Tool-Assisted Unit-Test Generation and Selection Based on Operational Abstractions[†]

Tao Xie[1] and David Notkin[2]
(xie@csc.ncsu.edu,notkin@cs.washington.edu)
[1] *Department of Computer Science, North Carolina State University, Raleigh, NC 27695*
[2] *Department of Computer Science and Engineering, University of Washington, Seattle, WA 98105*

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, programmers 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 programmers 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 generation and selection, a black-box approach without requiring *a priori* specifications. The approach dynamically generates operational abstractions from executions of the existing unit test suite. These operational abstractions guide test generation tools to generate tests to violate them. The approach selects those generated tests violating operational abstractions for inspection. These selected tests exercise some new behavior that has not been exercised by the existing tests. We implemented this approach by integrating the use of Daikon (a dynamic invariant detection tool) and Parasoft Jtest (a commercial Java unit testing tool), and conducted several experiments to assess the approach.

1. Introduction

The *test first principle*, as advocated by the extreme programming development process [2], 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. Programmers usually need to manually generate the test cases based on written or, more often, unwritten requirements. In practice, programmers tend to write a relatively small number of unit tests, which in turn

[†] This article is an invited submission to the Automated Software Engineering Journal. An earlier version of this article appeared in Proceedings of the 16th IEEE Conference on Automated Software Engineering (ASE 2003).

tend to be useful but insufficient for high software quality assurance. Some commercial tools for Java unit testing, such as Parasoft Jtest [32], 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 programmers do some additional work: in particular, they need to write some formal specifications or runtime assertions [7], which seems to be uncommon in practice. Without this additional work, the tools only check the program's robustness: checking whether any uncaught exception is thrown during test executions [25, 8]. 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 programmers can inspect the executions of this much smaller set of test inputs to check the correctness or robustness, and to add oracles.

When *a priori* specifications are not provided, test generation tools usually perform white-box test generation. They try to generate tests to increase structural coverage of the program. In white-box test generation, the tools cannot get the extra guidance provided by *a priori* specifications. For example, the preconditions in specifications can guide test generation tools to generate only those inputs that satisfy the preconditions, which are valid test inputs [32, 4]. The postconditions in specifications can guide test generation tools to generate test inputs to violate the postconditions, which are fault-exposing test inputs [32, 24, 17].

Operational violation is a black-box test generation and 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 [21]. We use the generated operational abstractions to guide test generation tools, so that the tools can more effectively generate test inputs to violate these operational abstractions. If the execution of an automatically generated test input violates an operational abstraction, we select this test input for inspection. 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 [12] (a dynamic invariant detection tool) and Parasoft Jtest [32] (a commercial Java unit testing tool).

The next section describes the example that we use to illustrate our approach. Section 3 presents background information on the unit-test generation and selection problems, and two methods that we integrate.

Section 4 presents the operational violation approach. Section 5 discusses the issues in the approach. Section 6 describes the experiments that we conducted to assess the approach. Section 7 discusses related work, and then Section 8 concludes.

2. Example

This section presents an example to illustrate how programmers can use our approach to test their programs. The example is a Java implementation `uniqueBoundedStack` of a bounded stack that stores unique elements of integer type. Figure 1 shows the class including several method implementations that we shall refer to in the rest of the paper. Stotts et al. coded this Java implementation to experiment with their algebraic-specification-based approach for systematically creating unit tests [37]; they provided a web link to the full source code and associated test suites. Stotts et al. also specified formal algebraic specifications for a bounded stack, whose behavior has a considerable overlap with `uniqueBoundedStack`. The specifications are described in the appendix. The only difference between the specified bounded stack and `uniqueBoundedStack` is the way of handling the uniqueness of the elements stored in a stack.

In the class implementation, the array `elems` contains the elements of the stack, and `numberOfElements` is the number of the elements and the index of the first free location in the stack. The `max` is the capacity of the stack. The `push` method first checks whether the element to be pushed exists in the stack. If the same element already exists in the stack, the method moves the element to the stack top. Otherwise, the method increases `numberOfElements` after writing the element if `numberOfElements` does not exceed the stack capacity `max`. If the stack capacity is exceeded, the method prints an error message and makes no changes on the stack. The `pop` method simply decreases `numberOfElements`. The `top` method returns the element in the array with the index of `numberOfElements-1` if `numberOfElements >= 0`. Otherwise, the method prints an error message and returns `-1` as an error indicator. The `getSize` method returns `numberOfElements`. Given an element, the `isMember` method returns `true` if it finds the same element in the subarray of `elems` up to `numberOfElements`, and returns `false` otherwise.

Stotts et al. have created two unit test suites for this class: a basic JUnit [23] 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 algebraic

```
public class uniqueBoundedStack {
  private int[] elems;
  private int numberOfElements;
  private int max;
  public uniqueBoundedStack() {
    numberOfElements = 0;
    max = 2;
    elems = new int[max];
  }
  public void push(int k) {
    int index;
    boolean alreadyMember;
    alreadyMember = false;
    for(index=0; index<numberOfElements; index++) {
       if(k==elems[index]) {
          alreadyMember = true;
          break;
       }
    }
    if (alreadyMember) {
       for (int j=index; j<numberOfElements-1; j++)
           elems[j] = elems[j+1];
       elems[numberOfElements-1] = k;
    } else {
       if (numberOfElements < max) {
          elems[numberOfElements] = k;
          numberOfElements++;
          return;
       } else {
          System.out.println("Stack full, cannot push");
          return;
       }
    }
  }
  public void pop(){
    numberOfElements--;
  }
  public int top(){
    if (numberOfElements < 1) {
      System.out.println("Empty Stack");
      return -1;
    } else {
      return elems[numberOfElements-1];
    }
  }
  public int getSize() {
    return numberOfElements;
  }
  public boolean isMember(int k) {
    for(int index=0; index<numberOfElements; index++)
      if (k==elems[index])
        return true;
    return false;
  }
  ...
}
```

Figure 1. The `uniqueBoundedStack` program

specifications (shown in the appendix). 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, programmers usually fix the faults exposed by the 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.

3. Background

We next describe the unit-test generation and selection problems that we address, and the operational abstraction generation and specification-based test generation that we integrate.

3.1. Unit-test generation and selection

In this work, the objective of *unit-test generation* is to automatically generate tests to augment the existing tests for a program unit. This activity is also called *unit-test augmentation*. More precisely, we want to generate tests to exercise a program unit's new behavior that is not exercised by the existing tests. More formally, the unit-test generation problem is described as follows.

PROBLEM 1. *Given a program unit u, a set S of the existing tests for u, generate new tests that exercise new features not being exercised by the execution of any test in S.*

The term *feature* is intentionally vague, since it can be defined in different ways. For fault detection, a new feature could be fault-revealing behavior that does not occur during the execution of the existing tests. In black-box test generation, a new feature could be a predicate in the specifications for the unit [6]. In white-box test generation, a new feature could be program behavior exhibited by executing a new structural entity, such as statement, branch, or def-use pair.

In practice, automatic unit-test generation tools often generate a large number of tests, and not all of these tests exercise new features. The objective of *unit-test selection* is to select the most valuable subset of the automatically generated test inputs, allowing a programmer 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 robustness testing [25, 8], the most valuable test inputs are those that have the highest probability of exposing failures. If the test inputs actually expose failures, adding preconditions or input-checking code against these test inputs can improve the robustness of the code. For black-box or white-box 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:

PROBLEM 2. *Given a program unit u, a set S of the 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. *Residual structural coverage* characterizes the structural entities that have not been covered by existing tests [33]. In white-box test augmentation, we select a test that can decrease residual structural coverage and thus increase structural coverage. In black-box test augmentation, we select a test that covers 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, 21]. 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.

