Specification-based Test Repair Using a Lightweight Formal Method

Guowei Yang, Sarfraz Khurshid, and Miryung Kim

The University of Texas at Austin
guoweiyang@utexas.edu, {khurshid, miryung}@ece.utexas.edu

Abstract. When a program evolves, its test suite must be modified to reflect changes in requirements or to account for new feature additions. This problem of modifying tests as a program evolves is termed *test repair*. Existing approaches either assume that updated implementation is correct, or assume that most test repairs require simply fixing compilation errors caused by refactoring of previously tested implementation. This paper focuses on the problem of repairing *semantically broken or outdated* tests by leveraging specifications. Our technique, SPECTR, employs a lightweight formal method to perform specification-based repair. Specifically, SPECTR supports the Alloy language for writing specifications and uses its SAT-based analyzer for repairing JUnit tests. Since SPECTR utilizes specifications, it works even when the specification is modified but the change has not yet been implemented in code—in such a case, SPECTR is able to repair tests that previous techniques would not even consider as candidates for test repair. An experimental evaluation using a suite of subject programs with pre-conditions and post-conditions shows SPECTR can effectively repair tests even for programs that perform complex manipulation of dynamically allocated data.

1 Introduction

Testing is the most commonly used technique for validating software quality. While conceptually simple, testing can be expensive and involves much manual effort, specifically in writing test cases and describing expected test outputs. To reduce this cost, regression test suites are commonly used to check behavioral modifications as a program evolves. However, behavioral modifications may render certain existing tests invalid due to new feature additions or bug fixes, which in turn modify the expected test outputs.

Several research projects have addressed this problem of *test repair* [5, 6, 8]. Existing techniques can fix compilation errors in tests caused by simple refactorings such as method renamings or signature changes, so that the old tests could run as before. Some can modify test assertions to ensure those tests that passed before could still pass. However, all existing techniques perform test updates with respect to implementation changes, assuming that implementation is always correct. If the specification changes, but the implementation has not yet been modified or has been modified incorrectly, existing techniques are not able to repair tests to correctly reflect the updated specifications.

This paper presents SPECTR, a novel specification-based test repair technique using a lightweight formal method. Given the specifications of a modified program—*pre-conditions* defining expected inputs and *post-conditions* defining expected behavior—and an existing test suite, SPECTR repairs each test that exercises modified behavior. Specifically, it repairs test assertions that check the actual output against the expected output, so that failing tests reflect specification violation and passing tests reflect specification conformance.

As an enabling technology, SPECTR uses the Alloy tool set [13]. Alloy is a first-order declarative language based on relations, and is particularly suitable for expressing structural invariants on graphs, such as class invariants on object-graphs in a Java program. The Alloy tool set includes a fully automatic SAT solver engine that checks Alloy formulas within a given *scope*, i.e., bound on the universe of discourse. The back-end deployment of state-of-the-art SAT solvers makes the Alloy tool set particularly effective for test repair.

Given Alloy specifications, SPECTR uses a SAT solver to compute expected outputs for test assertions using post-conditions. The key insight behind our approach is that because each test case represents a single program execution for deterministic programs, updating a test oracle needs to explore only one execution behavior and does not need enumerate all possible behaviors. Thus SPECTR differs from previous testing and verification techniques using Alloy [7,9, 16, 21] by avoiding the traditional state-space explosion. Moreover, for manually written tests, SPECTR allows utilizing the tester's intuition behind the design of test inputs, since they directly form a part of the repaired test cases.

SPECTR repairs JUnit [1] tests that have a fairly general structure with three primary components: (1) *initialization*—initializing input values, i.e., the pre-state, for the *sequence* of methods under test, e.g., using explicit object allocations and field assignments, (2) *execution*—invoking the sequence of methods under test on the inputs, and (3) *assertion*—checking the post-state for the sequence using a test assertion, e.g., using the `assertEquals` method in `org.junit.Assert`. To repair a test, SPECTR first uses the initialization component to initialize an Alloy *instance* that represents the pre-state. Next, it uses Alloy to compute an expected post-state subject to the execution component. Finally, it updates affected assertions to reflect behavioral conformance to the updated specification.

SPECTR makes it possible to repair tests even before the implementation is modified to reflect the updated specifications. Thus, SPECTR directly supports test-driven development, a key practice behind the success of Extreme Programming and other agile software development processes. To the best of our knowledge, SPECTR is the first such technique for test repair.

This paper makes the following contributions:

- **Specification-based test repair.** We introduce the idea of repairing tests to reflect modifications to expected behavior as encoded in specifications. Previous techniques for test repair are based on implementation changes, assuming that updated code is always correct. Therefore, they do not handle semantic changes with respect to modified specifications.

- **A lightweight formal method for test repair.** To our knowledge, SPECTR is the first technique for test repair using a lightweight formal method. It leverages the Alloy tool set and presents a non-conventional application of propositional satisfiability solvers for repairing tests.
- **Evaluation.** We perform an experimental evaluation using our prototype embodiment of SPECTR to repair tests for a suite of subject programs. Our experiments show that SPECTR effectively repairs tests, even for programs that perform complex manipulations of dynamically allocated data.

