger software units. A larger experiment with more varied kinds of
components may tell us whether our conclusion still holds for a larger population of software.

Construction validity may be threatened by the coverage metrics we use; that is, our coverage
measures may not represent a measure that is related to real bugs. However, there is evidence
from work by Andrews and his colleagues [5, 6] that our mutation operators are correlated
to real bugs. While often criticized for not being a strong measure of the test effort, our
code coverage metrics are measures that are nevertheless widely used in industry.

A further threat to external validity is the practicality of applying these techniques to real-
world software development practices. One of the major difficulties of our techniques is time:
the number of test cases we generate increases exponentially as we increase the length of the
sequence and the number of parameter values. In addition, mutation testing is also time
intensive. In the next section, we explore some ways of managing the cost of running and
measuring the effectiveness of tests.

5.4 Managing Expensive Tests

The effectivity of the all-triples test suite with two parameter inputs comes with a heavy
cost in terms of running time. The bar graph in figure 5.8 shows the dramatically increased
execution time for running the most comprehensive test suite on all of the reference compo-
nents.

At just over 13 minutes to run all ten of the all-triples with 2 parameters test suites, this
can still be practical for the regular developer. However, running mutation analysis on all
these takes about a day. Thus, while all-triples has good coverage, pruning many of the test cases can possibly produce similar results at a greatly reduced time.

Doing an analysis on the exhaustive enumeration of all \( n \) methods with \( k \) parameters may not be practical for \( n \) and \( k \) larger than 2 or 3. However, while perhaps not practical for running them as a matter of course, these may still be useful for giving us an *upper bound* on the effectiveness of any test-suites that uses a subset of the exhaustive set.

One way to manage large numbers of test cases is to generate fewer test cases for the same coverage. Two mechanisms for doing this is to reduce the number of *invalid* tests, and secondly to reduce the number of *equivalent* or redundant tests. Xie and his colleagues have identified several notions of redundant unit tests, and have shown that identifying redundant tests can sometimes eliminate a large number of test cases. We explore a novel method of reducing the number or invalid tests by augmenting the specification language in Section 6.

The Sulu infrastructure certainly can support more different test-cases generators. If other
test-cases generators for Sulu are developed, it can be plugged in to the rest of the system, and evaluated against the already completed components and evaluated using existing code coverage and mutation analysis tools.

However, because of the large (often virtually infinite) domain of test cases, we can always generate more tests. Thus, the problem of executing long-running tests will need to be addressed for the foreseeable future. If we already have coverage measures, we can use them along with execution time information to gauge the efficiency of each test suite, we can then order the execution of these test suites according to their efficiency.

Figure 5.9 shows one efficiency measure for our test suite: dividing the percent coverage by the execution time of each test suite. Unfortunately, because of the exponential nature of our test suites, the most efficient test suite, Singles1, is also the test suite with the least coverage; while our test suite that has the most coverage is also the most inefficient. We
believe this is an artifact of the test cases generation, however. We certainly hope that it is possible to generate test suites that has both high coverage and high efficiency. Be that as it may, we can use the efficiency information to order the tests with the highest efficiency first. That is, for example, if we go by statement coverage efficiency, we run *Singles* 1 first, then *Singles* 2, etc., in the order they are shown in Figure 5.9.

In many software development practices (especially in test-driven development) every change in the source code requires rerunning the changed component’s unit tests (i.e., a regression test). Longer running tests, such as the *Triples* 2 test suite, may be impractical to run synchronously, as it can waste developer time. Instead, in practice, we can schedule the execution of long-running tests (e.g., overnight, or through the weekend), and only run the fast running, high efficiency tests during development. We can also adapt Saff and Ernst’s [78, 79] idea of continuous testing, by executing long-running tests, starting with the most efficient, in the background as the programmer is modifying the code.

## 5.5 Does It Really Work? A User’s Perspective

This researcher’s own personal experience can perhaps illuminate some of the benefits that can be derived from using Sulu and its automated testing tools. The three java.util-like components (the two Vector realizations, and the hashtable-based Map) were the most difficult to implement. Not only were they larger, this researcher also had to hand-translate Java code into Sulu. The differences in the language idioms made it a non-trivial task. While this author tried to very carefully translate line-by-line the programs, we are only human, and errors creep in.

When the translations were completed, a few ad-hoc tests were written and executed. After those were run and the few bugs that the ad-hoc tests revealed were fixed, we ran the automated testing tool on the realizations. Several additional bugs were found by running the automatically generated test suites. The bugs were either in the implementation, the
Figure 5.10: **ensureCapacity** is not fully covered by the generated test cases

specification, or in the interpreter itself. One of the most common bugs in the implementation typically involves forgetting to add a line (say incrementing the loop counter), which lends credence to the “delete a statement” mutation operator as reflective of real-life bugs.

A common specification bug is not handling what would typically cause exceptions in Java, but would cause nothing to happen in the translated version. However, a third application of running the automated tests emerged: it actually revealed quite a few bugs in the Sulu interpreter, and the underlying “built in” components such as arrays and pointers. Because the automatic test-cases generator exhaustively executes different combinations of method calls, it often calls a sequence of calls that are unexpected, and has on several occasions caused the interpreter to fail catastrophically. Thus, automated testing includes one side effect benefit: testing the testing tools themselves.

After running the automatically generated test cases, however, the code coverage tools still
reported some parts of the implementation were not covered. Figure 5.10 shows a snapshot of the testing tool after the all-pairs with 2 parameters test suite were run against the ArrayBased Vector component. It shows that one method in particular is not well covered: ensureCapacity.

The cause of the poor code coverage for that particular method became apparent after some inspection. The ensureCapacity method increases the size of the array that stores the objects as it is close to being filled. Because the initial capacity of the Vector is 10, even the all-triples test cases will only have a maximum of three elements in it. We then handcrafted 4 manual test cases in a test suite that that included inserting more than 10 items into the Vector. Running both the manual tests and the automated test suites provided us with 100% code coverage. It is worth noting that the automatically generated test cases gets complete code coverage on all 3 metrics for a version of Vector with an initial capacity of 1.

It is not surprising to find out that most of the code not covered by our test suites are precisely the parts of the code that is most complex—resizing the underlying array of an array-based list; rehashing a hashtable; rebalancing a heap. Because of the small number of method calls and parameters, our test suites are not particularly good at testing complex behavior. We view our test suites as implementing tests that are simple to come up with, but often tedious to implement. Thus, this author believes that we achieved the goal of creating an extra layer of automatic error checking beyond the static checking provided by a compiler.

In the end, the author had confidence in the implemented software components only after all the tests passed on the automatically generated test-cases with the largest sequence of method calls that was practical to execute—typically, all sequences of four method calls with two parameter values, which takes about two hours on an implementation of Vector. Several bugs were found, providing experiential evidence that complements the experimental results of the practical effectiveness of our automatically generated test cases. While the generated test cases provided us with a set of baseline tests, the immediate feedback of the code coverage
tools provided insight into the weaknesses of the automated test cases, and allowed us to develop a complementary test suite that addressed these weaknesses. We believe this is a data point that lends evidence to show that automated testing in the manner we presented in this thesis can be a practical and integrated part of the normal software development process.
Chapter 6

Minimizing Invalid Tests By Specifying Method Sequences

In the previous chapter, we have seen that for a certain class of automatically generated test suites, we can achieve high test adequacy coverage for a set of collection components. However, our test generation strategy of exhaustively enumerating all sequences of method calls generates a large number of test cases, many of which are invalid.

A common precondition for Sulu collection components requires that the collection is not empty when a `remove` method is called. In fact virtually all of the invalid test cases that were generated violated this precondition. We observed that the precondition violation could be avoided if a method calling protocol was followed: never call `remove` more times than `insert` has been called.

Thus, to lower the number of generated test cases, we could use a mechanism to formally specify this method calling protocol directly; which would allow us to statically determine which sequences of method calls are invalid, and not generate them in the first place.

In Section 4.2 we described our automated test case generation algorithm as a walk through a flowgraph. In our stack example (Figure 4.2), we noted that there should not be an edge
between \textit{init} and \textit{pop} since \textit{pop} can never be called after object initialization. Figure \ref{fig:stack-flowgraph} shows the flowgraph of a stack class without the edge from \textit{init} to \textit{pop}.

The idea for this flowgraph representation is that every object’s lifetime corresponds to some path from \textit{init} to \textit{finalize} in this graph, and the graph itself characterizes all possible object lifetimes. Thus, the path \textit{init} $\rightarrow$ \textit{push} $\rightarrow$ \textit{pop} $\rightarrow$ \textit{finalize} represents creating a stack object, calling \textit{push} on it, then calling \textit{pop}, then the object is finalized. Every path in the graph this also represents a test case.

There is no edge from \textit{init} to \textit{pop}; this is because there is no object lifetime where \textit{pop} can be called immediately after it is initialized. That is, calling \textit{pop} immediately after creating the stack \textit{always} violates the precondition that the stack must not be empty when \textit{pop} is being executed.

We can see, however, that disallowing the edge from \textit{init} to \textit{pop} is only a special case of a more general rule: that you cannot call \textit{pop} more times than you’ve called \textit{push} previously. This call sequence protocol of never removing more items than have been inserted is very prevalent in Sulu (and Resolve \cite{resolve} from which many of our collection components are based),
and thus we wish to have a mechanism to specify exactly which sequences of method calls are allowed. To achieve this, we propose to use context-free language reachability.

6.1 Context-Free Language Reachability

To specify allowable sequences of method calls, we adapt an example by Reps [76] who used CFL-reachability to define a superset of feasible computations. CFL-reachability involves a graph $G$ with labeled edges, and a context-free language $L$. A path in $G$ is an $L$-path only if the word formed by concatenating the edge labels of the path is a word in $L$.

We apply CFL-reachability in a straightforward manner. To specify a superset of allowed sequences of method calls on a software component, we create a CFL-reachability graph that is similar to the component’s flow graph. However, we include all edges from *init* to every method, from every method to *finalize*, and between each pair of methods, i.e., the construction we used in Section 4.2. Instead of removing infeasible edges, we define feasible paths using a context-free grammar.