3.2. Operational abstraction generation

An *operational abstraction* is a collection of logical statements that abstract the program's runtime behavior [21]. 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 [12], a dynamic invariant detection tool, can be used to infer operational abstractions (also called invariants) from program executions of test suites. These operational abstractions are in the form of DbC annotations [27, 26, 31]. Daikon examines variable values computed during executions and generalizes over these values to obtain operational abstractions. In particular, Daikon predefines a set of grammars for common program properties and conjectures all possible properties in the grammars for the variables at method entries and exits. Then Daikon examines the execution data and discards any properties that are invalidated. After processing all the execution data, for each detected invariant, Daikon computes the probability that the invariant would appear by chance in a random set of samples. The invariant is reported only if its probability is smaller than a user-defined confidence parameter. This filtering mechanism used by Daikon is called *statistical tests*. 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. In other words, the executions of the test suites all satisfy the properties in the generated operational abstractions.

3.3. Specification-based unit-test generation

Given a formal specification, specification-based unit-test generation tools can automatically generate test inputs for a unit. In this work, we focus on a class's specification that consists of preconditions and postconditions for the methods in the class, in addition to class invariants for the class [27, 26, 31]. 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.

Some testing tools, such as Korat [4] and AsmLT [13, 16], filter the test input space based on preconditions to effectively automate unit-test generation. Both Korel et al. [24] and Gupta [17] reduce test generation for violating postconditions or assertions to the problem of test generation for exercising a particular statement or branch. A commercial Java unit testing tool, Parasoft Jtest 4.5 [32], can automatically generate unit tests for a Java class. 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. The provided preconditions, postconditions, or class invariants give extra guidance to Jtest in its test generation. If the code has preconditions, Jtest tries to generate test inputs that satisfy all of them. If the code has postconditions, Jtest generates test inputs that verify whether the code satisfies

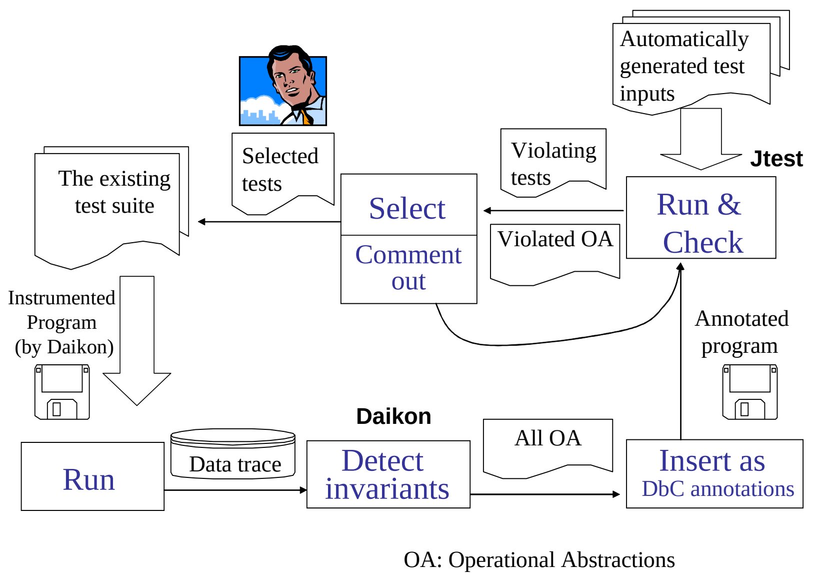

Figure 2. An overview of the basic technique

these conditions. If the code has class invariants, Jtest generates 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.

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.

4.1. Basic technique

In the basic technique (Figure 2), we run the existing unit test suite on the program that is instrumented by the Daikon frontend. The execution produces a data trace file, which contains variable values computed during execution. Then we use Daikon to infer operational abstractions from the data trace file. We extend the Daikon toolset to insert the operational abstractions into the source code as DbC annotations. We feed the resulting annotated code to Jtest, which automatically generates and executes new 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, Jtest stops generating test inputs and reports operational violations. Jtest exports all the operational violations, including the violating test inputs, to a text file. Given the exported text file, we automatically comment out the violated operational abstractions in the source code. At the same time, we collect the operational violations. Then we invoke Jtest again, which is given the program with reduced operational abstractions. We repeat the preceding procedure iteratively until we cannot find any operational violations. We call these iterations as *inner iterations* to avoid their being confused with the iterations described in Section 4.3. The inner iterations mainly comprise the activities of Jtest's test generation and execution, Jtest's violation report, and our violated-operational-abstraction collection and removal. These inner iterations enable Jtest to fully generate violating tests.

Given the collected operational violations, we select the first encountered test for each violated operational abstraction. So when there are multiple tests that violate the same operational abstraction, we select only the first encountered one instead of all of them. Since a selected violating test might violate multiple operational abstractions, we group together all of the operational abstractions violated by the same test. Then we sort the selected violating tests based on the number of their violated operational abstractions. We put the tests that violate more operational abstractions before those that violate fewer ones. The heuristic behind this is that a test that violates more operational abstractions might be more valuable than a test that violates fewer ones. When programmers cannot afford to inspect all violating tests, they can inspect just the top parts of the prioritized tests.

We finally produce a JUnit [23] test class, which contains the sorted list of violating test inputs as well as their violated operational abstractions. We developed a set of integration tools in Perl to fully automate the preceding steps, including invoking Daikon and Jtest, and postprocessing the text file. After running the integration tools, programmers can then execute or inspect the resulting sorted tests to verify the correctness of their executions. Optionally, programmers can add assertions for the test inputs as test oracles for regression testing.

One example of operational violations is shown in Figure 3. The example indicates a deficiency of the JAX test suite. The top part of

```
JAX Test 1:
  uniqueBoundedStack stack = new uniqueBoundedStack();
  assertTrue(!stack.isMember(2));
JAX Test 2:
  uniqueBoundedStack stack1 = new uniqueBoundedStack();
  uniqueBoundedStack stack2 = new uniqueBoundedStack();
  stack1.push(3);
  assertTrue(stack1.isMember(3));
  stack1.push(2);
  stack1.push(1);//because max is 2, this push cannot put 1 into stack1
  stack2.push(3);
  stack2.push(2);
  //the following assertion makes sure 1 is not in stack1
  assertTrue(stack1.isMember(1) == stack2.isMember(1));

Inferred postconditions for isMember:
  @post: [($pre(int, k) == 3) == ($result == true)]
  @post: [($pre(int, k) == 3) == (this.numberOfElements == 1)]

Violating Jtest-generated test input:
  uniqueBoundedStack THIS = new uniqueBoundedStack ();
  boolean RETVAL = THIS.isMember (3);
```

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

Figure 3 shows two relevant tests (JAX Tests 1 and 2) used for inferring the `isMember` method's two violated postconditions (`assertTrue` in the tests is JUnit's built-in assertion method). The postconditions are followed by the violating test input generated by Jtest. In the postconditions, `@post` is used to denote postconditions. The `$pre` keyword is used to refer to the value of an expression immediately before calling the method; the syntax to use `$pre` is `$pre(expressionType, expression)`. The `$result` keyword is used to represent the return value of the method.

The violated postconditions show the following behavior exhibited by the existing tests:

– The `isMember(3)` method is invoked iff its return value is `true`.
– The `isMember(3)` method is invoked iff the `numberOfElements` (after the method invocation) is `1`.

The test input of invoking `isMember(3)` method on an empty stack violates these two ungeneralizable postconditions.

4.2. Precondition removal technique

In the basic technique, when the existing test suite is deficient, the inferred preconditions might be overconstrained so that Jtest filters out valid test inputs during test generation and execution. However, we often need to exercise the unit under more circumstances than what is constrained by the inferred overconstrained preconditions. To address this, before we feed the annotated code to Jtest, we use a script to automatically remove all inferred preconditions, and we thus enable Jtest to 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]. Another issue with this technique is that removing inferred preconditions allows test generation tools to generate invalid test inputs if some values of a parameter type are invalid. We shall further discuss this invalid-input issue in Section 5.

