Nominated for **the best paper award** at ASE 2003.
Appears in *Proceedings of the 18th IEEE International Conference on Automated Software Engineering (ASE 2003)*
An extended version is invited for submission to a special issue of *Automated Software Engineering Journal.*

Tool-Assisted Unit Test Selection Based on Operational Violations

Tao Xie David Notkin
Department of Computer Science & Engineering, University of Washington
{taoxie, notkin}@cs.washington.edu

Abstract

Unit testing, a common step in software development, presents a challenge. When produced manually, unit test suites are often insufficient to identify defects. The main alternative is to use one of a variety of automatic unit test generation tools: these are able to produce and execute a large number of test inputs that extensively exercise the unit under test. However, without a priori *specifications, developers need to manually verify the outputs of these test executions, which is generally impractical. To reduce this cost, unit test selection techniques may be used to help select a subset of automatically generated test inputs. Then developers can verify their outputs, equip them with test oracles, and put them into the existing test suite. In this paper, we present the* operational violation *approach for unit test selection, a black-box approach without requiring* a priori *specifications. The approach dynamically generates operational abstractions from executions of the existing unit test suite. Any automatically generated tests violating the operational abstractions are identified as candidates for selection. In addition, these operational abstractions can guide test generation tools to produce better tests. To experiment with this approach, we integrated the use of Daikon (a dynamic invariant detection tool) and Jtest (a commercial Java unit testing tool). An experiment is conducted to assess this approach.*

1. Introduction

The "test first" principle, as advocated by the extreme programming development process [2][3], requires unit tests to be constructed and maintained before, during, and after the source code is written. A unit test suite comprises a set of test cases. A test case consists of a test input and a test oracle, which is used to check the correctness of the test result. Developers usually need to manually generate the test cases based on written or, more often, unwritten requirements. In practice, developers tend to write a relatively small number of unit tests, which in turn tend to be useful but insufficient. Some commercial tools for Java unit testing, such as ParaSoft's Jtest [14], attempt to fill the gaps not covered by any manually generated unit tests. These tools can automatically generate a large number of unit test inputs to exercise the program. However, no test oracles are produced for these automatically generated test inputs unless developers do some additional work: in particular, they need to write some formal specifications or runtime assertions [5], which seems to be uncommon in practice. Without *a priori* specifications, manually verifying the outputs of such a large number of test inputs requires intensive labor, which is impractical. Unit test selection is a means to address this problem by selecting the most valuable subset of the automatically generated test inputs. Then developers can inspect the executions of this much smaller set of test inputs to check the correctness and to add oracles.

O*perational violation* is a black-box test selection approach that does not require *a priori* specifications. An operational abstraction describes the actual behavior during program execution of an existing unit test suite [12]. If the execution of an automatically generated test input violates an operational abstraction, this test input is chosen as one of the test selection candidates. The key idea behind this approach is that the violating test exercises a new feature of program behavior that is not covered by the existing test suite. We have implemented this approach by integrating Daikon [8] (a dynamic invariant detection tool) and Jtest [14] (a commercial Java unit testing tool).

The next section presents background information on the unit test selection problem and two techniques that are integrated in our approach: operational abstraction generation and specification-based unit test generation. Section 3 describes the motivating example that is used to illustrate our approach. Section 4 presents our operational violation approach. Section 5 describes the experiment that is conducted to assess the approach. Section 6 discusses related work, and then Section 7 concludes.

2. Background

2.1. Unit test selection

In this work, the objective of unit test selection is to select the most valuable subset of the automatically generated test inputs, allowing a developer both to manually verify their test results and to augment the existing unit tests. There are two closely related goals.

For fault detection, the most valuable test inputs are those that have the highest probability of exposing faults and verifying their test results can improve the probability of detecting faults. For test augmentation, the most valuable test inputs are those that complement the existing tests to together achieve a better testing criterion.

More formally, the objective of unit test selection in this context is to answer the following question as inexpensively as possible:

Given a program unit u*, a set* S *of existing tests for* u*, and a test* t *from a set* S' *of unselected tests for* u*, does the execution of* t *exercise at least one new feature that is not exercised by the execution of any test in* S*?*

If the answer is *yes*, *t* is removed from *S'* and put into *S*. Otherwise, *t* is removed from *S'* and discarded. In this work, the initial set *S* is the existing unit tests, which are usually manually written. The set *S'* of unselected tests is automatically generated tests. The term *feature* is intentionally vague, since it can be defined in different ways. For fault detection, a new feature could be a fault-revealing behavior that does not occur during executions of the existing tests. In white-box test augmentation, a new feature could be a program behavior exhibited by executing a new structural entity, such as statement, branch, or def-use pair. In other words, the white-box test augmentation is based on residual structural coverage [18]. In black-box test augmentation, a feature could be a program behavior exhibited by covering a new predicate in *a priori* specifications [6].