The specifier then has to label the edges, and also define a context-free language $L$. The goal is to define the labeled graph $G$ and language $L$ such that a path in $G$, representing a sequence of method calls, generates a string; if the string is in the context-free language $L$, we recognize that sequence as potentially feasible, if it is not a string in $L$, we say that the sequence of method calls is infeasible.

As an example, let’s consider a simplified version of a stream reader class, with 3 methods: *open*, *read*, and *close*. The usual method calling protocol then applies: *open* must be called first on a stream, then *read* may be called multiple times, then *close* is called to close the stream. After *close* is called, either the object is finalized, or *open* is called again on another stream.

We construct the class’s CFL-reachability graph in the manner described above. In addition
to building the graph, we label every edge that goes to \textit{open} as the open parenthesis ‘(’; every edge that goes to \textit{close} as the close parenthesis ‘)’; and every edge that goes to \textit{read} as ‘r’ (all other edges are labeled with an empty string). The final constructed graph with labels is shown as Figure 6.2. Using this graph, the sequence of method calls of \textit{open} → \textit{read} → \textit{close} thus produces a string “(r)”; and an invalid sequence of method calls like \textit{open} → \textit{close} → \textit{read} would produce a string “()r”.

We can quickly see that the context-free language \( L \) that accepts feasible sequences of method calls from paths in the graph is one that accepts a string of parenthesized r’s equivalent to the regular expression \[ ‘(’ r\(^*\) ’)’ \]^*. Expressing this in BNF:
The language defined by \textit{parenr} is the CFL for our feasible sequence of method calls. It disallows a number of invalid sequences of methods such as calling \textit{open} or \textit{close} twice, and calling \textit{read} before an \textit{open} call.

### 6.2 Specifying Method Sequences for Collection Classes

The stream reader example required a simple regular expression to specify feasible sequences of method calls. A more complicated yet common method calling protocol, however, requires the broader scope of context-free languages. The stack component is one example. In general, we want to specify for a \textit{collection} class, that the equivalent of a \textit{remove} method cannot be called more times than an \textit{insert} method.

We shall show an example for specifying the feasible method sequences for a \textit{stack} component. Although this example is for stacks, it should be apparent how this can be similarly used for other collection components.

We begin by creating the stack’s CFL-reachability graph (Figure 4.2), and label the edges in this manner: for every edge that goes into \textit{push} we label it with an open parenthesis ‘(‘. And for every edge that goes into \textit{pop}, we label it with a close parenthesis ‘)‘. All other edges are labeled with an empty string. Figure 6.3 shows the graph with its labels.

The basic idea is to have every \textit{pop} call match a previous \textit{push} call. And we can see that
this idea corresponds to the language of partially matched nested parentheses. That is, you can have more \texttt{push} calls than \texttt{pop}, but not the other way around. More formally, we define a language \textit{pmatch}:

\[
\text{pmatch} \rightarrow \text{match pmatch} \\
| \ ( \text{pmatch} \\
| \ \epsilon \\
\text{match} \rightarrow \text{match match} \\
| \ ( \text{match} ) \\
| \ \epsilon
\]

The language \textit{pmatch} can be used for a wide array of collection components, simply label
the edges of the flowgraph that end in an inserting method with ‘(’ and every method that removes an element with ‘)’.

A variation on the simple collection object is a Sorting Machine. A Sorting Machine component recasts the sorting algorithm into an ADT [8]. Its basic usage pattern is that elements are inserted into a sorting machine by calling the insert method, then the sort method is called, and finally a remove method is called repeatedly (remove returns the smallest remaining object in the collection).

This is slightly similar to the stack component in that we still want every remove call to match a previous insert call. However, we must also make sure that sort was called prior to the remove call, but after every insert call. That is, we want to require that the sorting machine’s elements are sorted when remove is executed.

It turns out that we can modify the stack graph and its CFL slightly to specify the feasible method sequences of a sorting machine. We begin with the usual CFL-reachability graph, and label each edge going to insert as ‘(’ and each edge going to remove as ‘)’. We also label every edge going to sort as s. Figure 6.2 shows the graph. The sorting machine may have other methods such as querying the number of elements in the machine, and whether it’s sorted or not. We collapse those nodes into one (labeled “...”) for simplicity, noting that as long as the labeling rules are followed, the specification still works.

We then define the CFL for the language where for every close parenthesis ‘)’, no open parenthesis ‘(’ precedes it that is not followed by s:
Figure 6.4: CFL-reachability graph for a sorting machine

\[
\begin{align*}
\text{psmatch} & \rightarrow \text{smatch pmatch} \\
& \quad | ( \text{psmatch} ) \\
& \quad | \epsilon \\
\text{smatch} & \rightarrow \text{smatch smatch} \\
& \quad | ( \text{smatch} ) \\
& \quad | s
\end{align*}
\]

The language defined by \textit{psmatch} provides us exactly this specification.
6.3 Applications for Automated Testing

Just like black-box flowgraphs, our CFL-reachability graphs represents all object lifetimes. That is, every sequence of method calls can be represented as a path in the graph. However, the addition of a context-free language that defines a superset of allowed call sequences on the object provides us with a richer context to use in automated testing by preserving more of the information that is lost by “superimposing” all the lifetimes into one set of graph edges.

Generating method sequences using our current approach often degenerates to enumerating all \( n^k \) sequences of method calls of length \( k \). By allowing a more nuanced model of an object’s lifetime, generating method sequences using CFL-reachability offers a significant benefit by enumerating fewer method sequences viewed as invalid.

CFL-reachability graphs suggests several test adequacy criteria that can be used both to generate tests and evaluate the test effort. A test adequacy criteria that is direct translation from the flowgraph model might be all \( L \)-paths with \( k \) edges—for example, \( k = 3 \) means all pairs of method calls that is allowed by the context-free language \( L \). It is also possible to generate test cases from the BNF itself; for example, all sentences of length \( k \), or a set of sentences that exercises every term in the BNF. These test generation mechanisms all avoid nearly all of the invalid test cases we previously generated using the flowgraph approach.

For test cases not directly generated from the CFL, we can also apply dynamic checking of the string generated by following the edges of the flowgraph. That is, for every object that is to be checked, the runtime system can keep track of the previous method called, and generate the label for the edge in the flow graph in the next method call, and have a parser check the validity of the string as the method is called.

One application of this dynamic checking is as a partial oracle for test cases. It is a fairly
common practice, for example, to limit DBC checking to only preconditions to detect pre-
condition violations by the underlying software component. In the same manner, we can use
this CFL-reachability approach to dynamically check the compliance of underlying software
to method call sequence protocols. Assuming test cases were not generated using the soft-
ware under test’s associated context-free language, dynamic checking of the string composed
via calling the methods of the component under test can detect invalid test cases.

A further application could be to use dynamic checking to detect loose preconditions. A
behavioral (using pre- and postconditions, invariants, etc.) specification of a software com-
ponent should already implicitly encode which method call sequences are disallowed in their
preconditions (e.g., for a stack, that the stack should not be empty when \texttt{pop} is called).
That is, the sequence specification is only a redundant, partial specification of the software
component.

Because it is redundant, however, if dynamic checking of the sequence specification is en-
abled along with precondition checking, a sequence spec failure that does not coincide with
precondition failure can be flagged as a spec mismatch. The sequence that was flagged
indicates either the precondition needs to be stronger such that it fails on that sequence,
or that the sequence specification itself is wrong. Thus, our approach could be used as an
additional layer of checking not only the correctness of the underlying component, but also
of its behavioral specification.
6.4 Alternative Method Sequence Specification Mechanisms

The idea of having a direct specification of allowed sequences of method calls was inspired by the work of Cheon and Perumandla [26] on extending the Java Modeling Language to specify method sequences. In turn, their work was influenced by research on trace assertions in Jass [10]. Cheon and Perumandla’s specification language uses a regular-expression like syntax, and thus does not have the expressiveness needed to specify Sulu’s collection components’ method calling protocol.

Bartussek and Parnas [11] originated the concept of trace assertions. The trace assertion method is meant to be a mechanism to fully specify the behavior of software modules. The CFL-reachability approach in this paper is meant only to specify method sequences.

Although originally used for concurrency control, Campbell’s path expressions [24] can be also used for specifying sequences of procedures. Path expressions have been implemented in Pascal [25] as early as 1979. Basic path expressions have the same expressive power as regular expressions.

Predicate path expressions [4] augment basic path expressions, and is in the realm of context-free languages. Predicate path expressions have a regular-expression like syntax, but each procedure is predicated by an assertion on implicit counters on the number of times a procedure has been called. In essence the predicates act as a filter to the regular language.

Although having the same expressive power, applying predicate path expressions to Sulu will require the enumeration of all methods within the regular language. Our CFL approach allows us to elide methods that do not directly affect the method calling protocol. Path expressions will also require the redefinition of the path expression for subclasses with new
methods; while in our CFL approach, we expect that most subclasses will simply inherit and reuse the language of their parent classes.

Our application of context-free language reachability has been influenced particularly by Reps’s description of its use in program analysis [76]. We took Reps’s description of using CFL-reachability to determine a superset of feasible computation paths and applied it to Edwards’s use of a flowgraph as a black-box representation of a software component [39]. In our approach, however, we require the specifier to label the graph’s edges rather than determining the label from either the structure of the program or its behavioral specification.
Chapter 7

Conclusions and Future Research

7.1 Integrating Automated Testing

The Sulu language and tools provides an integrated view of unit testing. It is a view where the various stages of testing: generation, execution, and evaluation, is integrated into a single platform. It is also a view that encompasses the integration between human and mechanical testing; between different test case generation algorithms; and between different test effort evaluation mechanisms. Finally, it is a view where the testing process is integrated with the programming language, where testing issues are tackled from the design of the language, up to the execution of the software written in that language.