Figure 4 shows one example of operational violations and the use of this technique. `@invariant` is used to denote class invariants. The example indicates a deficiency of the basic JUnit test suite, and the violating test exposes the fault detected by the original JAX test suite. The violated postconditions and invariant show the following behavior exhibited by the existing tests:

– After the invocation of the `pop()` method, the element on top of the stack is equal to the element on the second to top of the stack before the method invocation.
– After the invocation of the `pop()` method, the `numberOfElements` is equal to `0` or `1`.
– In the entries and exits of all the public methods, the `numberOfElements` is equal to `0`, `1`, or `2`.

Since the capacity of the stack is `2`, the inferred behavior seems to be normal and consistent with our expectation. Jtest generates a test that invokes `pop()` on an empty stack. In the exit of the `pop()` method, the `numberOfElements` is equal to `-1`. This value causes the evaluation of the first postcondition to throw an exception, and the evaluation of the second postcondition or the invariant to get the `false` value. By looking into the specifications shown in the appendix, we can know that the implementation does not appropriately handle the case where the `pop()` method is invoked on an empty stack; the specifications specify that the empty stack should maintain the same empty state when the `pop()` method is invoked.

The example in Figure 5 shows a deficiency of the JAX test suite, and the violating test exposes another new fault. This fault is not reported in the original experiment [37]. The inferred postcondition states

```
Basic JUnit Test 1:
  uniqueBoundedStack stack = new uniqueBoundedStack();
  stack.push(3);
  stack.pop();
Basic JUnit Test 2:
  uniqueBoundedStack stack = new uniqueBoundedStack();
  stack.push(3);
  stack.push(2);
  stack.pop();

Inferred postconditions for pop:
  @post: [( this.elems[this.numberOfElements] ==
        this.elems[$pre(int, this.numberOfElements)-1] )]
  @post: [this.numberOfElements == 0 ||
          this.numberOfElements == 1]
Inferred class invariant for uniqueBoundedStack:
  @invariant: [this.numberOfElements == 0 ||
               this.numberOfElements == 1 ||
               this.numberOfElements == 2]

Violating Jtest-generated test input:
 uniqueBoundedStack THIS = new uniqueBoundedStack ();
 THIS.pop ();
```

Figure 4. The first example of operational violations using the precondition removal technique

that the method return is equal to `-1` iff the `numberOfElements` is equal to `0`. The code implementer uses `-1` as the error indicator for calling the `top()` method on an empty stack instead of an `topEmptyStack` exception specified by the specifications shown in the appendix[1]. According to the specifications, this stack should also accommodate negative integer elements; this operational violation shows that using `-1` as an error indicator makes the `top` method work incorrectly when the `-1` element 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. The basic technique does not report this violation because of the overconstrained preconditions. The existing tests push only positive integers into the stack, so Daikon infers several preconditions for the `top` method, which prevent the `-1` element from being on top of the

[1] The assertion in `JAX Test 4` does not faithfully reflect the expected behavior of throwing a `topEmptyStack` exception specified in the appendix. In other words, the test writer did not implement `JAX Test 4` correctly with respect to the specifications.

```
JAX Test 3:
  uniqueBoundedStack stack = new uniqueBoundedStack();
  stack.push(3);
  stack.push(2);
  stack.pop();
  stack.pop();
  stack.push(3);
  stack.push(2);
  int oldTop = stack.top();
JAX Test 4:
  uniqueBoundedStack stack = new uniqueBoundedStack();
  assertTrue(stack.top() == -1);
JAX Test 5:
  uniqueBoundedStack stack1 = new uniqueBoundedStack();
  uniqueBoundedStack stack2 = new uniqueBoundedStack();
  stack1.push(3);
  assertTrue(stack1.top() == 3);
  stack1.push(2);
  stack1.push(1);
  stack2.push(3);
  stack2.push(2);
  assertTrue(stack1.top() == stack2.top());
  stack1.push(3);
  assertTrue(stack1.top() == 3);

Inferred postcondition for top:
 @post: [($result == -1) == (this.numberOfElements == 0)]

Violating Jtest-generated test input:
 uniqueBoundedStack THIS = new uniqueBoundedStack ();
 THIS.push (-1);
 int RETVAL = THIS.top ();
```

Figure 5. The second example of operational violations using the precondition removal technique

stack. One such precondition is:

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

where `@pre` is used to denote a precondition and `$assert` is used to denote an assertion statement within the loop body. Both the loop and the assertion statement form the precondition.

```
(1st iteration)
Inferred postcondition for isMember:
 @post: [($result == true) == (this.numberOfElements == 1)]

Violating Jtest-generated test input:
 uniqueBoundedStack THIS = new uniqueBoundedStack ();
 THIS.top ();
 THIS.push (2);
 boolean RETVAL = THIS.isMember (1);

(2nd iteration)
Inferred postcondition for isMember:
 @post:[($result == true) $implies (this.numberOfElements == 1)]

Violating Jtest-generated test input:
 uniqueBoundedStack THIS = new uniqueBoundedStack ();
 THIS.push (2);
 THIS.push (0);
 boolean RETVAL = THIS.isMember (0);
```

Figure 6. Operational violations during iterations

4.3. Iterations

After we perform the test selection using the techniques in Section 4.1 and 4.2, we can further run all the violating tests together with the existing ones to infer new operational abstractions. By doing so, we can automatically remove or weaken the operational abstractions violated by the violating tests. Based on the new operational abstractions, Jtest might generate new violating tests for the weakened or other new operational abstractions. We repeat the process described in Section 4.1 or 4.2 until there are no reported operational violations or until the user-specified maximum number of iterations has been reached. We call these iterations as *outer iterations*. Different from the inner iterations described in Section 4.1, these outer iterations operate in a larger scale. They mainly comprise the activities of the existing tests' execution, Daikon's operational-abstraction generation, our DbC annotation insertion, the inner iterations, and our test selection and augmentation. We have used a script to automate the outer iterations. In the rest of the paper, for the sake of brevity, iterations will refer to outer iterations by default.

Figure 6 shows two operational violations during the first and second iterations on the JAX test suite. The JAX test suite exhibits that the return of the `isMember()` method is `true` iff the `numberOfElements` after the method execution is equal to `1`. In the first iteration, a violating test shows that if the `numberOfElements` after the method execution is equal to `1`, the return of the `isMember()` method is not necessarily `true` (it can be `false`). After the first iteration, we add this violating test to the existing test suite. In the second iteration, with the augmented test suite, Daikon infers an updated postcondition by weakening the `==` predicate (meaning iff or $\Leftrightarrow$) to the `$implies` predicate (meaning $\Rightarrow$). The updated postcondition shows that if the return of the `isMember()` method is `true`, the `numberOfElements` after the method execution is equal to `1`. In the second iteration, another violating test shows that if the return of the `isMember()` method is `true`, the `numberOfElements` after the method execution is not necessarily equal to `1` (it can be equal to `2`). After the second iteration, we add this violating test to the existing test suite. In the third iteration, Daikon eliminates this `$implies` predicate since Daikon does not observe any correlation between the return of the `isMember()` method and the `numberOfElements`.

5. Discussion

5.1. Inferred Preconditions

The operational abstractions generated from the existing tests might not be consistent with the *oracle specifications*, which are the actual specifications (if any) supplied by the programmers. Assume that `OA_PRE` and `OS_PRE` are the domains constrained by the preconditions of the operational abstractions and the oracle specifications, respectively. Valid domains are the ones that satisfy the preconditions of the oracle specifications, and invalid domains are the ones that do not satisfy these preconditions. Recall that specification-based test generation tools such as Jtest usually generate and execute only the test inputs that satisfy the preconditions. The following is the analysis of different potential relationships between `OA_PRE` and `OS_PRE` and their effects on Jtest's test generation:

1. If `(OS_PRE - OA_PRE)` $\neq \emptyset$, Jtest does not generate any test inputs in the valid domain of `(OS_PRE - OA_PRE)`;
2. If `(OA_PRE - OS_PRE)` $\neq \emptyset$, Jtest might generate test inputs in the invalid domain of `(OA_PRE - OS_PRE)`;
3. If `OA_PRE = OS_PRE`, Jtest performs traditional specification-based test generations as if precondition specifications were written by programmers.