Since manual effort is required to verify the results of selected test inputs, it is important to select a relatively small number of tests. This is different from the problems that traditional test selection techniques address [6][12]. In those problems, there are test oracles for unselected test inputs. Therefore, selecting a relatively large number of tests during selection is usually acceptable for those problems, but is not practical in this work.

2.2. Operational abstraction generation

An *operational abstraction* is a collection of logical statements that abstract the program's runtime behavior [12]. It is syntactically identical to a formal specification. In contrast to a formal specification, which expresses desired behavior, an operational abstraction expresses observed behavior. Daikon [8], a dynamic invariant detection tool, is used to infer operational abstractions from program executions of test suites. Like other dynamic analysis techniques, the quality of the test suite affects the quality of the analysis. Deficient test suites or a subset of sufficient test suites may not help to infer a generalizable program property. Nonetheless, operational abstractions inferred from the executed test suites constitute a summary of the test execution history.

2.3. Specification-based unit test generation

Given a formal specification, specification-based unit test generation tools can automatically generate test inputs for a unit. A specification for a class generally consists of preconditions and postconditions for methods, in addition to class invariants for the class [16]. Preconditions specify conditions that must hold before a method can be executed. Postconditions specify conditions that must hold after a method is completed. Class invariants specify conditions that the objects of the class should always satisfy. They are checked for every non-static, non-private method entry and exit, and for every non-private constructor exit. Class invariants can be treated as preconditions and postconditions for these methods.

Filtering the test input space based on preconditions has been used to effectively automate unit test generation [4]. Automatically generating test inputs to exercise some particular postconditions or assertions has also been attempted [9]. A commercial Java unit testing tool, ParaSoft's Jtest, can automatically generate unit tests for a Java class [14]. When no specifications are provided, Jtest can automatically generate test inputs to perform white-box testing. When specifications are provided with the class, Jtest can make use of them to perform black-box testing. If the code has preconditions, Jtest tries to find inputs that satisfy all of them. If the code has postconditions, Jtest creates test inputs that verify whether the code satisfies these conditions. If the code has class invariants, Jtest creates test inputs that try to make them fail. By default, Jtest tests each method by generating arguments for them and calling them independently. In other words, Jtest basically tries the calling sequences of length one by default. Tool users can set the length of calling sequences in the range of one to three. If a calling sequence of length three is specified, Jtest first tries all calling sequences of length one followed by all those of length two and three sequentially.

3. Motivating Example

As a motivating example, we use a Java implementation of a bounded stack that stores unique elements of integer type. Figure 1 shows the class including the implementations of several methods that we shall refer to in the next section. Stotts et al. coded this Java implementation to experiment their algebraic-specification-based method for systematically creating unit tests [20]; they provide a web link to the full source code and associated test suites. They have defined two unit test suites for this class: a basic JUnit [15] test suite (8 tests), in which one test method is generated for a public method in the target class; and a JAX test suite (16 tests), in which one test method is generated for an axiom

in the abstract data type specification. The basic JUnit test suite does not expose any fault but one of the JAX test cases exposes one fault (handling a pop operation on an empty stack incorrectly). In practice, developers usually fix the faults exposed by existing unit tests before they augment the unit test suite. In this example and for our analysis of our approach, instead of fixing the exposed fault, we remove this fault-revealing test case from the JAX test suite to make all the existing test cases pass.

```
public class uniqueBoundedStack {
  /** @invariant this.max==this.elems.length */
  private int[] elems;
  private int numberOfElements;
  private int max;

  public uniqueBoundedStack() {
    numberOfElements = 0;
    max = 2;
    elems = new int[max];
  }

  /** @pre 0<= this.numberOfElements <= this.max */
  /** @post $result==$pre(int,this.numberOfElements)*/
  public int getNumberOfElements() {
    return numberOfElements;
  }

  public void pop(){
    numberOfElements--;
  }

  public int top(){
    if (numberOfElements < 1) {
      System.out.println("Empty Stack");
      return -1;
    } else {
      return elems[numberOfElements-1];
    }
  }

  public boolean isMember(int k) {
    for(int index=0; index<numberOfElements; index++)
      if (k==elems[index])
        return true;
    return false;
  }

  public void push(int k) {…}
  public boolean isEmpty() {…}
  public boolean isFull() {…}
  public int maxSize() {…}
  public boolean equals(uniqueBoundedStack s) {…}
  public int[] getArray() {…}
};
```

Figure 1. The uniqueBoundedStack program

The code in Figure 1 is annotated with some Design-by-Contract (DbC) comments [13], the approach used in this example to define the class specification. `@invariant` is used to denote class invariants. `@pre` and `@post` are used to denote the preconditions and postconditions, respectively. In the postconditions, the `$pre` keyword is used to refer to the value of an expression immediately before calling the method. The syntax to use it is `$pre(ExpressionType, Expression)`. The `$result` keyword is used to represent the return value of the method.