Integration of the stages of unit testing means that for every software component, we can automatically generate tests for it, and immediately execute and evaluate those tests for their thoroughness. Integration between test generation algorithms lets the programmer choose or implement the best test case generation algorithm for the class of software components he is interested in.
Integration between mechanically- and human-generated tests means that armed with the evaluation data, the tester can augment the weaknesses of the automatically generated test cases with his own unit tests. That is, the programmer is enabled by the automated unit testing system to focus on the often challenging parts where human attention is most needed. And integration among test evaluation metrics allows programmers to use the metric that most represents the kind of thoroughness that the test effort is meant to achieve.

This dissertation also espouses a view of programming languages that integrates unit testing into the programming language. Testing is an essential part of the software development process; we should therefore design programming languages and their runtimes that are cognizant of this fact—that our programming languages will not only be used for building software, but also for testing them.

Sulu is a research language; we do not expect the next missile control system to be written in Sulu anytime soon. However, we believe that many parts of this vision of integrated software unit testing if implemented in current software development tools, will be a practical and effective part of the software development process.

7.2 Future Research

It is our belief that this research and the Sulu language and provides a foundation for future research. By providing a platform for the implementation of automated test cases generators, and mutation operators, we believe it can be used as a benchmark of sorts for different unit testing strategies. We present here some avenues for future research.
Larger Experiments

We would like to see the set of ten collection components used in the experiment expanded to include not only more collection components, but also different kinds of components. For example, it is unclear how the test suites from our test generation mechanism would fare with GUI-based components, or I/O based ones.

To be able to test these components, however, we will need to write specifications as well as implementations for them. While the specification of collection classes has been well established [90], it may be less clear how best to specify GUI and I/O based software components. Thus, along the way, we would need to discern how best to specify event-based and I/O bound components, especially for use in unit testing.

Different Test Generation Algorithms

Currently, Sulu has three different test generation plugins. However, the latest plugin can generate the same test suites as the other two. We would like to see more different test generation mechanisms implemented, including test generation based on the black-box adequacy criteria using CFL-reachability in Section 6. This would however require some additions to the specification language.

It will also be useful to reimplement in Sulu test generation mechanisms that were developed for other languages. This author and Edwards have advocated [82] a benchmark for automated unit test generation. This, along with an expanded set of reference software components would benefit researchers and adopters by making it easier to compare different test generations strategies. One of the testing strategies we briefly surveyed in Section 2.3 may be a good candidate for implementation.
Integration With Formal Verification Tools

The Sulu programming language is loosely based on Resolve. However, while Sulu was designed with software testing in mind, Resolve was designed for formal verification. Leavens and his colleagues [61] argue that these may not be competing goals; that is, a specification language used for runtime assertion checking (like in Sulu) may still be useful for formal verification.

Sulu provides many of the features found in Resolve such as value semantics, support for generics, separation of concepts from realizations, etc.. It may be a worthwhile endeavor to explore other linkages between recent work on Resolve tools [77] with Sulu.

Programming Language Design and the Practice of Programming

Because the focus of this research is primarily on the software testing aspects, we have deferred many programming language issues that remain with Sulu. For example, further exploration of the impact of value semantics on program construction could be a future direction. Similarly, the use of the “matching” relationship instead of subtyping should be addressed more carefully (see Section B.2).

From the design perspective, a more thorough review of the type system might provide useful insights; a proof of type safety (and corresponding changes to the type system, to make it happen) may be a useful endeavor. The construction of a compiler for Sulu may provide some valuable insights as well, including tying in strands of previous research by this author [81] and his adviser [42], on dynamically wrapping assertion checks to support the same kind of specification checking as that of the currently implemented interpreter.
7.3 Conclusion

The Sulu language and tools provide an integrated platform for the automated generation, execution, and evaluation of unit tests. We presented a vision for the integration of automated testing in the software development process; and developed a proof-of-concept language and tools for the kind of automation we envision.

We performed an experiment to evaluate six test suites from a family of test suites generated by a test generation algorithm. This experiment demonstrates that we can achieve high adequacy criteria coverage by using the Sulu tools to generate test suites. These results, coupled with the ability in Sulu of the programmer to augment the mechanically generated tests with human-written ones, give evidence that automated testing along the lines of our testing vision gives rise to more reliable software.
## Appendix A

### Evaluation Data

#### A.1 Code Coverage Tables

<table>
<thead>
<tr>
<th>Concept</th>
<th>Realization</th>
<th>methods</th>
<th>statements</th>
<th>decision count</th>
<th>c/d count</th>
</tr>
</thead>
<tbody>
<tr>
<td>BinaryTree</td>
<td>Standard</td>
<td>3</td>
<td>19</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>List</td>
<td>TwoStacks</td>
<td>5</td>
<td>15</td>
<td>6</td>
<td>12</td>
</tr>
<tr>
<td>Map</td>
<td>Hashtable</td>
<td>9</td>
<td>133</td>
<td>44</td>
<td>116</td>
</tr>
<tr>
<td>Sorter</td>
<td>BubbleSort</td>
<td>3</td>
<td>29</td>
<td>10</td>
<td>20</td>
</tr>
<tr>
<td>Sorter</td>
<td>HeapBased</td>
<td>5</td>
<td>63</td>
<td>22</td>
<td>46</td>
</tr>
<tr>
<td>Sorter</td>
<td>ListBased</td>
<td>5</td>
<td>18</td>
<td>6</td>
<td>14</td>
</tr>
<tr>
<td>Sorter</td>
<td>MinFinding</td>
<td>5</td>
<td>13</td>
<td>4</td>
<td>8</td>
</tr>
<tr>
<td>Stack</td>
<td>LinkedList</td>
<td>5</td>
<td>14</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Vector</td>
<td>ArrayBased</td>
<td>18</td>
<td>139</td>
<td>60</td>
<td>138</td>
</tr>
<tr>
<td>Vector</td>
<td>LinkedList</td>
<td>18</td>
<td>157</td>
<td>50</td>
<td>120</td>
</tr>
<tr>
<td>TOTAL</td>
<td></td>
<td>76</td>
<td>600</td>
<td>204</td>
<td>478</td>
</tr>
</tbody>
</table>
## Code Coverage All Singles With 1 Parameter Value

<table>
<thead>
<tr>
<th>Concept</th>
<th>Realization</th>
<th>statement</th>
<th>%</th>
<th>decision</th>
<th>%</th>
<th>cond./dec.</th>
<th>%</th>
</tr>
</thead>
<tbody>
<tr>
<td>BinaryTree</td>
<td>Standard</td>
<td>12</td>
<td>63.16</td>
<td>1</td>
<td>50.00</td>
<td>2</td>
<td>50.00</td>
</tr>
<tr>
<td></td>
<td>TwoStacks</td>
<td>8</td>
<td>53.33</td>
<td>2</td>
<td>33.33</td>
<td>4</td>
<td>33.33</td>
</tr>
<tr>
<td></td>
<td>HashTable</td>
<td>75</td>
<td>56.39</td>
<td>15</td>
<td>34.09</td>
<td>39</td>
<td>33.62</td>
</tr>
<tr>
<td>Sorter</td>
<td>BubbleSort</td>
<td>19</td>
<td>65.52</td>
<td>5</td>
<td>50.00</td>
<td>10</td>
<td>50.00</td>
</tr>
<tr>
<td></td>
<td>HeapBased</td>
<td>11</td>
<td>17.46</td>
<td>1</td>
<td>4.55</td>
<td>2</td>
<td>4.35</td>
</tr>
<tr>
<td></td>
<td>ListBased</td>
<td>9</td>
<td>50.00</td>
<td>2</td>
<td>33.33</td>
<td>5</td>
<td>37.71</td>
</tr>
<tr>
<td>Sorter</td>
<td>MinFinding</td>
<td>3</td>
<td>23.08</td>
<td>0</td>
<td>0.00</td>
<td>0</td>
<td>0.00</td>
</tr>
<tr>
<td>Stack</td>
<td>LinkedList</td>
<td>8</td>
<td>57.14</td>
<td>0</td>
<td>N/A</td>
<td>0</td>
<td>N/A</td>
</tr>
<tr>
<td>Vector</td>
<td>ArrayBased</td>
<td>87</td>
<td>62.59</td>
<td>23</td>
<td>38.33</td>
<td>55</td>
<td>39.86</td>
</tr>
<tr>
<td>Vector</td>
<td>LinkedList</td>
<td>96</td>
<td>61.15</td>
<td>17</td>
<td>34.00</td>
<td>44</td>
<td>36.67</td>
</tr>
<tr>
<td>TOTAL</td>
<td></td>
<td>328</td>
<td>54.67</td>
<td>66</td>
<td>32.35</td>
<td>161</td>
<td>33.68</td>
</tr>
</tbody>
</table>

## Code Coverage For All Pairs With 1 Parameter Value

<table>
<thead>
<tr>
<th>Concept</th>
<th>Realization</th>
<th>statement</th>
<th>%</th>
<th>decision</th>
<th>%</th>
<th>cond./dec.</th>
<th>%</th>
</tr>
</thead>
<tbody>
<tr>
<td>BinaryTree</td>
<td>Standard</td>
<td>18</td>
<td>94.74</td>
<td>1</td>
<td>50.00</td>
<td>2</td>
<td>50.00</td>
</tr>
<tr>
<td></td>
<td>TwoStacks</td>
<td>12</td>
<td>80.00</td>
<td>3</td>
<td>50.00</td>
<td>6</td>
<td>50.00</td>
</tr>
<tr>
<td></td>
<td>HashTable</td>
<td>94</td>
<td>70.68</td>
<td>28</td>
<td>63.64</td>
<td>76</td>
<td>65.52</td>
</tr>
<tr>
<td>Sorter</td>
<td>BubbleSort</td>
<td>20</td>
<td>68.97</td>
<td>5</td>
<td>50.00</td>
<td>10</td>
<td>50.00</td>
</tr>
<tr>
<td></td>
<td>HeapBased</td>
<td>32</td>
<td>50.79</td>
<td>6</td>
<td>27.27</td>
<td>12</td>
<td>26.09</td>
</tr>
<tr>
<td></td>
<td>ListBased</td>
<td>15</td>
<td>83.33</td>
<td>4</td>
<td>66.67</td>
<td>9</td>
<td>64.29</td>
</tr>
<tr>
<td>Sorter</td>
<td>MinFinding</td>
<td>9</td>
<td>69.23</td>
<td>1</td>
<td>25.00</td>
<td>2</td>
<td>25.00</td>
</tr>
<tr>
<td>Stack</td>
<td>LinkedList</td>
<td>14</td>
<td>100.00</td>
<td>0</td>
<td>N/A</td>
<td>0</td>
<td>N/A</td>
</tr>
<tr>
<td>Vector</td>
<td>ArrayBased</td>
<td>116</td>
<td>83.45</td>
<td>43</td>
<td>71.67</td>
<td>98</td>
<td>71.01</td>
</tr>
<tr>
<td>Vector</td>
<td>LinkedList</td>
<td>137</td>
<td>87.26</td>
<td>37</td>
<td>74.00</td>
<td>86</td>
<td>71.68</td>
</tr>
<tr>
<td>TOTAL</td>
<td></td>
<td>467</td>
<td>77.83</td>
<td>128</td>
<td>62.75</td>
<td>173</td>
<td>62.97</td>
</tr>
</tbody>
</table>
### Code Coverage For All Triples With 1 Parameter Value