When the existing tests are only a few or insufficient, the basic technique suffers mainly from the effect in the first case. But the basic technique can guide test generation tools to fully exercise the space constrained by `OA_PRE`, especially the boundaries of the space. For example, assume there are two arguments `x` and `y` for a method, and the existing tests exercise the `(x, y)` with different combinations, such as the repeated occurrences of `(0, 1)`, `(1, 10)`, `(5, 0)`, `(7, 7)`, and `(10, 5)`. Daikon might infer the following preconditions:

```
@pre: 0 <= x <= 10
@pre: 0 <= y <= 10
```

even if the existing tests never exercise any of the following `(x, y)` combinations: `(0, 0)`, `(0, 10)`, `(10, 0)`, or `(10, 10)`. Test generation tools can generate these boundary combinations as test inputs to augment the existing tests. However, these boundary test inputs might not be selected by our approach unless they violate the existing operational abstractions.

The precondition removal technique involves removing all the automatically generated preconditions (`OA_PRE`). This addresses the the first case by changing the situation to the second case. Class invariants can be treated as preconditions and postconditions for every non-static, non-private method of the class. Class invariants usually do not involve arguments for a particular method but some object-field variables. Therefore, keeping overconstrained class invariants does not overrestrict the test generation tools that construct object states by calling method sequences, such as Jtest. In the integration of Daikon and Jtest, we keep the inferred class invariants in the precondition removal technique.

Since we do not have or require an oracle specification in our approach, we are uncertain whether the guidance of our inferred preconditions is underconstrained or overconstrained. We develop the basic technique to tackle the issue of being underconstrained and develop the precondition removal technique to tackle the issue of being overconstrained. In future work, we plan to explore the spectrum that lie between these two techniques. For example, we plan to use heuristics to remove or weaken parts of the inferred preconditions.

5.2. Inferred Postconditions

After we write oracle specifications for a method, a fault is exposed if a test input satisfies the preconditions, but its execution violates the postconditions. However, in our operational violation approach, a postcondition violation does not necessarily expose a fault, since either inferred preconditions or postconditions are not necessarily the same as the ones in the oracle specifications.

If the execution of a valid test input causes the violation of an inferred postcondition, there are two possible causes. The first cause could be that the postcondition in the operational abstractions is more constrained than the one in the oracle specifications. The operational violation indicates that the violating test exercises a new program feature, 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 the violating test reveals a fault in terms of correctness. Running the existing test suite on the code exhibits the normal behavior reflected by the postcondition, whereas the violating test makes the code exhibit the faulty behavior.

If the execution of an invalid test input (generated due to under-constrained preconditions) causes the violation of an inferred postcondition, the test input reveals a potential failure in terms of robustness [25, 8]. The undesirable behavior could suggest programmers to add the corresponding preconditions or input-checking code for defensive programming. Indeed, the programmers might feel that the invalid test input might not arise in the environment of using the code or it is just too costly to add preconditions or checking code; therefore, the programmers are not obligated to modify their existing code for handling the selected invalid test input. In practice, programmers often do not write down the preconditions or input-checking code that guards against invalid test inputs; these implicit assumptions are hidden and pose difficulties for other programmers to reuse the code or even for the code owners to maintain the code. The problems can be alleviated after our approach augments the existing test suite with these selected failure-revealing test inputs, even if programmers do not modify the code for adding preconditions or input checking. Indeed, if the tool that implements our approach keeps suggesting the programmers to inspect invalid test inputs that might not arise in the application environment, the programmers would possibly stop using the tool. To validate our approach against this pitfall, we need to conduct case studies with programmers in future work. To alleviate this potential pitfall, we could also improve the tool in future work to incorporate heuristics for classifying some violating tests as invalid test inputs and exclude them for inspection.

Human inspection and modification of the program or its specifications can take place either at the end of each iteration or after all the iterations terminate. In the latter way, we re-initiate a new sequence of iterations on the modified program or specifications. In either way, if programmers modify the program by fixing exposed bugs or adding input-checking code, we automatically incorporate these changes in the subsequent iteration by operating on the modified program. If programmers add preconditions for a method, we keep these manually added preconditions while applying the basic technique or the precondition removal technique in the subsequence iteration. Therefore these manually added preconditions can prevent test generation tools from generating corresponding invalid test inputs in the subsequent iteration.

If postconditions in the operational abstractions are less constrained than the ones in the oracle specifications, our approach might miss the chance of selecting fault-revealing tests if there is any fault. For example, the postconditions in the oracle specifications might not be able to be instantiated using Daikon's predefined grammars [12, 29, 10] or the existing tests are not sufficient to pass statistical tests for inferring postconditions. Even if the existing tests are sufficient enough for inferring the exact set of postconditions in the oracle specifications and the code is free of faults, our approach could not select any violating test either.

Note that some violating test inputs for `uniqueBoundedStack` (such as the ones in Figures 3 and 5) could be considered as equivalent to some existing tests if we make a uniformity hypothesis [3, 15] on the domain of stack elements. For example, the violating test in Figure 3 exercises the method sequence of invoking the constructor method, the `insert` method (with an integer argument), and the `top` method; this method sequence has been exercised by `JAX Test 5` if the program behaves similarly for different stack elements. However, the uniformity hypothesis is sometimes not satisfied when testing the behavior of this program: inserting `-1` into the stack can cause the subsequent method call of `top` to exhibit special behavior. In general, `uniqueBoundedStack` does not satisfy the uniformity hypothesis because inserting an element that already exists in the stack exhibits different behavior from the one exhibited by inserting an element that does not exist in the stack.

5.3. Inference Mechanisms of Daikon

Daikon predefines a set of grammars used to describe inferred invariants during invariant detection [12, 29, 10]. For scalar variables, Daikon defines three types of grammars: SingleScalar, TwoScalar, and ThreeScalar. SingleScalar is defined for invariants over a single numeric variable, such as equality with a constant "`x == a`", non-zero "`x != 0`", modulus "$\texttt{x} \equiv \texttt{a (mod b)}$", a small set of constants "$\texttt{x} \in \{\texttt{a, b, c}\}$", and lying within a range "$\texttt{a} \leqslant \texttt{x} \leqslant \texttt{b}$". The last two types of SingleScalar could be possibly expanded by running additional Jtest-generated tests. For example, theoretically "$\texttt{x} \in \{\texttt{a, b, c}\}$" can be violated and expanded to "$\texttt{x} \in \{\texttt{a, b, c, d}\}$" when Jtest generates a `d` value for `x`; however, in Daikon's default configuration, the maximum set size for this type of invariant is three. Therefore, "`x` $\in$ {`a, b, c`}" is eliminated when x with a `d` value is observed. Theoretically "`a` $\leqslant$ `x` $\leqslant$ `b`" can be violated and expanded to "`a` $\leqslant$ `x` $\leqslant$ `(b+1)`" when Jtest generates a `(b+1)` value for `x`. Statistical tests used by Daikon avoid the inference of "`a` $\leqslant$ `x` $\leqslant$ `(b+1)`" because an invariant is not displayed unless the confidence that the invariant occurs by chance is lower than a low threshold. These Daikon built-in mechanisms prevent our approach from selecting an infinite sequence of less interesting tests such as `x` with the values of `(a-1)`, `(b+1)`, `(a-2)`, `(b+2)`, ...

TwoScalar is defined for invariants over two numeric variables, such as ordering "`x` $\leqslant$ `y`" and functions "`y = fn(x)`". ThreeScalar is defined for invariants over three numeric variables, such as linear relationships "`z = ax + by + c`".