4. Operational Violation Approach

This section describes the operational violation approach. Section 4.1 explains the basic technique of the approach. Section 4.2 presents the precondition removal technique to complement the basic technique. Section 4.3 describes the iterative process of applying these techniques. Section 4.4 illustrates the rationales behind this approach.

4.1. Basic technique

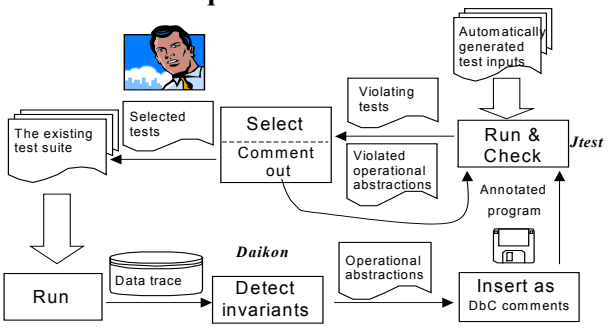

Figure 2. An overview of the basic technique

In the basic technique (Figure 2), operational abstractions are inferred from program executions of the existing unit test suite using Daikon. The Daikon toolset is extended to insert the operational abstractions into the source code as DbC comments. The resulting code is fed to Jtest, which automatically generates and executes tests. The two symptoms of an operational violation are that an operational abstraction is evaluated to be false, or that an exception is thrown while evaluating an operational abstraction. When a certain number of operational violations have occurred before Jtest exhausts its testing repository, it stops generating test inputs and reports operational violations. All the reported operational violations, including the violating test inputs, are exported to a text file. Given the exported text file, a Perl script was developed to automatically comment out the violated operational abstractions in the source code. At the same time, the operational violations are collected. Then Jtest is invoked again, using a script, given the program with reduced operational abstractions. The preceding procedure is repeated automatically until no operational violations are found.

Given the collected operational violations, a Perl script was developed to select the first encountered test for each violated operational abstraction. Then the selected violating tests are sorted based on the number of their violated operational abstractions. The tests that violate more operational abstractions are put before those that violate fewer ones. The script produces a JUnit [15] test class, which contains the sorted list of violating test inputs as well as their violated operational abstractions.

An integration tool was developed to fully automate the preceding steps, including invoking Daikon and Jtest, and postprocessing the text file. After running the integration tool, developers can then inspect the resulting sorted tests to verify the correctness of their executions. Optionally, developers can add assertions for the test inputs as test oracles for regression testing.

One example of an operational violation is shown in Figure 3. The example, which indicates a deficiency of the JAX test suite, shows method `isMember`'s two violated postconditions followed by the violating test. This violating test should be added to the existing test suite.

```
isMember:
@post: [($pre(int , k) == 3) == ($result == true)]
@post: [($pre(int , k) == 3) == (this.numberOfElements == 1)]
For input:
 uniqueBoundedStack THIS = new uniqueBoundedStack ();
 boolean RETVAL = THIS.isMember (3);
```

Figure 3. An example of operational violations using the basic technique

4.2. Precondition removal technique

In the basic technique, when the existing test suite is deficient, the inferred preconditions might be so restrictive that Jtest filters out those legal test inputs during test generation and execution. However, we need to exercise the unit under more circumstances than what is constrained by the inferred preconditions. To do this, before the annotated code is fed to Jtest, we remove all precondition comments and we thus exercise the unit under a broader variety of test inputs. Indeed, removing preconditions can make test generation tools less guided, especially those tools that generate tests mainly based on preconditions [4]. The basic technique and this precondition removal technique are used together to complement each other.