<table>
<thead>
<tr>
<th>Concept</th>
<th>Realization</th>
<th>statement</th>
<th>%</th>
<th>decision</th>
<th>%</th>
<th>cond./dec.</th>
<th>%</th>
</tr>
</thead>
<tbody>
<tr>
<td>BinaryTree</td>
<td>Standard</td>
<td>18</td>
<td>94.74</td>
<td>1</td>
<td>50.00</td>
<td>2</td>
<td>50.00</td>
</tr>
<tr>
<td>List</td>
<td>TwoStacks</td>
<td>15</td>
<td>100.00</td>
<td>6</td>
<td>100.00</td>
<td>12</td>
<td>100.00</td>
</tr>
<tr>
<td>Map</td>
<td>Hashtable</td>
<td>94</td>
<td>70.68</td>
<td>28</td>
<td>63.64</td>
<td>76</td>
<td>66.67</td>
</tr>
<tr>
<td>Sorter</td>
<td>BubbleSort</td>
<td>27</td>
<td>93.10</td>
<td>8</td>
<td>80.00</td>
<td>16</td>
<td>80.00</td>
</tr>
<tr>
<td>Sorter</td>
<td>HeapBased</td>
<td>47</td>
<td>74.60</td>
<td>11</td>
<td>50.00</td>
<td>22</td>
<td>45.83</td>
</tr>
<tr>
<td>Sorter</td>
<td>ListBased</td>
<td>15</td>
<td>83.33</td>
<td>4</td>
<td>66.67</td>
<td>9</td>
<td>62.50</td>
</tr>
<tr>
<td>Sorter</td>
<td>MinFinding</td>
<td>12</td>
<td>92.31</td>
<td>3</td>
<td>75.00</td>
<td>6</td>
<td>75.00</td>
</tr>
<tr>
<td>Stack</td>
<td>LinkedList</td>
<td>14</td>
<td>100.00</td>
<td>0</td>
<td>N/A</td>
<td>0</td>
<td>N/A</td>
</tr>
<tr>
<td>Vector</td>
<td>ArrayBased</td>
<td>117</td>
<td>84.17</td>
<td>44</td>
<td>73.33</td>
<td>100</td>
<td>71.79</td>
</tr>
<tr>
<td>Vector</td>
<td>LinkedList</td>
<td>137</td>
<td>87.26</td>
<td>37</td>
<td>74.00</td>
<td>86</td>
<td>70.00</td>
</tr>
<tr>
<td>TOTAL</td>
<td></td>
<td>496</td>
<td>82.67</td>
<td>142</td>
<td>69.61</td>
<td>329</td>
<td>71.67</td>
</tr>
</tbody>
</table>

### Code Coverage For All Singles With 2 Parameter Values

<table>
<thead>
<tr>
<th>Concept</th>
<th>Realization</th>
<th>statement</th>
<th>%</th>
<th>decision</th>
<th>%</th>
<th>cond./dec.</th>
<th>%</th>
</tr>
</thead>
<tbody>
<tr>
<td>BinaryTree</td>
<td>Standard</td>
<td>13</td>
<td>68.42</td>
<td>2</td>
<td>100.00</td>
<td>4</td>
<td>100.00</td>
</tr>
<tr>
<td>List</td>
<td>TwoStacks</td>
<td>8</td>
<td>53.33</td>
<td>2</td>
<td>33.33</td>
<td>4</td>
<td>33.33</td>
</tr>
<tr>
<td>Map</td>
<td>Hashtable</td>
<td>76</td>
<td>57.14</td>
<td>16</td>
<td>36.36</td>
<td>41</td>
<td>35.34</td>
</tr>
<tr>
<td>Sorter</td>
<td>BubbleSort</td>
<td>19</td>
<td>65.52</td>
<td>5</td>
<td>50.00</td>
<td>10</td>
<td>50.00</td>
</tr>
<tr>
<td>Sorter</td>
<td>HeapBased</td>
<td>11</td>
<td>17.46</td>
<td>1</td>
<td>4.55</td>
<td>2</td>
<td>4.35</td>
</tr>
<tr>
<td>Sorter</td>
<td>ListBased</td>
<td>9</td>
<td>50.00</td>
<td>2</td>
<td>33.33</td>
<td>5</td>
<td>35.71</td>
</tr>
<tr>
<td>Sorter</td>
<td>MinFinding</td>
<td>3</td>
<td>23.08</td>
<td>0</td>
<td>0.00</td>
<td>0</td>
<td>0.00</td>
</tr>
<tr>
<td>Stack</td>
<td>LinkedList</td>
<td>8</td>
<td>57.14</td>
<td>0</td>
<td>N/A</td>
<td>0</td>
<td>N/A</td>
</tr>
<tr>
<td>Vector</td>
<td>ArrayBased</td>
<td>96</td>
<td>69.06</td>
<td>31</td>
<td>51.67</td>
<td>74</td>
<td>53.62</td>
</tr>
<tr>
<td>Vector</td>
<td>LinkedList</td>
<td>115</td>
<td>73.25</td>
<td>27</td>
<td>54.00</td>
<td>68</td>
<td>56.67</td>
</tr>
<tr>
<td>TOTAL</td>
<td></td>
<td>358</td>
<td>59.67</td>
<td>86</td>
<td>42.16</td>
<td>208</td>
<td>43.51</td>
</tr>
</tbody>
</table>
### Code Coverage For All Pairs With 2 Parameter Values

<table>
<thead>
<tr>
<th>Concept</th>
<th>Realization</th>
<th>statement</th>
<th>% statement</th>
<th>% decision</th>
<th>% cond./dec.</th>
<th>%</th>
</tr>
</thead>
<tbody>
<tr>
<td>BinaryTree</td>
<td>Standard</td>
<td>19</td>
<td>100.00</td>
<td>2</td>
<td>100.00</td>
<td>4</td>
</tr>
<tr>
<td>List</td>
<td>TwoStacks</td>
<td>12</td>
<td>80.00</td>
<td>3</td>
<td>50.00</td>
<td>6</td>
</tr>
<tr>
<td>Map</td>
<td>Hashtable</td>
<td>104</td>
<td>78.20</td>
<td>34</td>
<td>77.27</td>
<td>90</td>
</tr>
<tr>
<td>Sorter</td>
<td>BubbleSort</td>
<td>20</td>
<td>68.97</td>
<td>5</td>
<td>50.00</td>
<td>10</td>
</tr>
<tr>
<td>Sorter</td>
<td>HeapBased</td>
<td>33</td>
<td>52.38</td>
<td>7</td>
<td>31.82</td>
<td>14</td>
</tr>
<tr>
<td>Sorter</td>
<td>ListBased</td>
<td>18</td>
<td>100.00</td>
<td>6</td>
<td>100.00</td>
<td>14</td>
</tr>
<tr>
<td>Sorter</td>
<td>MinFinding</td>
<td>9</td>
<td>69.23</td>
<td>1</td>
<td>25.00</td>
<td>2</td>
</tr>
<tr>
<td>Stack</td>
<td>LinkedList</td>
<td>14</td>
<td>100.00</td>
<td>0</td>
<td>N/A</td>
<td>0</td>
</tr>
<tr>
<td>Vector</td>
<td>ArrayBased</td>
<td>128</td>
<td>92.09</td>
<td>54</td>
<td>90.00</td>
<td>124</td>
</tr>
<tr>
<td>Vector</td>
<td>LinkedList</td>
<td>153</td>
<td>97.45</td>
<td>46</td>
<td>92.00</td>
<td>109</td>
</tr>
<tr>
<td>TOTAL</td>
<td></td>
<td>510</td>
<td>85.00</td>
<td>158</td>
<td>77.45</td>
<td>373</td>
</tr>
</tbody>
</table>