For sequence (array) variables, Daikon defines three types of grammars: SingleSequence, TwoSequence, and SequenceScalar. SingleSequence is defined for invariants over one sequence variable, such as "`a[]` contains no duplicates". TwoSequence is defined for invariants over two sequences, such as "`a[]` is a subsequence of `b[]`". SequenceScalar is defined for invariants over a scalar and a sequence, such as "`x` is a member of `a[]`".

Conditional invariants are invariants that are true only part of the time. One example of a conditional invariant is an inferred postcondition shown in Figure 6:

```
[($result == true) $implies (this.numberOfElements == 1)]
```

This postcondition is a conditional invariant because it depends on the predicate `($result == true)` being true. By default, Daikon turns off its inference for conditional invariants. Our experience shows that these conditional invariants are often useful for applying our approach; therefore, we turn on the inference for conditional invariants when using Daikon to implement our approach.

6. Experiments

Testing is used not only for finding bugs but also for increasing our confidence in the code under test. For example, generating and selecting tests for achieving better structural coverage can increase our confidence in the code although they do not find bugs; indeed, these tests can be used as regression tests executed on later versions for detecting regression bugs. Although our approach tries to fill gaps in the existing test suite or identify its weakness in order to improve its quality, our approach does not intend to be considered as a general approach for generating and selecting tests (based on the current program version) to increase the existing test suite's capability of exposing future arbitrarily introduced bugs (on future program versions) during program maintenance. Therefore, when we designed our experiments for assessing the approach, we did not use mutation testing [5] to measure the capability of the selected tests in finding arbitrary bugs in general. Instead, we conducted experiments to primarily measure the capability of the selected tests in revealing anomalous behavior on the real code, such as revealing a fault in terms of correctness or a failure in terms of robustness. We do not distinguish these two types of anomalous behavior because in the absence of specifications we often could not distinguish these two cases precisely. For example, the violating tests shown in Figure 4 and Figure 5 would have been considered as invalid tests for revealing failures if the actual precondition for `pop()` were `(this.numberOfElements > 0)` and the actual precondition for `push(int k)` were `(k >= 0)`; however, these two tests are valid fault-revealing tests based on the specifications shown in the appendix. Indeed, we could try to hand-construct specifications for these programs; however, the code implementation and comments for these programs alone are not sufficient for us to recover the specifications (especially preconditions) easily and we do not have easy access to the program intentions originally residing in code authors' mind. Note that if a selected test does not expose anomalous behavior, it might still provide value in filling gaps in the existing test suite. However, in the absence of specifications, it would be too subjective in judging these tests in terms of providing value; therefore, we did not perform such a subjective judgment in our experiments.

In particular, the general questions we wish to answer include:

1. Is the number of automatically generated tests large enough for programmers to adopt unit-test selection techniques?
2. Is the number of tests selected by our approach small enough for programmers to inspect affordably?
3. Do the tests selected by our approach have a high probability of exposing anomalous program behavior?
4. Do the operational abstractions guide test generation tools to better generate tests for violating the operational abstractions?
5. Does the tests selected by our approach have a higher probability of exposing anomalous program behavior than the tests selected by the residual branch coverage approach [33], which is a representative of the existing test selection techniques in practice?

We cannot answer all of these questions easily, so we designed experiments to give an initial sense of the general questions of efficacy of this approach. In the remaining of this section, we first describe the measurements in the experiments. We then present the experiment instrumentation. We finally describe the experimental results and threats to validity.

6.1. Measurements

In particular, we collected 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 alone in the absence of any operational abstraction. We call these tests as *unguided-generated tests*. 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, as well as the fourth and fifth questions.
– Anomaly-revealing selected test count (`#ART`): We measured the number of anomaly-revealing tests among the selected tests. These anomaly-revealing tests expose anomalous program behavior (related to either faults in terms of correctness or failures in terms of robustness). After all the iterations terminate, we manually inspect the selected tests, violated postconditions, and the source code to determine the anomaly-revealing tests. Although our test selection mechanism described in Section 4.1 guarantees that no two selected tests violate the same set of postconditions, multiple anomaly-revealing tests might suggest the same precondition or expose the same fault in different ways. This measurement is related to the third question, as well as the fourth and fifth questions.

We collected the `#AutoT` measurement for each subject program. We collected the `#SelT` and `#ART` measurements for each combination of the basic/precondition removal techniques, subject programs, and number of iterations. These measurements help answer the first three questions.

To help answer the fourth question, we used Jtest alone to produce unguided-generated tests, then ran the unguided-generated tests, and check them against the operational abstractions (keeping the preconditions) generated from the existing tests. We selected those unguided-generated tests that satisfied preconditions and violated postconditions. We then collected the `#SelT` and `#ART` measurements for each subject program, and compared the measurements with the ones for the basic technique.

In addition, we used Jtest alone to produce unguided-generated tests, then ran the unguided-generated tests, and check them against the operational abstractions (removing the preconditions) generated from the existing tests. We selected those unguided-generated tests that violated postconditions. We then collected the `#SelT` and `#ART` measurements for each subject program, and compared the measurements with the ones for the precondition removal technique.

To help answer the fifth question, we collected the `#SelT` and `#ART` measurements for each subject program using the residual branch coverage approach. We ran the unguided-generated tests, and selected those tests that exercised a new branch, which was not covered by any existing test or any previously selected test. Then we collected the `#SelT` and `#ART` measurements for the residual branch coverage approach.

6.2. Experiment instrumentation

Table I lists the subject programs that we used in the experiments. 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 6.3.

Among these subjects, `UB-Stack(JUnit)` and `UB-Stack(JAX)` are the example (Section 2) with the basic JUnit test suite and the JAX test suite (with one failing test removed), respectively [37]. `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 [38]. Daikon group members developed unit tests for 10 data structure classes in the textbook. Most of these unit tests use random inputs to extensively exercise the programs. We applied our approach on these classes, and five classes (the last five at the end of Table I) have at least one operational violation.

In the experiments, we used Daikon and Jtest to implement our approach. We developed a set of Perl scripts to integrate these two tools. In Jtest's configuration for the experiments, we set the length of calling sequence as two. We used Daikon's default configuration for the generation of operational abstractions except that we turned on the inference of conditional invariants. In particular, we first ran Jtest on each subject program to collect the `#AutoT` measurement in the absence of any operational abstraction. We exported the unguided-generated tests for each program to a JUnit test class. Then for each program, we conducted the experiment using the basic technique, and repeated it until we reached the third iteration or until no operational viola-

Table I. Subject programs used in the experiments.

program	*#pmethod*	*#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	14	166	2
`BinarySearchTree`	16	50	15	147	0
`DisjSets`	4	11	3	24	4
`QueueAr`	7	27	11	120	1
`StackAr`	8	20	16	133	1
`StackLi`	9	21	16	99	0

tions were reported for the operational abstractions generated from the previous iteration. At the end of each iteration, we collected the `#SelT` and `#ART` measurements. We performed a similar procedure for the precondition removal technique.

The Hansel tool is an extension to JUnit that adds code coverage testing to the testing framework [20]. Based on the Hansel tool, we developed a test selection tool based on residual branch coverage. We used the test selection tool to collect the `#SelT` and `#ART` measurements based on the residual branch coverage among the unguided-generated tests.

6.3. Experimental results

The fifth column of Table I shows the `#AutoT` results. From the results, we observed that except for the especially small `DisjSets` program, Jtest automatically generated nearly 100 or more tests. 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 we need test selection techniques since it is not practical to manually check the outputs of all these automatically generated tests.

The last column (`#ExT`) of Table I shows the number of the automatically generated tests that cause uncaught runtime exceptions. In the experiments, since all the test selection methods under comparison additionally select this type of tests, the `#SelT` and `#ART` measurements do not count them for the sake of better comparison.