Figure 4 shows two examples of operational violations and the use of this technique. The example in the upper part indicates a deficiency of the basic JUnit test suite, and the violating test exposes the fault detected by the original JAX test suite. The example in the lower part shows a deficiency of the JAX test suite that exposes another new fault, one not reported in the original experiment [20]. If this stack implementation can accommodate negative integer elements, this operational violation shows that using –1 as an exception indicator makes the *top* method work incorrectly when the integer –1 is put on top of the stack. This is a typical value-sensitive fault and even a full-path-coverage test suite cannot guarantee to expose this fault. This violation is not reported by the basic technique, since there are several inferred preconditions for method *top* to prevent the element –1 from being on top of the stack, such as

```
{ for (int i = 0 ; i <= this.elems.length-1; i++)
                $assert ((this.elems[i] >= 0));   }
```

where *$assert* is used to denote an assertion statement.

```
pop:
@post: [( this.elems[this.numberOfElements] ==
       this.elems[$pre(int, this.numberOfElements)-1] )]
@post: [this.numberOfElements == 0 ||
                this.numberOfElements == 1]
@invariant: [this.numberOfElements == 0 ||
                this.numberOfElements == 1 ||
                this.numberOfElements == 2]
For input:
 uniqueBoundedStack THIS = new uniqueBoundedStack ();
 THIS.pop ();
```

```
top:
@post: [($result == -1) == (this.numberOfElements == 0)]
For input:
 uniqueBoundedStack THIS = new uniqueBoundedStack ();
 THIS.push (-1);
 int RETVAL = THIS.top ();
```

Figure 4. Examples of operational violations using the precondition removal technique

4.3. Iterations

After the test selections are performed using the techniques in Section 4.1 and 4.2, all the violating tests can be further run together with the existing ones to infer the refined operational abstractions. The process described in Sections 4.1 and 4.2 is repeated until there are no operational violations reported for the operational abstractions generated from the previous iteration (or until the user-specified maximum number of iterations has been reached).

Figure 5 shows operational violations during the first and second iterations on the JAX test suite. After the first iteration, a violating test is added to the existing test suite to weaken the "==" predicate to the "`$implies`" predicate. After the second iteration, another violating test further removes this "`$implies`" predicate since it is inferred owing only to the deficiency of the tests.

```
 (1st iteration)
 isMember:
 @post: [($result == true) == (this.numberOfElements == 1)]
For input:
 uniqueBoundedStack THIS = new uniqueBoundedStack ();
 THIS.top ();
 THIS.push (2);
 boolean RETVAL = THIS.isMember (1);

 (2nd iteration)
  isMember:
   @post:[($result == true) $implies (this.numberOfElements == 1)]
For input:
 uniqueBoundedStack THIS = new uniqueBoundedStack ();
 THIS.push (2);
 THIS.push (0);
 boolean RETVAL = THIS.isMember (0);
```

Figure 5. Operational violations during iterations

4.4. Why it works

The operational abstractions might not be consistent with the oracle specifications, which are the actual specifications for the unit. Assume that OA_PRE and OS_PRE are the domains constrained by the preconditions of the operational abstractions and the oracle specifications respectively. Legal domains are the ones that satisfy the preconditions of the oracle specifications, and illegal domains are the ones that do not satisfy these preconditions. The following are analyses of different potential relationships between OA_PRE and OS_PRE:

1. If (OA_PRE $\cap$ OS_PRE) $\subset$ OS_PRE, then no test inputs can be generated in the legal domain of (OS_PRE – OA_PRE);
2. If (OA_PRE – OS_PRE) $\neq \emptyset$, then test inputs can be generated in the illegal domain of (OA_PRE – OS_PRE);
3. If OA_PRE=OS_PRE, then traditional specification-based test generations are performed.

The precondition removal technique addresses the first case by changing the situation to the second case. In order to test the robustness of a unit, it is useful to generate illegal test inputs to exercise it. If an illegal test input causes any operational violation, then the tool reports it, which catches the attentions of developers.

If the execution of a legal test input generates a violation of a class invariant or postcondition, there are two possible causes. The first cause could be that the class invariant or postcondition in the operational abstractions is more restrictive than the one in the oracle specifications. Operational violations indicate that the violating test exercises a new feature of the program, which is not covered by the existing test suite. It is desirable to select this violating test to augment the existing test suite. The second cause could be that there is a fault revealed by the violating test. Running the existing test suite on the code exhibits the normal behavior reflected by the operational abstractions, whereas the violating test makes the code exhibit the abnormal behavior.

5. Experiment

This section presents the experiment that assesses our approach. First, the measurements in the experiment are described. Then, the experiment instrumentation is presented. Finally, the experimental results and threats to validity are described.