### Code Coverage For All Triples With 2 Parameter Values

<table>
<thead>
<tr>
<th>Concept</th>
<th>Realization</th>
<th>statement</th>
<th>% statement</th>
<th>% decision</th>
<th>% cond./dec.</th>
<th>%</th>
</tr>
</thead>
<tbody>
<tr>
<td>BinaryTree</td>
<td>Standard</td>
<td>19</td>
<td>100.00</td>
<td>2</td>
<td>100.00</td>
<td>4</td>
</tr>
<tr>
<td>List</td>
<td>TwoStacks</td>
<td>15</td>
<td>100.00</td>
<td>6</td>
<td>100.00</td>
<td>12</td>
</tr>
<tr>
<td>Map</td>
<td>Hashtable</td>
<td>104</td>
<td>78.20</td>
<td>34</td>
<td>77.27</td>
<td>90</td>
</tr>
<tr>
<td>Sorter</td>
<td>BubbleSort</td>
<td>29</td>
<td>100.00</td>
<td>10</td>
<td>100.00</td>
<td>20</td>
</tr>
<tr>
<td>Sorter</td>
<td>HeapBased</td>
<td>48</td>
<td>76.19</td>
<td>12</td>
<td>54.55</td>
<td>24</td>
</tr>
<tr>
<td>Sorter</td>
<td>ListBased</td>
<td>18</td>
<td>100.00</td>
<td>6</td>
<td>100.00</td>
<td>14</td>
</tr>
<tr>
<td>Sorter</td>
<td>MinFinding</td>
<td>13</td>
<td>100.00</td>
<td>4</td>
<td>100.00</td>
<td>8</td>
</tr>
<tr>
<td>Stack</td>
<td>LinkedList</td>
<td>14</td>
<td>100.00</td>
<td>0</td>
<td>N/A</td>
<td>0</td>
</tr>
<tr>
<td>Vector</td>
<td>ArrayBased</td>
<td>129</td>
<td>92.81</td>
<td>55</td>
<td>91.67</td>
<td>126</td>
</tr>
<tr>
<td>Vector</td>
<td>LinkedList</td>
<td>153</td>
<td>97.45</td>
<td>46</td>
<td>92.00</td>
<td>109</td>
</tr>
<tr>
<td>TOTAL</td>
<td></td>
<td>542</td>
<td>90.33</td>
<td>175</td>
<td>85.78</td>
<td>407</td>
</tr>
</tbody>
</table>
## A.2 Mutation Coverage Tables

<table>
<thead>
<tr>
<th>Component</th>
<th>Realization</th>
<th>Mutants</th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>del stmt</td>
<td>chg arith</td>
<td>chg comp</td>
<td>chg bool</td>
<td>chg const</td>
</tr>
<tr>
<td>Sorter</td>
<td>MinFinding</td>
<td>13</td>
<td>4</td>
<td>10</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>Stack</td>
<td>LinkedList</td>
<td>13</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>6</td>
</tr>
<tr>
<td>List</td>
<td>TwoStacks</td>
<td>15</td>
<td>4</td>
<td>15</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Sorter</td>
<td>ListBased</td>
<td>18</td>
<td>4</td>
<td>10</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>BinaryTree</td>
<td>Standard</td>
<td>19</td>
<td>8</td>
<td>5</td>
<td>0</td>
<td>8</td>
</tr>
<tr>
<td>Sorter</td>
<td>BubbleSort</td>
<td>28</td>
<td>8</td>
<td>20</td>
<td>1</td>
<td>13</td>
</tr>
<tr>
<td>Sorter</td>
<td>HeapBased</td>
<td>63</td>
<td>8</td>
<td>60</td>
<td>1</td>
<td>17</td>
</tr>
<tr>
<td>Map</td>
<td>Hashtable</td>
<td>133</td>
<td>60</td>
<td>85</td>
<td>23</td>
<td>103</td>
</tr>
<tr>
<td>Vector</td>
<td>ArrayBased</td>
<td>139</td>
<td>88</td>
<td>140</td>
<td>13</td>
<td>100</td>
</tr>
<tr>
<td>Vector</td>
<td>LinkedList</td>
<td>156</td>
<td>52</td>
<td>115</td>
<td>15</td>
<td>57</td>
</tr>
<tr>
<td><strong>TOTAL</strong></td>
<td></td>
<td><strong>597</strong></td>
<td><strong>236</strong></td>
<td><strong>460</strong></td>
<td><strong>56</strong></td>
<td><strong>308</strong></td>
</tr>
</tbody>
</table>

### Mutation Coverage: All Singles With 1 Parameter Value (%)

<table>
<thead>
<tr>
<th>Component</th>
<th>Realization</th>
<th>del stmt</th>
<th>chg arith</th>
<th>chg comp</th>
<th>chg bool</th>
<th>chg const</th>
<th>force t/f</th>
</tr>
</thead>
<tbody>
<tr>
<td>BinaryTree</td>
<td>Standard</td>
<td>2 (10.53)</td>
<td>0 (0.00)</td>
<td>0 (0.00)</td>
<td>0 (N/A)</td>
<td>0 (0.00)</td>
<td>0 (0.00)</td>
</tr>
<tr>
<td>List</td>
<td>TwoStacks</td>
<td>2 (13.33)</td>
<td>2 (50.00)</td>
<td>3 (20.00)</td>
<td>0 (N/A)</td>
<td>0 (N/A)</td>
<td>1 (16.67)</td>
</tr>
<tr>
<td>Map</td>
<td>Hashtable</td>
<td>35 (26.32)</td>
<td>19 (31.67)</td>
<td>12 (14.12)</td>
<td>13 (56.52)</td>
<td>26 (25.24)</td>
<td>12 (27.27)</td>
</tr>
<tr>
<td>Sorter</td>
<td>BubbleSort</td>
<td>7 (25.00)</td>
<td>2 (25.00)</td>
<td>6 (30.00)</td>
<td>0 (0.00)</td>
<td>5 (38.46)</td>
<td>3 (30.00)</td>
</tr>
<tr>
<td>Sorter</td>
<td>HeapBased</td>
<td>6 (9.52)</td>
<td>0 (0.00)</td>
<td>3 (5.00)</td>
<td>0 (0.00)</td>
<td>2 (11.76)</td>
<td>1 (4.55)</td>
</tr>
<tr>
<td>Sorter</td>
<td>ListBased</td>
<td>4 (22.22)</td>
<td>2 (50.00)</td>
<td>3 (30.00)</td>
<td>2 (66.67)</td>
<td>2 (100.00)</td>
<td>2 (33.33)</td>
</tr>
<tr>
<td>Sorter</td>
<td>MinFinding</td>
<td>2 (15.38)</td>
<td>2 (50.00)</td>
<td>0 (0.00)</td>
<td>0 (N/A)</td>
<td>0 (0.00)</td>
<td>0 (0.00)</td>
</tr>
<tr>
<td>Stack</td>
<td>LinkedList</td>
<td>5 (38.46)</td>
<td>0 (N/A)</td>
<td>0 (N/A)</td>
<td>0 (N/A)</td>
<td>3 (50.00)</td>
<td>0 (N/A)</td>
</tr>
<tr>
<td>Vector</td>
<td>ArrayBased</td>
<td>29 (20.86)</td>
<td>8 (9.09)</td>
<td>24 (17.14)</td>
<td>4 (30.77)</td>
<td>14 (14.00)</td>
<td>9 (15.00)</td>
</tr>
<tr>
<td>Vector</td>
<td>LinkedList</td>
<td>53 (33.97)</td>
<td>2 (3.85)</td>
<td>16 (13.91)</td>
<td>3 (20.00)</td>
<td>7 (12.28)</td>
<td>11 (22.00)</td>
</tr>
<tr>
<td><strong>TOTAL</strong></td>
<td></td>
<td><strong>145</strong></td>
<td><strong>37</strong></td>
<td><strong>67</strong></td>
<td><strong>22</strong></td>
<td><strong>59</strong></td>
<td><strong>39</strong></td>
</tr>
</tbody>
</table>
### Mutation Coverage: All Pairs With 1 Parameter Value (%)

<table>
<thead>
<tr>
<th>Concept</th>
<th>Realization</th>
<th>del stmt</th>
<th>chg arith</th>
<th>chg comp</th>
<th>chg bool</th>
<th>chg const</th>
<th>force t/f</th>
</tr>
</thead>
<tbody>
<tr>
<td>BinaryTree</td>
<td>Standard</td>
<td>6 (31.58)</td>
<td>4 (50.00)</td>
<td>0 (0.00)</td>
<td>0 (N/A)</td>
<td>3 (37.50)</td>
<td>0 (0.00)</td>
</tr>
<tr>
<td>List</td>
<td>TwoStacks</td>
<td>7 (46.67)</td>
<td>4 (100.00)</td>
<td>3 (20.00)</td>
<td>0 (N/A)</td>
<td>0 (N/A)</td>
<td>1 (16.67)</td>
</tr>
<tr>
<td>Map</td>
<td>Hashtable</td>
<td>62 (46.62)</td>
<td>23 (38.33)</td>
<td>34 (40.00)</td>
<td>16 (69.57)</td>
<td>32 (31.07)</td>
<td>21 (47.73)</td>
</tr>
<tr>
<td>Sorter</td>
<td>BubbleSort</td>
<td>8 (28.57)</td>
<td>3 (37.50)</td>
<td>8 (40.00)</td>
<td>0 (0.00)</td>
<td>6 (46.15)</td>
<td>3 (30.00)</td>
</tr>
<tr>
<td>Sorter</td>
<td>HeapBased</td>
<td>18 (28.57)</td>
<td>4 (50.00)</td>
<td>10 (16.67)</td>
<td>0 (0.00)</td>
<td>10 (58.82)</td>
<td>4 (18.18)</td>
</tr>
<tr>
<td>Sorter</td>
<td>ListBased</td>
<td>10 (55.56)</td>
<td>3 (75.00)</td>
<td>3 (30.00)</td>
<td>2 (66.67)</td>
<td>2 (100.00)</td>
<td>2 (33.33)</td>
</tr>
<tr>
<td>Sorter</td>
<td>MinFinding</td>
<td>4 (30.77)</td>
<td>3 (75.00)</td>
<td>3 (30.00)</td>
<td>0 (N/A)</td>
<td>1 (50.00)</td>
<td>1 (25.00)</td>
</tr>
<tr>
<td>Stack</td>
<td>LinkedList</td>
<td>9 (69.23)</td>
<td>0 (N/A)</td>
<td>0 (N/A)</td>
<td>5 (83.33)</td>
<td>0 (N/A)</td>
<td></td>
</tr>
<tr>
<td>Vector</td>
<td>ArrayBased</td>
<td>56 (40.29)</td>
<td>36 (40.91)</td>
<td>47 (33.57)</td>
<td>8 (61.54)</td>
<td>37 (37.00)</td>
<td>24 (40.00)</td>
</tr>
<tr>
<td>Vector</td>
<td>LinkedList</td>
<td>92 (58.97)</td>
<td>20 (38.46)</td>
<td>38 (33.04)</td>
<td>7 (46.67)</td>
<td>24 (42.11)</td>
<td>25 (50.00)</td>
</tr>
<tr>
<td>TOTAL</td>
<td></td>
<td>272 (45.56)</td>
<td>100 (42.37)</td>
<td>146 (31.74)</td>
<td>33 (58.93)</td>
<td>120 (39.71)</td>
<td>81 (39.71)</td>
</tr>
</tbody>
</table>