Table II and Table III show the `#SelT` and `#ART` measurements for the basic technique and the precondition removal technique, respectively. In either table, the *iteration 1*, *iteration 2*, and *iteration 3* columns

Table II. The numbers of selected tests and anomaly-revealing selected tests using the basic technique for each iteration and the unguided-generated tests

program	*iteration 1*		*iteration 2*		*iteration 3*		*unguided*	
	`#SelT`	`#ART`	`#SelT`	`#ART`	`#SelT`	`#ART`	`#SelT`	`#ART`
`UB-Stack(JUnit)`	1	0	2	0				
`UB-Stack(JAX)`	3	0						
`RatPoly-1`	2	2						
`RatPoly-2`	1	1	1	1				
`RatPolyStack-1`								
`RatPolyStack-2`	1	0						
`BinaryHeap`	3	2	1	0			2	2
`BinarySearchTree`								
`DisjSets`	1	1					1	1
`QueueAr`	6	1					2	1
`StackAr`	5	1	1	0			1	1
`StackLi`								
median(`#ART`/`#SelT`)	20%		0%		0%		100%	
average(`#ART`/`#SelT`)	45%		25%		0%		88%	

Table III. The numbers of selected tests and anomaly-revealing selected tests using the precondition removal technique for each iteration and the unguided-generated tests

program	*iteration 1*		*iteration 2*		*iteration 3*		*unguided*	
	`#SelT`	`#ART`	`#SelT`	`#ART`	`#SelT`	`#ART`	`#SelT`	`#ART`
`UB-Stack(JUnit)`	15	5	6	1	1	0	4	1
`UB-Stack(JAX)`	25	9	4	0			3	1
`RatPoly-1`	1	1						
`RatPoly-2`	1	1					1	1
`RatPolyStack-1`	12	8	5	2	1	0		
`RatPolyStack-2`	10	7	2	0				
`BinaryHeap`	8	6	8	6	6	0	4	3
`BinarySearchTree`	3	3					1	1
`DisjSets`	2	2					1	1
`QueueAr`	11	1	4	1			4	1
`StackAr`	9	1	1	0			1	1
`StackLi`	2	0					1	0
median(`#ART`/`#SelT`)	68%		17%		0%		75%	
average(`#ART`/`#SelT`)	58%		22%		0%		62%	

show the results for three iterations. In Table II, the *unguided* column shows the results for selecting unguided-generated tests that satisfy preconditions and violate postconditions. In Table III, the *unguided* column shows the results for selecting unguided-generated tests that violate postconditions (no matter whether they satisfy preconditions). In either table, for those `#SelT` with the value of zero, their entries and their associated `#ART` entries are left blank. The bottom two rows of either table show the median and average percentages of `#ART` among `#SelT`. In the calculation of the median or average percentage, the entries with a `#SelT` value of zero are not included.

The numbers of tests selected by both techniques vary across different programs but on average their numbers are not large, so their executions and outputs could be verified with affordable human effort. The basic technique selects fewer tests than the precondition removal technique. This is consistent with our hypothesis that the basic technique might overconstrain test generation tools. We observed that the number of tests selected by either technique is higher than the number of tests selected by checking unguided-generated tests against operational abstractions. This indicates that operational abstractions guide Jtest to better generate tests to violate them. Specifically, the precondition removal technique gives more guidance to Jtest for generating anomaly-revealing tests than the basic technique. There are only two subjects for which the basic technique generates anomaly-revealing tests but Jtest alone does not generate any (shown in Table II); however, the precondition removal technique generates more anomaly-revealing tests than Jtest alone for most subjects (shown in Table III).

We observed that, in the experiments, the selected tests by either technique have a high probability of exposing anomalous program behavior. In the absence of specifications, we suspect that many of these anomaly-revealing tests are failure-revealing test inputs; programmers can add preconditions, condition-checking code, or just pay attention to the undesirable behavior when the code's implicit assumptions are not written down.

We describe a concrete case for operational violations in the experiments as follows. `RatPoly-1` and `RatPoly-2` are two student solutions to an assignment of implementing `RatPoly`, which represents an immutable single-variate polynomial expression, such as "0", "$x - 10$", and "$x^3 - 2 * x^2 + 53 * x + 3$". In `RatPoly`'s class interface, there is a method `div` for `RatPoly`'s division operation, which invokes another method `degree`; `degree` returns the largest exponent with a non-zero coefficient, or 0 if the `RatPoly` is "0". After we ran with Daikon the instructor-provided test suite on both `RatPoly-1` and `RatPoly-2`, we got the same DbC annotations for both student solutions. The precondition

```
Inferred postcondition for degree:
  $result >= 0

Violating Jtest-generated test input (for RatPoly-1):
   RatPoly t0 = new RatPoly(-1, -1);//represents -1*x^-1
   RatPoly THIS = new RatPoly (-1, 0);//represents -1*x^0
   RatPoly RETVAL = THIS.div (t0);//represents (-1*x^0)/(-1*x^-1)

Violating Jtest-generated test input (for RatPoly-2):
   RatPoly t0 = new RatPoly(1, 0);//represents 1*x^0
   RatPoly THIS = new RatPoly (1, -1);//represents 1*x^-1
   RatPoly RETVAL = THIS.div (t0);//represents (1*x^-1)/(1*x^0)
```

Figure 7. Operational violations for RatPoly-1/RatPoly-2

removal technique selects one violating test for each student solution. The selected violating test for `RatPoly-1` is different from the one for `RatPoly-2`; this result shows that Jtest takes the code implementation into account when generating tests to violate the given DbC annotations. The selected test for `RatPoly-1` makes the program infinitely loop until a Java out-of-memory error occurs and the selected test for `RatPoly-2` runs normally with termination and without throwing exceptions. These tests are not generated by Jtest alone without being guided with operational abstractions. After inspecting the code and its comments, we found that these selected tests are invalid one, because there is a precondition `e >= 0` for `RatPoly(int c, int e)`. This case shows that the operational abstraction approach can help generate test inputs to crash a program and then programmers can improve their code's robustness when specifications are absent.

We observed that although those non-anomaly-revealing selected 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. We observed, in the experiments, 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 we comment out the violated operational abstractions.

Table IV shows the experimental results for the residual branch coverage approach. The last two columns of Table IV show the `#SelT` and `#ART` measurements in the residual branch coverage approach. The

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

program	*#total branch*	*#pre residual*	*#post residual*	`#SelT`	`#ART`
`UB-Stack(JUnit)`	41	13	5	5	1
`UB-Stack(JAX)`	41	1	1		
`RatPoly-1`	125	3	3		
`RatPoly-2`	139	9	9		
`RatPolyStack-1`	22	7	6	1	0
`RatPolyStack-2`	16	0	0		
`BinaryHeap`	34	2	0	1	0
`BinarySearchTree`	56	7	7		
`DisjSets`	10	0	0		
`QueueAr`	21	2	0	2	0
`StackAr`	20	1	0	1	0
`StackLi`	21	6	5	1	0
median(`#ART`/`#SelT`)	–			0%	
average(`#ART`/`#SelT`)	–			3%	

second column (`#total branch`) of Table IV shows the count of the total branches for each subject. The third column (`#pre residual`) presents the count of residual branches after executing the existing tests but before executing any unguided-generated tests. The fourth column (`#post residual`) presents the count of residual branches after executing both the existing tests and the selected unguided-generated tests. The bottom two rows of Table IV show the median and average percentages of `#ART` among `#SelT`.

We observed that the existing tests have already left no residual branches on two of the subjects. The unguided-generated tests can further reduce the count of residual branches on half of the subjects. The number of the selected tests or anomaly-revealing tests in the residual coverage approach is fewer than the one 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, the tests selected by the residual branch coverage approach cannot expose most of the anomalous program behavior that is exposed by the tests selected by the operational violation approach. This suggests that combining the residual branch coverage approach and the operational violation approach may provide a better solution for unit-test selection.