The general questions we wish to answer include:

1. Is the number of automatically generated tests large enough for developers to adopt unit test selection techniques?
2. Is the number of tests selected by our approach small enough for developers to inspect affordably?
3. Do the selected tests by our approach have a high probability of exposing faults?
4. How does the operational violation approach compare with the residual branch coverage approach [18]?

5.1. Measurements

We cannot answer all of these questions with care, so we designed an experiment to give an initial sense of the general questions of efficacy of this approach. In particular, we performed the following measurements to address these questions directly or indirectly:

- **Automatically generated test count in the absence of any operational abstraction (#AutoT)**: We measured the number of tests automatically generated by Jtest in the absence of any operational abstraction. This measurement is related to the first question.
- **Selected test count (#SelT):** We measured the number of the tests selected by a test selection technique. This measurement is related to the second question.
- **Fault-revealing selected test count (#FRT):** We measured the number of fault-revealing tests among the selected tests. We manually inspect the selected tests and the source code to determine the fault-revealing tests. Note that multiple fault-revealing tests might expose the same fault in different ways. This measurement is related to the third question.

The first measurement is performed for each subject program. The second and third measurements are performed for each combination of the basic/precondition removal techniques, subject programs, and number of iterations. To help answer the fourth question, the second and third measurements are also performed for each subject program using the residual branch coverage approach. The residual branch coverage approach selects an automatically generated test, only if the test exercises a new branch, which is not covered by any existing test or any previously selected test.

5.2. Experiment instrumentation

5.2.1. Subject programs. Table 1 lists the subject programs that we used in the experiment. Each subject program is a Java class equipped with a manually written unit test suite. The first column shows the names of the subject programs. The second and third columns show the number of public methods, and the number of lines of

executable code for each program respectively. The fourth column shows the number of test cases in the test suite of each program. The last two columns present some measurement results that we shall describe in Section 5.3.

Table 1. Subject programs used in the experiment

Program	#Public Method	#LOC	# Tests	#AutoT	#ExT
UB-Stack (JUnit)	11	47	8	96	1
UB-Stack (JAX)	11	47	15	96	1
RatPoly-1	13	161	24	223	1
RatPoly-2	13	191	24	227	1
RatPolyStack-1	13	48	11	128	4
RatPolyStack-2	12	40	11	90	3
BinaryHeap	10	31	_	166	2
BinarySearchTree	16	50	_	147	0
DisjSets	4	11	_	24	4
QueueAr	7	27	_	120	1
StackAr	8	20	_	133	1
StackLi	9	21	_	99	0

Among these subjects, `UB-Stack(JUnit)` and `UB-Stack(JAX)` are the motivating example (Section 3) with the basic JUnit test suite and the JAX test suite (with one failing test removed), respectively [20]. `RatPoly-1/RatPoly-2` and `RatPolyStack-1/RatPolyStack-2` are the student solutions to two assignments in a programming course at MIT. These selected solutions passed all the unit tests provided by instructors. The rest of the subjects come from a data structures textbook [23]. Daikon group members developed unit tests for 10 data structure classes in the textbook. Most of these unit tests use random input generation to fully exercise the programs. We applied our approach on these classes, and five classes – the last five at the end of Table 1 – have at least one operational violation. Since the test suite for these classes are not organized as a set of test cases, the fourth column does not apply.

5.2.2. Tools. Daikon and Jtest are used in the experiment to implement our approach. We developed a set of Perl scripts to integrate these two tools. In Jtest's configuration for the experiment, we set the length of calling sequence as two. We used Daikon's default configuration for the generation of operational abstractions. Based on the Hansel tool [10], we developed a test selection tool based on residual branch coverage.

In particular, we first run Jtest on the subject programs to collect the `#AutoT` measurement in the absence of any operational abstraction. Then for each subject program, we performed the experiment using the basic technique and repeated it until the third iteration was reached or until no operational violations were reported for the operational abstractions generated from the previous iteration. A similar procedure was performed on the precondition removal technique. At the end of each iteration, the `#SelT` and `#FRT` measurements were collected. Finally, we used the tool based on residual branch coverage to collect the `#SelT` and `#FRT` measurements on the tests automatically generated by Jtest in the absence of any operational abstraction.

5.3. Experimental results

The fifth column of Table 1 shows the `#AutoT` results. From the results, we observed that except for the especially small `DisjSets` program, nearly 100 or more tests are automatically generated. We also tried setting the length of the calling sequence to three, which caused Jtest to produce thousands of tests for the programs. This shows that test selection techniques are needed since it is not practical to manually check the outputs of all these automatically generated tests.