### Mutation Coverage: All Triples With 1 Parameter Value (%)

<table>
<thead>
<tr>
<th>Concept</th>
<th>Realization</th>
<th>del stmt</th>
<th>chg arith</th>
<th>chg comp</th>
<th>chg bool</th>
<th>chg const</th>
<th>force t/f</th>
</tr>
</thead>
<tbody>
<tr>
<td>BinaryTree</td>
<td>Standard</td>
<td>8 (42.11)</td>
<td>4 (50.00)</td>
<td>0 (0.00)</td>
<td>0 (N/A)</td>
<td>5 (62.50)</td>
<td>0 (0.00)</td>
</tr>
<tr>
<td>List</td>
<td>TwoStacks</td>
<td>9 (60.00)</td>
<td>4 (100.00)</td>
<td>4 (26.67)</td>
<td>0 (N/A)</td>
<td>0 (N/A)</td>
<td>2 (33.33)</td>
</tr>
<tr>
<td>Map</td>
<td>Hashtable</td>
<td>38 (28.57)</td>
<td>26 (43.33)</td>
<td>40 (47.06)</td>
<td>17 (73.91)</td>
<td>35 (33.98)</td>
<td>24 (54.55)</td>
</tr>
<tr>
<td>Sorter</td>
<td>BubbleSort</td>
<td>15 (53.57)</td>
<td>3 (37.50)</td>
<td>11 (55.00)</td>
<td>0 (0.00)</td>
<td>6 (46.15)</td>
<td>4 (40.00)</td>
</tr>
<tr>
<td>Sorter</td>
<td>HeapBased</td>
<td>28 (44.44)</td>
<td>6 (75.00)</td>
<td>16 (26.67)</td>
<td>0 (0.00)</td>
<td>16 (94.12)</td>
<td>6 (27.27)</td>
</tr>
<tr>
<td>Sorter</td>
<td>ListBased</td>
<td>10 (55.56)</td>
<td>4 (100.00)</td>
<td>3 (30.00)</td>
<td>2 (66.67)</td>
<td>2 (100.00)</td>
<td>2 (33.33)</td>
</tr>
<tr>
<td>Sorter</td>
<td>MinFinding</td>
<td>6 (46.15)</td>
<td>3 (75.00)</td>
<td>3 (30.00)</td>
<td>0 (N/A)</td>
<td>1 (50.00)</td>
<td>1 (25.00)</td>
</tr>
<tr>
<td>Stack</td>
<td>LinkedList</td>
<td>10 (76.92)</td>
<td>0 (N/A)</td>
<td>0 (N/A)</td>
<td>6 (100.00)</td>
<td>0 (N/A)</td>
<td></td>
</tr>
<tr>
<td>Vector</td>
<td>ArrayBased</td>
<td>56 (40.29)</td>
<td>41 (46.59)</td>
<td>47 (33.57)</td>
<td>8 (61.54)</td>
<td>43 (43.00)</td>
<td>24 (40.00)</td>
</tr>
<tr>
<td>Vector</td>
<td>LinkedList</td>
<td>98 (62.82)</td>
<td>24 (46.15)</td>
<td>38 (33.04)</td>
<td>7 (46.67)</td>
<td>27 (47.37)</td>
<td>25 (50.00)</td>
</tr>
<tr>
<td>TOTAL</td>
<td></td>
<td>278 (46.57)</td>
<td>115 (48.73)</td>
<td>162 (35.22)</td>
<td>34 (60.71)</td>
<td>141 (45.78)</td>
<td>88 (43.14)</td>
</tr>
</tbody>
</table>
### Mutation Coverage: All Singles With 2 Parameter Values (%)

<table>
<thead>
<tr>
<th>Concept</th>
<th>Realization</th>
<th>del stmt</th>
<th>chg arith</th>
<th>chg comp</th>
<th>chg bool</th>
<th>chg const</th>
<th>force t/f</th>
</tr>
</thead>
<tbody>
<tr>
<td>BinaryTree</td>
<td>Standard</td>
<td>8 (42.11)</td>
<td>4 (50.00)</td>
<td>0 (0.00)</td>
<td>0 (N/A)</td>
<td>3 (37.50)</td>
<td>0 (0.00)</td>
</tr>
<tr>
<td>List</td>
<td>TwoStacks</td>
<td>2 (13.33)</td>
<td>2 (50.00)</td>
<td>3 (20.00)</td>
<td>0 (N/A)</td>
<td>0 (N/A)</td>
<td>1 (16.67)</td>
</tr>
<tr>
<td>Map</td>
<td>Hashtable</td>
<td>39 (29.32)</td>
<td>25 (41.67)</td>
<td>13 (15.29)</td>
<td>13 (56.52)</td>
<td>29 (28.16)</td>
<td>13 (29.55)</td>
</tr>
<tr>
<td>Sorter</td>
<td>BubbleSort</td>
<td>7 (25.00)</td>
<td>2 (25.00)</td>
<td>6 (30.00)</td>
<td>0 (0.00)</td>
<td>5 (38.46)</td>
<td>3 (30.00)</td>
</tr>
<tr>
<td>Sorter</td>
<td>HeapBased</td>
<td>6 (9.52 )</td>
<td>0 (0.00)</td>
<td>3 (5.00 )</td>
<td>0 (0.00)</td>
<td>2 (11.76)</td>
<td>1 (4.55)</td>
</tr>
<tr>
<td>Sorter</td>
<td>ListBased</td>
<td>4 (22.22)</td>
<td>2 (50.00)</td>
<td>3 (30.00)</td>
<td>2 (66.67)</td>
<td>2 (100.00)</td>
<td>2 (33.33)</td>
</tr>
<tr>
<td>Sorter</td>
<td>MinFinding</td>
<td>2 (15.38)</td>
<td>2 (50.00)</td>
<td>0 (0.00)</td>
<td>0 (N/A)</td>
<td>0 (0.00)</td>
<td>0 (0.00)</td>
</tr>
<tr>
<td>Stack</td>
<td>LinkedList</td>
<td>6 (46.15)</td>
<td>0 (N/A)</td>
<td>0 (N/A)</td>
<td>0 (N/A)</td>
<td>3 (50.00)</td>
<td>0 (N/A)</td>
</tr>
<tr>
<td>Vector</td>
<td>ArrayBased</td>
<td>39 (28.06)</td>
<td>17 (19.32)</td>
<td>31 (22.14)</td>
<td>5 (38.46)</td>
<td>18 (18.00)</td>
<td>14 (23.33)</td>
</tr>
<tr>
<td>Vector</td>
<td>LinkedList</td>
<td>73 (46.79)</td>
<td>13 (25.00)</td>
<td>25 (21.74)</td>
<td>6 (40.00)</td>
<td>17 (29.82)</td>
<td>18 (36.00)</td>
</tr>
<tr>
<td>TOTAL</td>
<td></td>
<td>186 (31.16)</td>
<td>67 (28.39)</td>
<td>84 (18.26)</td>
<td>26 (46.43)</td>
<td>79 (25.65)</td>
<td>52 (25.49)</td>
</tr>
</tbody>
</table>