6.4. Threats to validity

The threats to external validity primarily include the degree to which integrated third-party tools, the subject programs, and test cases are representative of true practice. These threats could be reduced by more experiments on wider types of subjects and third-party tools. Parasoft Jtest 4.5 is one of the testing tools popularly used in industry and the only specification-based test generation tool available to us at the moment. Daikon is the only publicly available tool for generating operational abstractions. Daikon's scalability has recently been tackled by using incremental algorithms for invariant detection [34]. In our approach, we use Daikon to infer invariants based on only manual tests in addition to selected violating tests; the size of these tests is often small. However, Jtest 4.5 is not designed for being used in an iterative way; if some operational abstractions can be violated, we observed that the number of inner iterations can be more than a dozen and the elapsed time could be longer than five minutes for some subjects. We expect that the scalability of Jtest in our setting could be addressed by enhancing it to support incremental test generation when DbC annotations are being changed. Furthermore, the elapsed time for Jtest's test generation can be reduced by enhancing it to avoid generating redundant tests [41]. Alternatively we can use other specification-based tools with more efficient mechanisms for test generation, such as Korat [4].

We mainly used data structures as our subject programs and the programs are relatively small (the scalability of Jtest 4.5 poses difficulties for us to try large subjects, but note that this is not the inherent limitation of our approach but the limitation of one particular implementation of our approach). Although data structures are better suited to the use of invariant detection and design-by-contract specifications, Daikon has been used on wider types of programs [10]. The success of our approach on wider types of programs also depends on the underlying testing tool's capability of generating test inputs to violate specifications if there exist violating test inputs. We expect that the potential of our approach for wider types of programs could be further improved if we use specification-based testing tools with more powerful test generation capability, such as Korat [4] or CnC [9].

The main threats to internal validity include instrumentation effects that can bias our results. Faults in our integration scripts, Jtest, or Daikon might cause such effects. To reduce these threats, we manually inspected the intermediate results of most program subjects. The main threats to construct validity include the uses of those measurements in our experiments to assess our approach. We measured the number of anomaly-revealing tests to evaluate the value of selected tests. In future work, we plan to measure some other possible attributes of the selected tests. We used residual branch coverage approach as a representative of the existing test selection approaches to compare with our approach. In future work, we plan to use residual dataflow coverage for comparison, since the fault-exposing capability of data flow coverage criteria has been shown to be better than the one of branch coverage criteria [14].

7. Related work

We first discuss how our work relates to other projects on specification-based, state-based, and structural testing. We then compare our work with other testing approaches based on operational abstractions.

7.1. Specification-based, state-based, and structural testing

Boyapati et al. develop the Korat tool to systematically explore the input space of a given Java predicate that represents the method preconditions [4]. Korat can efficiently generate all non-isomorphic inputs (within a given input size) that satisfy the preconditions. The AsmLT [13, 16] tool adopts Korat's input space pruning technique to generate test inputs from an abstract-state-machine specification language. Based on DbC specifications for a Java class, Parasoft Jtest automatically generate unit tests that satisfy preconditions and try to violate postconditions [32]. All these specification-based test generation tools require *a priori* specifications. When specifications are provided for a unit *a priori*, Chang and Richardson use specification coverage criteria to suggest a candidate set of test cases that exercise new aspects of the specification [6]. Our approach uses generated operational abstractions to guide test generation tools and select generated tests without requiring *a priori* specifications. Our approach allows programmers to gain some benefits of specification-based testing without the pain of writing specifications. Although in this work, we integrated Daikon and Jtest to implement our approach, we plan to implement our approach by using some other specification-based unit-test generation tools [4, 17].

In state-based testing, our previous work develops Rostra [41], a framework for detecting redundant unit tests, and presents five fully automatic techniques for representing and comparing object states. Rostra has been used to select non-redundant tests generated by test-generation tools; the selected tests preserve the ability to detect faults and the structural coverage of the original test suite. The operational violation approach selects a much smaller set of tests for inspection and the selected tests are not intended to preserve the fault-detection capability; the operational violation approach intends to help programmers spend their inspection time on a small subset of generate tests that exhibit new interesting behavior.

In white-box testing (such as testing based on residual structural coverage [33]), programmers 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 [35, 36]. Clustering and sampling the execution profiles can also be used to select a list of tests for inspection and selection [11]. Our approach complements these existing test selection approach based on structural coverage.

7.2. Operational-abstraction-based testing

Harder et al. present a testing technique based on operational abstractions [21]. Their operational difference technique starts with an operational abstraction generated from 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. In addition, their operational difference approach selects tests mainly for regression testing, whereas our approach selects tests mainly for inspection.

Hangal and Lam develop the DIDUCE tool to detect bugs and track down the root causes [19]. 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. 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 access sites. Also DIDUCE does not investigate the effects of operational abstractions on test generation.

Nimmer and Ernst use failed static verification attempts to indicate the deficiencies in the unit tests [30]. The unverifiable invariants indicate unintended properties and programmers 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 overconstrained preconditions, it is hard to tell whether it is generalizable to the actual preconditions. In our approach, the precondition removal technique tries to tackle this problem.

Gupta and Heidepriem propose a new structural coverage criterion called invariant-coverage criterion for dynamic detection of program invariants [18]. Their experimental results showed that invariant-coverage test suites could effectively augment branch coverage and definition-use pair coverage test suites to remove spurious inferred invariants. They manually generated all the tests in their experiments. The main goal of their approach is to generate suitable tests that support accurate detection of program invariants. Our approach focuses on selecting the tests that violate the inferred postconditions. Our approach additionally considers the effects of inferred preconditions and postconditions on automatic test generation. Their invariant-coverage criterion can be incorporated in the underlying test generation that our approach integrates.

In previous work, we propose a generic framework for a feedback loop between test generation and dynamic behavior inference [42]. In the feedback loop, we can mutually enhance both test generation and dynamic behavior inference iteratively. We have implemented a feedback loop between test generation and equivalent-object-state inference [40]. The approach in this work focuses on operational abstractions in the form of DbC specifications, which is another instance of the feedback loop. The framework also accommodates other kinds of operational abstractions, such as protocol specification [1, 39] and algebraic specification [22]. In future work, we plan to implement these other operational abstractions in our approach.

8. Conclusion

Selecting automatically generated test inputs 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. These operational abstractions can guide test generation tools to better produce test inputs to violate the abstractions. We have developed the operational violation approach for selecting generated tests that violate operational abstractions; these selected violating tests are good candidates for inspection, since they exercise new program features that are not covered by the existing tests. We have conducted experiments on applying the approach on a set of data structures. Our experimental results have shown that the size of the selected tests is reasonably small for inspection, the selected tests generally expose new interesting behavior filling the gaps not covered by the existing test suite, and the selected tests have a high probability of exposing anomalous program behavior (either faults or failures) in the code.

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.

Acknowledgments

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 thank Parasoft Co. for their assistance in our use of their Jtest tool. We thank Michael Ernst, Ryan Newton, and the anonymous ASE 03 reviewers and ASE journal reviewers for their valuable feedback on the earlier version of this paper. 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.

References

1. Glenn Ammons, Rastislav Bodik, and James R. Larus. Mining specifications. In *Proc. 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 4–16, 2002.
2. Kent Beck. *Test Driven Development: By Example.* Addison-Wesley, 2003.
3. Gilles Bernot, Marie Claude Gaudel, and Bruno Marre. Software testing based on formal specifications: a theory and a tool. *Software Engineering Journal*, 6(6):387–405, 1991.