The last column of Table 1 shows the number of the automatically generated tests that cause uncaught runtime exceptions. These tests should also be selected along with those tests selected by the operational violation approach or the residual branch coverage approach.

Table 2 shows the number of selected tests (`#SelT`) and fault-revealing selected tests (`#FRT`). The data in the "`With Preconds`" columns are for the basic technique (with preconditions). The data in the "`W/O Preconds`" columns are for the precondition removal technique (without preconditions). For those data with the value of zero, their entries are left blank. The bottom row of Table 2 shows the median percentage of `#FRT` among `#SelT`. In the calculation of the median percentage, entries with a `#SelT` value of zero are not included.

The numbers of selected tests vary across different programs but on average their numbers are not large, so their outputs could be verified with affordable human effort. We observed that, in this experiment, the selected tests have a high probability of exposing fault. These fault-revealing tests are usually inputs that cause the program to exhibit abnormal behavior, such as illegal arguments or special object states. As one concrete example, the only selected test for the `RatPoly-1` program using the precondition removal technique makes the program infinitely loop until a Java out-of-memory error occurs. This test is not in the set of automatically generated tests by Jtest in the absence of any operational abstraction. This result is suggestive that our approach may have value. In addition, we observed that although those non-fault-revealing tests do not expose any fault, most of them represent some special class of inputs and thus may be valuable if selected for regression testing.

Table 2. The numbers of selected tests and fault-revealing selected tests using the basic technique and precondition removal technique for each program and each iteration

Programs	Iteration 1				Iteration 2				Iteration 3			
	With Preconds		W/O Preconds		With Preconds		W/O Preconds		With Preconds		W/O Preconds	
	#SelT	#FRT	#SelT	#FRT	#SelT	#FRT	#SelT	#FRT	#SelT	#FRT	#SelT	#FRT
UBS (JUnit)	1		15	5	2		6	1			1	
UBS (JAX)	3		25	9			4					
RatPoly-1	2	2	1	1								
RatPoly-2	1	1	1	1	1	1						
RatPolyStack-1			12	8			5	2			1	
RatPolyStack-2	1		10	7			2					
BinaryHeap	3	2	8	6	1		8	6			6	
BinarySearchTree			3	3								
DisjSets	1	1	2	2								
QueueAr	6	1	11	1			4	1				
StackAr	5	1	9	1	1		1					
StackLi			2									
(Median of #FRT/ #SelT)	20%		68%		0%		17%		_		0%	

Table 3. The numbers of selected tests and fault-revealing selected tests using the residual branch coverage approach

Program	#Total-Branch	#BR-Branch	#AR-Branch	#SelT	#FRT
UB-Stack (JUnit)	41	13	5	5	1
UB-Stack (JAX)	41	1	1	0	0
RatPoly-1	125	3	3	0	0
RatPoly-2	139	9	9	0	0
RatPolyStack-1	22	7	6	1	0
RatPolyStack-2	16	0	0	0	0
BinaryHeap	34	2	0	1	0
BinarySearchTree	56	7	7	0	0
DisjSets	10	0	0	0	0
QueueAr	21	2	0	2	0
StackAr	20	1	0	1	0
StackLi	21	6	5	1	0
(Median of #FRT/ #SelT)	--			0%	

We observed that a violating test generated by Jtest in our approach is often not in the set of automatically generated tests by Jtest in the absence of any operational abstraction. This suggests that operational abstractions can effectively guide Jtest to generate tests to violate them.

Based on the median percentage of `#FRT` among `#SelT`, the precondition removal technique is overall more effective than the basic technique. By inspecting the violating tests, we found that sometimes the basic technique does guide Jtest to generate some new violating test inputs that are not generated in the precondition removal technique. We observed, in this experiment, that a couple of iterations are good enough in our approach. Jtest's test generation and execution time dominates the running time of applying our approach. Most subjects took several minutes, but the `BinaryHeap` and `RatPolyStack` programs took on the order of 10 to 20 minutes. We expect that the execution time can be optimized if future versions of Jtest can better support the resumption of test generation and execution after the violated operational abstractions are commented out.

The last two columns of Table 3 shows the `#SelT` and `#FRT` measurements in the residual branch coverage approach. The second column of Table 3 shows the count of the total branches for each subject. The third column presents the count of residual branches after the execution of the existing tests. The fourth column presents the count of residual branches after the execution of both the existing tests and the selected tests. The bottom row of Table 3 shows the median percentage of `#FRT` among `#SelT`.