### Mutation Coverage: All Pairs With 2 Parameter Values (%)

<table>
<thead>
<tr>
<th>Concept</th>
<th>Realization</th>
<th>del stmt</th>
<th>chg arith</th>
<th>chg comp</th>
<th>chg bool</th>
<th>chg const</th>
<th>force t/f</th>
</tr>
</thead>
<tbody>
<tr>
<td>BinaryTree</td>
<td>Standard</td>
<td>15 (78.95)</td>
<td>8 (100.00)</td>
<td>4 (80.00)</td>
<td>0 (N/A)</td>
<td>6 (75.00)</td>
<td>2 (100.00)</td>
</tr>
<tr>
<td>List</td>
<td>TwoStacks</td>
<td>11 (73.33)</td>
<td>4 (100.00)</td>
<td>7 (46.67)</td>
<td>0 (N/A)</td>
<td>0 (N/A)</td>
<td>3 (50.00)</td>
</tr>
<tr>
<td>Map</td>
<td>Hashtable</td>
<td>92 (69.17)</td>
<td>28 (46.67)</td>
<td>56 (65.88)</td>
<td>19 (82.61)</td>
<td>46 (44.66)</td>
<td>33 (75.00)</td>
</tr>
<tr>
<td>Sorter</td>
<td>BubbleSort</td>
<td>9 (32.14)</td>
<td>3 (37.50)</td>
<td>8 (40.00)</td>
<td>0 (0.00)</td>
<td>6 (46.15)</td>
<td>3 (30.00)</td>
</tr>
<tr>
<td>Sorter</td>
<td>HeapBased</td>
<td>20 (31.75)</td>
<td>4 (50.00)</td>
<td>10 (16.67)</td>
<td>0 (0.00)</td>
<td>10 (58.82)</td>
<td>4 (18.18)</td>
</tr>
<tr>
<td>Sorter</td>
<td>ListBased</td>
<td>14 (77.78)</td>
<td>3 (75.00)</td>
<td>3 (30.00)</td>
<td>2 (66.67)</td>
<td>2 (100.00)</td>
<td>3 (50.00)</td>
</tr>
<tr>
<td>Sorter</td>
<td>MinFinding</td>
<td>5 (38.46)</td>
<td>3 (75.00)</td>
<td>3 (30.00)</td>
<td>0 (N/A)</td>
<td>1 (50.00)</td>
<td>1 (25.00)</td>
</tr>
<tr>
<td>Stack</td>
<td>LinkedList</td>
<td>12 (92.31)</td>
<td>0 (N/A)</td>
<td>0 (N/A)</td>
<td>0 (N/A)</td>
<td>5 (83.33)</td>
<td>0 (N/A)</td>
</tr>
<tr>
<td>Vector</td>
<td>ArrayBased</td>
<td>101 (72.66)</td>
<td>67 (76.14)</td>
<td>84 (60.00)</td>
<td>10 (76.92)</td>
<td>54 (54.00)</td>
<td>43 (71.67)</td>
</tr>
<tr>
<td>Vector</td>
<td>LinkedList</td>
<td>134 (85.90)</td>
<td>38 (73.08)</td>
<td>80 (69.57)</td>
<td>10 (66.67)</td>
<td>35 (61.40)</td>
<td>42 (84.00)</td>
</tr>
<tr>
<td>TOTAL</td>
<td></td>
<td>413 (69.18)</td>
<td>158 (66.95)</td>
<td>255 (55.43)</td>
<td>41 (73.21)</td>
<td>165 (53.57)</td>
<td>134 (65.69)</td>
</tr>
<tr>
<td>Concept</td>
<td>Realization</td>
<td>del stmt</td>
<td>chg arith</td>
<td>chg comp</td>
<td>chg bool</td>
<td>chg const</td>
<td>force t/f</td>
</tr>
<tr>
<td>-------------</td>
<td>-------------</td>
<td>----------</td>
<td>-----------</td>
<td>----------</td>
<td>----------</td>
<td>-----------</td>
<td>-----------</td>
</tr>
<tr>
<td>BinaryTree</td>
<td>Standard</td>
<td>17 (89.47)</td>
<td>8 (100.00)</td>
<td>4 (80.00)</td>
<td>0 (N/A)</td>
<td>8 (100.00)</td>
<td>2 (100.00)</td>
</tr>
<tr>
<td>List</td>
<td>TwoStacks</td>
<td>15 (100.00)</td>
<td>4 (100.00)</td>
<td>12 (80.00)</td>
<td>0 (N/A)</td>
<td>0 (N/A)</td>
<td>6 (100.00)</td>
</tr>
<tr>
<td>Map</td>
<td>Hashtable</td>
<td>95 (71.43)</td>
<td>31 (51.67)</td>
<td>56 (65.88)</td>
<td>19 (82.61)</td>
<td>50 (48.54)</td>
<td>34 (77.27)</td>
</tr>
<tr>
<td>Sorter</td>
<td>BubbleSort</td>
<td>18 (64.29)</td>
<td>3 (37.50)</td>
<td>15 (75.00)</td>
<td>0 (0.00)</td>
<td>7 (53.85)</td>
<td>5 (50.00)</td>
</tr>
<tr>
<td>Sorter</td>
<td>HeapBased</td>
<td>33 (52.38)</td>
<td>7 (87.50)</td>
<td>21 (35.00)</td>
<td>0 (0.00)</td>
<td>15 (88.24)</td>
<td>9 (40.91)</td>
</tr>
<tr>
<td>Sorter</td>
<td>ListBased</td>
<td>16 (88.89)</td>
<td>4 (100.00)</td>
<td>8 (80.00)</td>
<td>3 (100.00)</td>
<td>2 (100.00)</td>
<td>6 (100.00)</td>
</tr>
<tr>
<td>Sorter</td>
<td>MinFinding</td>
<td>10 (76.92)</td>
<td>3 (75.00)</td>
<td>8 (80.00)</td>
<td>0 (N/A)</td>
<td>2 (100.00)</td>
<td>4 (100.00)</td>
</tr>
<tr>
<td>Stack</td>
<td>LinkedList</td>
<td>13 (100.00)</td>
<td>0 (N/A)</td>
<td>0 (N/A)</td>
<td>0 (N/A)</td>
<td>6 (100.00)</td>
<td>0 (N/A)</td>
</tr>
<tr>
<td>Vector</td>
<td>ArrayBased</td>
<td>111 (79.86)</td>
<td>73 (82.95)</td>
<td>94 (67.14)</td>
<td>11 (84.62)</td>
<td>71 (71.00)</td>
<td>45 (75.00)</td>
</tr>
<tr>
<td>Vector</td>
<td>LinkedList</td>
<td>141 (90.38)</td>
<td>43 (82.69)</td>
<td>87 (75.65)</td>
<td>11 (73.33)</td>
<td>40 (70.18)</td>
<td>43 (86.00)</td>
</tr>
<tr>
<td>TOTAL</td>
<td></td>
<td>469 (78.56)</td>
<td>176 (74.58)</td>
<td>305 (66.30)</td>
<td>44 (78.57)</td>
<td>201 (65.26)</td>
<td>154 (75.49)</td>
</tr>
</tbody>
</table>
Appendix B

Some Details of the Sulu Language

While sufficiently complete for the purposes of our automated testing research, Sulu is also still a work in progress. In this chapter, we present some of the details of the Sulu programming language that may not directly impact our main subject of automated testing, but was part of the work of this research, and which may be of interest to the reader.

The Sulu programming language counts the Resolve [80] programming language, and Java and JML [60] as its two main influences, and thus Tako [58] is its sibling. Sulu is named after an island in the Philippines, the author’s home country (it is not named after the Star Trek character); it is named so partly because Java is an island in Indonesia, and partly because Sulu was a center for barter trade (swapping) in the early history of the Philippines.

A snapshot of the Sulu source code is available with the electronic thesis and dissertation submission of this document. The latest version of Sulu may be obtained through Sourceforge at: http://sourceforge.net/projects/sulu-lang.
B.1 Inheritance in Sulu

Edwards and his colleagues [38, 41] have pointed out that inheritance in conventional object-oriented languages is one mechanism that is used to supports many different relationships between components. In Sulu we separate some of these different uses of inheritance, by identifying three different inheritance relationships among Sulu modules, as seen in Figure B.1: concepts can inherit from other concepts by extending the behavior of the parent concept; realizations inherit from other realizations and reuse the implementation of the parent realization; and realizations can inherit from concepts by implementing the methods defined by the parent concept.

Sulu concepts are similar to Java interfaces, and realizations are similar to Java classes that implement these interfaces. However, Sulu is different in that every realization has to implement a concept. This separation of the specification of a component’s behavior and its
actual implementation is one key feature we adopted from Resolve. This is fully supported by the Sulu interpreter.

The interpreter also partially supports inheritance between concepts. However, specification inheritance along the lines of Dhara and Leavens [34] is not. The workaround for this is to re-specify the methods in the child concept, taking care to preserve the behavior of the parent.

Inheritance between realizations in Sulu seek to preserve both the reuse relationship and the subtyping (or matching, see Section B.2) relationship by requiring this rule: a realization $R_1$ that implements the concept $C_1$ is only allowed to inherit from realization $R_2$ if $R_2$’s corresponding concept $C_2$ is an ancestor of $C_1$ in its inheritance hierarchy. That is, if $R_1$ inherits from $R_2$, it also implements $R_2$’s concept $C_2$. As of this writing, the Sulu interpreter does not support inheritance between realizations.

### B.2 Supporting Binary Methods

One feature of the Sulu programming language is that the inheritance relationship also defines a matching relationship, instead of the traditional “is a” relationship. Sulu’s support for the matching relationship follows the work of Bruce [19].

Sulu’s use of the matching relationship stems from the need for binary operations. It is often the case where an object needs to operate on other objects of the same type. One class of operators, for example, are comparison operators. Our need for the matching relationship was highlighted when we needed to implement the `SortingMachine` concept.

Imagine we want to build a sorting machine. Properly designed, this sorting machine should be able to sort all kinds of objects that can be compared with each other. So perhaps
we create a concept called `Comparable`, with the idea that this concept should be in the inheritance hierarchy of every realization whose objects can be ordered with a `lessThan` method. Figure B.2 is an example of how the concept could be written.

There are many kinds of comparable objects: integers, strings, and doubles, for example. Using conventional object-oriented design, we might want to build a subtyping hierarchy that looks like Figure B.3, where integers, doubles, and strings inherit from `Comparable`.

Normally, interface inheritance should define a subtyping relationship. For the subtyping relationship to hold, however, input parameters must be contravariant, and output parameters must be covariant. Sulu parameters are always in-out, so they have to be invariant.

That is, to preserve subtyping, each of the children in the inheritance hierarchy must be able to take objects of type `Comparable`, this means that `Integer` objects must be able to accept other types of `Comparable` objects (say, `String` objects) as the parameter to the `lessThan` method. In Java, and many other object-oriented languages, the solution is to use an `instanceof` operator (or equivalent), and throw an exception if the parameter is of the wrong type. Sulu does not have exceptions, but a similar solution may be to have an extra
out parameter that tells you the status of the operation—whether the comparison succeeded or not. While this solution preserves subtyping, it is quite awkward.

A second possibility is to junk the subtyping relationship and allow covariance. This has been the path taken by the Eiffel programming language. However, allowing unrestricted covariance is not type safe.

Using Generics to Break Up the Subtyping Hierarchy

A third possibility is to use generic parameters to break up the monolithic hierarchy into several subtyping hierarchies. That is, we can add a parameter to the Comparable concept that determines what types of objects can be compared. Unfortunately, the resulting solution in Sulu necessitates self-referential parameters.

Figure B.4 shows the Comparable concept that takes in self-referential types. When the Sulu class is created, by for example, saying:
concept Comparable( SelfType: concept Comparable( SelfType ) ) {

    method greaterThan( other: SelfType ): Bool;

    method lessThan( other: SelfType ): Bool;

    method equals( other: SelfType ): Bool;

}

Figure B.4: A Comparable concept with self-referential generic parameters

class Int: Integer( Int ) realization Builtin();

We create an inheritance hierarchy where Integer(Int) inherits only from Comparable(Int) and not, say String(Int). Figure B.5 shows how the self-referential generic parameters break up the inheritance hierarchy.

This was the mechanism employed in earlier version of Sulu, and this kind of self-referential generic parameters are still supported. However, the syntax is very confusing, and the circularity of the types made programs much more difficult to implement and understand.