4. Chandrasekhar Boyapati, Sarfraz Khurshid, and Darko Marinov. Korat: automated testing based on Java predicates. In *Proc. International Symposium on Software Testing and Analysis*, pages 123–133, 2002.
5. Timothy A. Budd, Richard A. DeMillo, Richard J. Lipton, and Frederick G. Sayward. Theoretical and empirical studies on using program mutation to test the functional correctness of programs. In *Proc. 7th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 220–233, 1980.
6. Juei Chang and Debra J. Richardson. Structural specification-based testing: automated support and experimental evaluation. In *Proc. 7th ESEC/FSE*, pages 285–302, 1999.
7. Yoonsik Cheon and Gary T. Leavens. A simple and practical approach to unit testing: The JML and JUnit way. In *Proc. 16th European Conference Object-Oriented Programming*, pages 231–255, June 2002.
8. Christoph Csallner and Yannis Smaragdakis. JCrasher: an automatic robustness tester for Java. *Software: Practice and Experience*, 34:1025–1050, 2004.
9. Christoph Csallner and Yannis Smaragdakis. Check 'n' Crash: Combining static checking and testing. In *Proc. 27th International Conference on Software Engineering*, pages 422–431, May 2005.
10. Publications using the Daikon invariant detector tool, 2006. `http://pag.csail.mit.edu/daikon/pubs/`.
11. William Dickinson, David Leon, and Andy Podgurski. Finding failures by cluster analysis of execution profiles. In *Proc. 23rd International Conference on Software Engineering*, pages 339–348, 2001.
12. Michael D. Ernst, Jake Cockrell, William G. Griswold, and David Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE Transactions on Software Engineering*, 27(2):99–123, 2001.
13. Foundations of Software Engineering, Microsoft Research. The AsmL test generator tool. `http://research.microsoft.com/fse/asml/doc/AsmLTester.html`.
14. Phyllis G. Frankl and Stewart N. Weiss. An experimental comparison of the effectiveness of branch testing and data flow testing. *IEEE Transactions on Software Engineering*, 19(8):774–787, August 1993.
15. Marie-Claude Gaudel. Testing can be formal, too. In *Proc. 6th International Joint Conference CAAP/FASE on Theory and Practice of Software Development*, pages 82–96, 1995.
16. Wolfgang Grieskamp, Yuri Gurevich, Wolfram Schulte, and Margus Veanes. Generating finite state machines from abstract state machines. In *Proc. International Symposium on Software Testing and Analysis*, pages 112–122, 2002.
17. Neelam Gupta. Generating test data for dynamically discovering likely program invariants. In *Proc. ICSE 2003 Workshop on Dynamic Analysis*, pages 21–24, May 2003.
18. Neelam Gupta and Zachary V. Heidepriem. A new structural coverage criteria for dynamic detection of program invariants. In *Proc. 18th IEEE International Conference on Automated Software Engineering*, pages 49–58, 2003.
19. Sudheendra Hangal and Monica S. Lam. Tracking down software bugs using automatic anomaly detection. In *Proc. 24th International Conference on Software Engineering*, pages 291–301, 2002.
20. Hansel 1.0, 2003. `http://hansel.sourceforge.net/`.

21. Michael Harder, Jeff Mellen, and Michael D. Ernst. Improving test suites via operational abstraction. In *Proc. 25th International Conference on Software Engineering*, pages 60–71, 2003.
22. Johannes Henkel and Amer Diwan. Discovering algebraic specifications from Java classes. In *Proc. 17th European Conference on Object-Oriented Programming*, pages 431–456, 2003.
23. JUnit, 2003. `http://www.junit.org`.
24. Bogdan Korel and Ali M. Al-Yami. Assertion-oriented automated test data generation. In *Proc. the 18th International Conference on Software Engineering*, pages 71–80, 1996.
25. Nathan P. Kropp, Philip J. Koopman Jr., and Daniel P. Siewiorek. Automated robustness testing of off-the-shelf software components. In *Proc. the 28th IEEE International Symposium on Fault Tolerant Computing*, pages 230–239, 1998.
26. Gary T. Leavens, Albert L. Baker, and Clyde Ruby. Preliminary design of JML: A behavioral interface specification language for Java. Technical Report TR 98-06i, Department of Computer Science, Iowa State University, June 1998.
27. Bertrand Meyer. *Eiffel: The Language.* Prentice Hall, New York, N.Y., 1992.
28. Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML.* MIT Press, Cambridge, MA, 1989.
29. Jeremy W. Nimmer. Automatic generation and checking of program specifications. Technical Report 852, MIT Laboratory for Computer Science, Cambridge, MA, June 2002.
30. Jeremy W. Nimmer and Michael D. Ernst. Static verification of dynamically detected program invariants: Integrating Daikon and ESC/Java. In *Electronic Notes in Theoretical Computer Science*, volume 55. Elsevier, 2001.
31. Parasoft. Jcontract manuals version 1.5. Online manual, October 2002. `http://www.parasoft.com/`.
32. Parasoft. Jtest manuals version 4.5. Online manual, April 2003. `http://www.parasoft.com/`.
33. Christina Pavlopoulou and Michal Young. Residual test coverage monitoring. In *Proc. 21st International Conference on Software Engineering*, pages 277–284, 1999.
34. Jeff H. Perkins and Michael D. Ernst. Efficient incremental algorithms for dynamic detection of likely invariants. In *Proc. 12th ACM SIGSOFT 12th International Symposium on Foundations of Software Engineering*, pages 23–32, 2004.
35. Gregg Rothermel, Roland Untch, Chengyun Chu, and Mary Jean Harrold. Prioritizing test cases for regression testing. *IEEE Transactions on Software Engineering*, 27(10):929–948, 2001.
36. Amitabh Srivastava and Jay Thiagarajan. Effectively prioritizing tests in development environment. In *Proc. International Symposium on Software Testing and Analysis*, pages 97–106, 2002.
37. David Stotts, Mark Lindsey, and Angus Antley. An informal formal method for systematic JUnit test case generation. In *Proc. the 2002 XP/Agile Universe*, pages 131–143, 2002.
38. Mark Allen Weiss. *Data Structures and Algorithm Analysis in Java.* Addison Wesley, 1999.
39. John Whaley, Michael C. Martin, and Monica S. Lam. Automatic extraction of object-oriented component interfaces. In *Proc. the International Symposium on Software Testing and Analysis*, pages 218–228, 2002.

40. Tao Xie, Darko Marinov, and David Notkin. Improving generation of object-oriented test suites by avoiding redundant tests. Technical Report UW-CSE-04-01-05, University of Washington Department of Computer Science and Engineering, Seattle, WA, Jan. 2004.
41. Tao Xie, Darko Marinov, and David Notkin. Rostra: A framework for detecting redundant object-oriented unit tests. In *Proc. 19th IEEE International Conference on Automated Software Engineering*, pages 196–205, Sept. 2004.
42. Tao Xie and David Notkin. Mutually enhancing test generation and specification inference. In *Proc. 3rd International Workshop on Formal Approaches to Testing of Software*, volume 2931 of *LNCS*, pages 60–69, 2003.

9. Appendix

Stotts et al. [37] provided formal algebraic specifications for a bounded stack, whose behavior has a considerable overlap with `uniqueBoundedStack` (shown in Figure 1). The only difference between the specified bounded stack and `uniqueBoundedStack` is the way of handling the uniqueness of the elements stored in a stack. The specification is specified using the functional language SML [28]:

```
datatype BST = New of int |
               push of BST * int ;
fun isEmpty (New(n)) = true |
    isEmpty (push(B,e)) = false ;
fun maxSize (New(n)) = n |
    maxSize (push(B,e)) = maxSize(B) ;
fun getSize (New(n)) = 0 |
    getSize (push(B,e)) = if getSize(B)=maxSize(B)
                          then maxSize(B)
                          else getSize(B)+1 ;
fun isFull (New(n)) = n=0 |
    isFull (push(B,e)) = if getSize(B)>=maxSize(B)-1
                         then true
                         else false ;
exception topEmptyStack;
fun top (New(n)) = raise topEmptyStack |
    top (push(S,e)) = if isFull(S)
                      then top(S)
                      else e ;
fun pop (New(n)) = New(n) |
    pop (push(S,e)) = if isFull(S)
                      then pop(S)
                      else S ;
```