We observed that the existing tests have already left no residual branches on two of the subjects. The tests automatically generated by Jtest can further reduce the count of residual branches on half of the subjects. The number of the selected tests or fault-revealing tests in the residual coverage approach is fewer than in the operational violation approach. We further measured the residual branch coverage after the execution of both the existing tests and the tests selected by the operational violation approach. The count of residual branches is usually larger than the one in the residual branch coverage approach. This indicates that the residual branch coverage approach is more effective in selecting tests to

achieve better branch coverage. On the other hand, although the programs contain faults that are exposed by the tests selected by the operational violation approach, the tests selected by the residual branch coverage approach cannot expose most of them. This suggests that combining the residual branch coverage approach and the operational violation approach may provide a better solution for unit test selection.

5.4. Threats to validity

The threats to validity primarily include the degree to which the subject programs, faults, and test cases are representative of true practice. The subjects in the experiment are small, although the faults in them are real, some of which were not detected before. These threats could be reduced by more experiments on wider types of subjects. The threats to validity also include instrumentation effects that can bias our results. Faults in our Perl scripts, Daikon, or Jtest might cause such effects. To reduce these threats, we manually inspected the results for each program subject.

6. Related Work

Harder et al. present a specification-based technique without requiring *a priori* specification [12]. Their operational difference technique starts with an operational abstraction generated by an existing test suite. Then it generates a new operational abstraction from the test suite augmented by a candidate test case. If the new operational abstraction differs from the previous one, it adds the candidate test case to the suite. This process is repeated until some number *n* of candidate cases have been consecutively considered and rejected. Both operational difference and our approach use the operational abstractions generated from test executions. Our approach exploits operational abstractions' guidance to test generation, whereas operational difference operates on a fixed set of given tests.

The DIDUCE tool can continuously check a program's behavior against the incrementally inferred invariants during the run(s), and produce a report of all invariant violations detected along the way [11]. This can help detect bugs and track down the root causes. A usage model of DIDUCE is proposed, which is similar to the unit test selection problem in this work. Both DIDUCE and our approach make use of violations of the inferred invariants. The inferred invariants used by our approach are produced by Daikon at method entry and exit points, whereas DIDUCE infers a limited set of simpler invariants from procedure call sites and object/static variable read/write sites. Also DIDUCE does not investigate the effects of operational abstractions on test generation.

Failed static verification attempts are used to indicate the deficiencies in the unit tests [17]. The unverifiable invariants indicate unintended properties and developers can get hints on how to improve the tests. Our approach reports not only the violated invariants but also the executable counterexamples for them. In addition, the over-restrictiveness of preconditions makes static verification of inferred invariants less effective. Even if a static verifier could confirm an inferred postcondition given some over-restrictive preconditions, it is hard to tell whether it is generalizable to the actual preconditions. In our approach, the precondition removal technique tackles this problem.

When specifications are provided for a unit *a priori*, specification coverage criteria are used to suggest a candidate set of test cases that exercise new aspects of the specification [6]. Like the preceding related work based on operational abstractions, our approach does not require a specification *a priori*.

In white-box testing (such as the residual structural coverage [18]), developers can select and inspect the tests that provide new structural coverage unachieved by the existing test suite. Test case prioritization techniques, such as additional structural coverage techniques, can produce a list of sorted tests for regression testing [19][21]. Clustering and sampling the execution profiles can also be used to select a list of tests for inspection and selection [7]. Although in this work, we only integrated Daikon and Jtest to implement our approach, some other specification-based unit test generation tools can also be used to implement the approach [4][9]. Other kinds of operational abstraction generation, such as sequencing constraints or protocol inferences [1][22], can be used in this approach as well. In future work, we plan to experiment other implementations of the approach.

7. Conclusion

Selecting automatically generated tests to check correctness and augment the existing unit test suite is an important step in unit testing. Inferred operational abstractions act as a summary of the existing test execution history. A new test that violates an operational abstraction is a good candidate for inspection and selection, since it exercises a new program feature that is not covered by the existing tests. The violating test also has a high probability of exposing faults in the code if there are any. In addition, operational abstractions can guide test generation tools to produce better test inputs.

Instead of considering the test augmentation as a one-time phase, it should be considered as a frequent activity in software evolution, perhaps as frequent as regression unit testing. When a program is changed, the operational abstractions generated from the same unit test suite might change as well, presenting opportunities for possible

operational violations. Tool-assisted unit test augmentation may be a practical means of evolving unit tests and assuring better unit quality.

The unit test selection tool based on operational violations is available for download from

http://www.cs.washington.edu/homes/taoxie/jov/.