**Supporting Covariance for Selftypes Only**

Because of the difficulties with self-referential types, Sulu began to support selftypes, in a manner not unlike that of Bruce and his colleagues [19, 21]. It essentially replaces subtyping with a new relationship called matching that allows covariance but only for self types. In Sulu, we’ve adopted a keyword called selftype which when used as the type of the parameter in a method signature means the type of the object the method belongs to. Figure B.6 shows how the comparable concept is defined.
concept Comparable() {
    method greaterThan( other: selftype ): Bool;
    method lessThan( other: selftype ): Bool;
    method equals( other:selftype ): Bool;
}

Figure B.6: A Comparable concept using the selftype keyword

Figure B.5: Using generic parameters breaks up the subtyping hierarchy
Using selftype has these advantages: unlike the invariant parameter approach, using selftype limits binary operations only to objects of the same type; Unlike the covariant parameters approach, using selftype makes it possible to build a sound type system; Unlike the generic parameters approach, the syntax is straightforward and more easily understood.

For a more detailed look at the matching relationship, we refer the reader to the book by Bruce [20], where he shows the soundness of the matching relationship, and argues the utility of matching for binary operations. Matching has been implemented in other programming languages, includeing a dialect of Eiffel [31], and Bruce’s own PolyToil[21].

B.3 Using Nested Maps to Implement the Referencing Environment

Almost every entity in the Sulu runtime environment is a mapping between a name and another entity. For example, the global environment is a mapping from names to concepts, classes, and global variables. A concept, in turn, maps names to realizations, formal generic parameters, and method specifications. Realizations maps names to method bodies, etc. We generalize this into a notion of a nested map.

Recall that a map is an object that associates objects from a set of domain values into a set of range values. A nested map associates every domain value (a textual identifier in this case) with a pair consisting of a range value together with another (possibly empty) nested map representing the inner scope associated with the given target. Figure B.7 illustrates this idea.

In the Sulu interpreter, every entity in its runtime environment is required to extend a base class called Environment. The Environment class defines a mapping from strings to other
environments; it also has a reference to its parent environment—the static scoping enclosing environment. This class contains a `get()` method that, given a string will return another `Environment` object in its local scope; it also has a `find()` method, that given a string will search the map, and all its enclosing parents’ maps for an associated value.

Subclasses of `Environment` are intended to represent specific Sulu entities. They can add additional data members and methods as needed. For example, concepts and realizations have additional data members that store information about their formal generic parameters. Methods store the AST associated with the code of the method body as a data member, and also an additional `call()` method that is responsible for setting up the environment where the procedure is called (an activation record instance), and recursively invoking the interpreter to execute the operation’s code.

Implementing the runtime environment in this manner naturally reflects the hierarchical relationship among nested scopes in a programming language. Conventional lexical scoping
is easy to achieve by searching through the backlinks to the enclosing scopes; fully qualified, or “dotted”, names like `ModuleX.Func5` are also easy to handle. One can use `find()` to locate the map associated with the first name in the dotted sequence, and then drill down through the nested maps one “dot” at a time to find the desired object (although in the design of Sulu, the dot notation is always only one level deep).

Nested maps are also a natural fit for creating the scoped environments of what are traditionally thought of as activation record instances. In the Sulu interpreter, a subclass of the `Environment` class called `Method` acts as an activation record. Objects of type `Method` stores information about its formal parameters, and the AST associated with the method body. It also has a `call()` method, which when invoked, creates a new `Environment` object that acts as an activation record instance (ARI). The `call()` method then binds or associates the parameter values passed into the method call with the formal parameters of the method.

Instantiating a generic component (i.e., creating a Sulu class—recall that a class in Sulu is a fully actualized concrete component) then is simply adapting the idea of binding actual parameters to formal parameters. For every Sulu class, the interpreter creates and `Environment` associated with a realization, and binds the actual types to their associated formal generic parameter name.

The use of nested maps to implement referencing environments have appeared in the literature. Kamin [56] presents a number of simple interpreters where the referencing environment is represented as a record containing a pair of linked lists—a list of names, and a corresponding list of values—together with a pointer to the enclosing scope’s environment. Some values in such a list might themselves contain an environment (e.g., functions). His environment structure can be viewed as a nested map, where the map implementation consists of a pair of lists rather than a hash table or some other data structure. Compiler books typically do
not recommend nested maps for implementing symbol tables. For example Elder [43] notes the space cost of having a symbol table for every nested scope.

While modern implementations of the Map component may allay some of the cost worries, there is some overhead in creating maps for every nested scope, as opposed to using, for example a single hashtable. However, our own motivation for implementing Sulu’s referencing environment is the natural way all the various parts of the runtime (especially generics instantiation) can be represented as nested maps. Thus, using nested maps as the basis for an interpreter’s referencing environment may prove to be useful when ease of implementation and understanding is the primary goal.

**B.4 Sulu Grammar**

In this section we present the grammar of the Sulu programmign language, in the ANTLR format. For brevity, we have eliminated directives used to build the abstract syntax tree, and predicates that simplify the work of the parser generator.

```plaintext
//-----
// TOP LEVEL
//-----

unit  :  ( class_def
    \| concept
    \| realization
    \| include_directive
    \| statement
  )*  
  ;

include_directive: (INCLUDE STRING_LITERAL);
```
//----
//  CONCEPTS, REALIZATIONS, CLASSES
//----

model_prefix: MODEL;

concept : concept_header (extends_clause)? concept_block;

concept_header:
  (model_prefix)? CONCEPT IDENT LPAREN (generic_param_list)? RPAREN;

generic_param_list:
  generic_param (COMMA generic_param)*
  ;

generic_param:
  (IDENT (COLON type_decl))?;

extends_clause:
  EXTENDS IDENT LPAREN (IDENT (COMMA IDENT)*)? RPAREN;

realization_extends_clause:
  EXTENDS (CONCEPT IDENT (LPAREN (IDENT (COMMA IDENT)*)? RPAREN))
            (REALIZATION IDENT (LPAREN (IDENT (COMMA IDENT)*)? RPAREN));

concept_block: LCURLY
  ( initially )?
  ( invariant )?
  ( ( (MODEL)? CLASS) => class_def
    | ( (MODEL)? METHOD) => (proc_header SEMI) )
* RCURLY;

invariant:
  INVARIANT ((expression SEMI) | expression_block);

initially:
  INITIALLY ((expression SEMI) | expression_block);
realization : realization_header
    (realization_extends_clause)?
    implements_clause
    realization_block
    ;

realization_header:
    (model_prefix)? REALIZATION IDENT LPAREN (generic_param_list)? RPAREN;

implements_clause:
    IMPLEMENTS IDENT LPAREN (generic_param_list)? RPAREN;

realization_block: LCURLY
    ( (MODEL)? METHOD => method
    | ( (MODEL)? CLASS) => class_def
    | ( (MODEL)? VAR ) => var_def
    )*
    RCURLY;

class_def: (MODEL)? CLASS IDENT EXTENDS type_decl SEMI;

//-----
// METHODS
//-----

method: proc_header block;

proc_header : ( (MODEL)? METHOD IDENT
    LPAREN proc_formal_parameter_list RPAREN
    (COLON (type))? 
    (precondition)?
    (postcondition)?
    )
    ;

precondition: REQUIRES ( expression | expression_block );
postcondition: ENSURES ( expression | expression_block );

expression_block: LCURLY (expression SEMI) RCURLY;
proc_formal_parameter_list : ( proc_formal_parameter
    (COMMA proc_formal_parameter)*
  )?
  ;

proc_formal_parameter : IDENT COLON (type);

//-----
// TYPE and VAR declarations
//-----
type: (IDENT | SELFTYPE | type_decl);

type_decl: interface_decl (class_decl)?
  ;

interface_decl: (CONCEPT IDENT LPAREN type_list RPAREN);

class_decl: (REALIZATION IDENT LPAREN type_list RPAREN);

type_list: ( type (COMMA type)* )?;

var_def : ( (MODEL)? VAR IDENT COLON (type) SEMI
  )
  ;

//-----
// STATEMENTS
//-----

statement : ( if_statement
  | while_statement
  | procedure_call SEMI
  | swap_statement SEMI
  | assign_statement SEMI
  | consume_statement SEMI
  | no_op SEMI
  | var_def
  )
  ;

block : LCURLY (statement)* RCURLY;
swap_statement : ( IDENT SWAP IDENT );

assign_statement : IDENT ASSIGN expression;

no_op: NOOP;

consume_statement : IDENT CONSUME expression;

old_procedure_call : old_expression DOT IDENT LPAREN proc_param_list RPAREN
                   (DOT IDENT LPAREN proc_param_list RPAREN)*
                   ;

procedure_call : receiver DOT IDENT LPAREN proc_param_list RPAREN
                (DOT IDENT LPAREN proc_param_list RPAREN)*
                ;

receiver : IDENT
          | literals
          ;

proc_param_list : (expression (COMMA expression)*)?
                ;

if_statement : IF_TOK
              LPAREN expression RPAREN block ELSE block
              ;

while_statement : WHILE_TOK
                LPAREN expression RPAREN block
                ;

eexpression : and_or;

and_or : ( not_expression ( (AND|OR) not_expression )* )

not_expression : ( (NOT gt_lt)
                | (TAUT gt_lt)
                | gt_lt
                );
gt_lt: ( plus_minus ( ( GT
| LT
| GTEQ
| LTEQ
| EQ
| NEQ
) )
  plus_minus )*
);

plus_minus: ( times_divide ( (PLUS | MINUS) times_divide )*);

times_divide : ( negate ( (TIMES | DIVIDE | MODULO) negate )* ) ;
negate: ( (MINUS dot_expression))
  | dot_expression;

dot_expression: factor;

factor:
  ( IDENT
  | literals
  | procedure_call
  | (old_expression DOT) => old_procedure_call
  | old_expression
  | LPAREN expression RPAREN
  ) ;

old_expression : (OLD LPAREN expression RPAREN);

literals :
  ( NUMERIC_LITERAL
  | STRING_LITERAL
  | TRUE
  | FALSE
  ) ;

ident_list : (IDENT (COMMA IDENT)*)?;
Bibliography