8. Acknowledgment

We thank Michael Ernst and the Daikon project members at MIT for their assistance in our use of the Daikon tool and preparation of the experimental subjects. We also thank ParaSoft Inc. for their sponsorship on the Jtest tool. This work was supported in part by the National Science Foundation under grant ITR 0086003. We acknowledge support through the High Dependability Computing Program from NASA Ames cooperative agreement NCC-2-1298.

9. References

[1] G. Ammons, R. Bodik, and J. Larus, "Mining Specifications", *Proceedings of Principles of Programming Languages* (POPL), Portland, Oregon, January 2002, pp. 16-18.

[2] K. Beck, *Extreme programming explained*, Addison-Wesley, 2000.

[3] Kent Beck and Erich Gamma, "Test infected: Programmers love writing tests", *Java Report,* 3(7), July 1998.

[4] C. Boyapati, S. Khurshid, and D. Marinov, "Korat: Automated testing based on Java predicates", *Proceedings of the 2002 International Symposium on Software Testing and Analysis* (ISSTA), Rome, Italy, July 2002, pp.123-133.

[5] Y. Cheon and G. T. Leavens, "A simple and practical approach to unit testing: The JML and JUnit way", *Proceedings of 16th European Conference Object-Oriented Programming* (ECOOP), 2002, pp. 231-255.

[6] J. Chang and D. J. Richardson, "Structural Specification-Based Testing: Automated Support and Experimental Evaluation", *Proceedings of the 7th European Software Engineering Conference / 7th ACM Sigsoft Symposium on the Foundations of Software Engineering* (ESEC/FSE), Sept. 1999, pp. 285-302.

[7] W. Dickinson, D. Leon, and A. Podgurski, "Finding Failures by Cluster Analysis of Execution Profiles", *Proceedings of the International Conference on Software Engineering* (ICSE), 2001, pp 339-348.

[8] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin, "Dynamically discovering likely program invariants to support program evolution", *IEEE Transactions on Software Engineering*, vol. 27, no. 2, Feb. 2001, pp. 1-25.

[9] N. Gupta, "Generating Test Data for Dynamically Discovering Likely Program Invariants", *Proceedings of ICSE 2003 Workshop on Dynamic Analysis* (WODA), May 2003, pp. 21-24.

[10] Hansel 1.0, http://hansel.sourceforge.net/.

[11] S. Hangal and M. S. Lam, "Tracking down software bugs using automatic anomaly detection", *Proceedings of the International Conference on Software Engineering* (ICSE), May 2002, pp. 291-301.

[12] M. Harder, J. Mellen, and M. D. Ernst, "Improving test suites via operational abstraction", *Proceedings of the International Conference on Software Engineering* (ICSE), Portland, Oregon, May 2003, pp. 60-71.

[13] Parasoft Corporation, *Jcontract manuals version 1.5*, http://www.parasoft.com/, October 9, 2002.

[14] Parasoft Corporation, *Jtest manuals version 4.5*, http://www.parasoft.com/, October 23, 2002.

[15] JUnit, http://www.junit.org.

[16] B. Meyer, *Object-Oriented Software Construction*, New York, London, Prentice Hall, Second Edition, 1997.

[17] J. W. Nimmer and M. D. Ernst, "Static verification of dynamically detected program invariants: Integrating Daikon and ESC/Java", *Proceedings of First Workshop on Runtime Verification* (RV), Paris, France, July 23, 2001.

[18] C. Pavlopoulou and M. Young. "Residual test coverage monitoring", *Proceedings of International Conference of Software Engineering* (ICSE)*, 1999*, pp. 277-284.

[19] G. Rothermel, R. Untch, C. Chu, and M. J. Harrold, "Test Case Prioritization", *IEEE Transactions on Software Engineering*, vol. 27, no. 10, October 2001, pp. 929-948.

[20] P. D. Stotts, M. Lindsey, and A. Antley. "An informal formal method for systematic JUnit test case generation", *Proceedings of 2nd XP Universe and 1st Agile Universe Conference* (XP/Agile Universe), 2002, pp 131-143.

[21] A. Srivastava and J. Thiagarajan, "Effectively prioritizing tests in development environment", *Proceedings of International Symposium on Software Testing and Analysis* (ISSTA), Rome, Italy, July 2002, pp. 97-106.

[22] J. Whaley, M. C. Martin and M. S. Lam, "Automatic Extraction of Object-Oriented Component Interfaces", *Proceedings of the International Symposium on Software Testing and Analysis* (ISSTA), July 2002, pp. 218-228.

[23] M. A. Weiss, *Data Structures and Algorithm Analysis in Java*. Addison Wesley Longman, 1999.