While our approach is based on Alloy specifications, our ideas generalize to programs annotated using different specification languages, such as the Java Modeling Language, which enhances the applicability of our approach. In fact, our approach directly applies to code with Java Modeling Language (JML) annotations: the JForge tool [7] performs Alloy-based static analysis of JML annotated code and provides an enabling technology for our technique.

2 Related Work

Test Repair The need of test repair is well-recognized in regression testing [3, 19] and software evolution [26]. Recent years have seen several frameworks that automate test repair [5,6,8,18]. The key difference between SPECTR and previous work is SPECTR's use of specifications for test repair. Previous techniques for test repair use changes in implementation to repair tests, and hence can only repair tests to reflect actual behavior, which may not be the *intended* behavior. In contrast, SPECTR can repair tests even when the modified implementation is buggy. Indeed, SPECTR does not even require the implementation to be modified before the tests are repaired.

Daniel et al. [5,6] proposed a technique which performs a combined dynamic and static analysis to find test repairs that developers are likely to accept. However, their approach assumes the implementation is correct, and then repairs failing tests by recording its runtime behavior. [18] proposed an approach to repairing test cases for evolving method declarations. It only repairs test case compilation errors that depend on changes in parameters or return values. It assumes that the original functionality is preserved for the given test inputs.

Test repair has also been investigated for GUI-based systems, where it is common for developers to create test scripts using record-and-replay testing tools in GUI testing. The scripts generated in this way are quite fragile and easy to be broken when the system changes. To address this problem, Memon [17] proposed techniques for correcting sequences of test scripts so that they compile with the tested application. More recently, Grechanik et al. [11] presented a technique to identify modified GUI objects and locate test script statements that reference these modified GUI objects, so the test engineers can fix the test scripts.

Debugging Recent years have seen much progress in automated techniques for removing bugs, i.e., debugging – the process of locating faults, i.e., fault localization [4, 12, 15] and fixing them, i.e., program repair [10, 14, 24, 27]. Test repair is a special case of program repair where the program to repair is the old test suite. However, existing techniques for program repair are not well suited

for test repair since they are ambivalent of the specific structure of test case. In contrast, test repair techniques utilize this structure for enhanced effectiveness.

Alloy The Alloy tool-set has provided an enabling technology for various analyses for Java programs, including static checking using Jalloy [21], systematic testing using TestEra [16], data structure repair using Tarmeem [25], and most recently for program repair by Gopinath et al [10]. Our work shares insights with previous work and provides a novel use of the Alloy tool-set in test repair. The problem of test repair has similarities with the problem of test generation and the problem of program repair. SPECTR's technical approach is different from TestEra's, which generates inputs at the concrete level using sequences of field assignments and uses the Alloy Analyzer to evaluate Alloy post-conditions as test oracles. In contrast, SPECTR supports method sequences for input creation, enables re-use of existing test inputs, and generates test assertions that directly check correctness criteria. Also, SPECTR's approach is different from Gopinath et al.'s approach for program repair, which repairs faulty object field assignment statements. In contrast, SPECTR repairs JUnit test assertions, which are written using arbitrary Java expressions.

N-version Programming Our work bears resemblance[1] to N-version programming—a methodology where the same initial specification is used to create $N \geq 2$ functionally equivalent programs to enable fault tolerance [2]. There are three basic differences between our approach and N-version programming. First, we are performing *specification*-based repair where the specification is in a *declarative* language. We do not have two (or more) *imperative* programs implementing the same specification – the central condition for N-version programming. Second, N-version programming does not account for specification evolution, which is the central theme of our work. Third, N-version programming is defined for fault tolerance, not for test repair. However, we could generalize the spirit of N-version programming to view a specification—assuming it is *executable*—itself as one program version that may evolve. Then, after an evolution, the results of specification execution, if feasible, can be used to repair tests. For Alloy specifications, execution is made feasible by Alloy's SAT-based back-end, which is indeed the enabling technology for SPECTR's test repair. It is plausible to optimize solving of Alloy formulas in the specific context of test repair, but that is an open research problem.

3 Illustrative Example

This section presents an example to illustrate SPECTR's test repair process; we describe basic Alloy syntax and semantics as we introduce it; details on Alloy can be found elsewhere [13].

SPECTR takes as input an old test and a modified specification and repairs the old test. To illustrate, consider a singly-linked acyclic list data structure that stores integers in sorted order. Fig. 1 illustrates test repair for this example; the figure shows ① a Java declaration for lists; ② an old Alloy specification that

[1] We thank an anonymous reviewer for pointing us to N-version programming.

①	**Code**	`public class List {`
		`  Node header;`
		`  static class Node {int elem; Node next;}`
		`  public int size(){...}`
		`  public void add(int){...}`
		`}`

②	**Old Spec**:
	acyclic, sorted lists with unique elements
		`pred RepOk(l: List, s: State) {`
		` all n: l.(header.s).*(next.s) {`
		`  n not in n.^(next.s) // list is acyclic`
		`  // list is sorted with unique elements`
		`  some n.(next.s) => n.(elem.s) < n.(next.s).(elem.s)`
		` }`
		`}`
		
		`pred add_pre(l: List, x: Int, s: State) {`
		` RepOk[l, s]`
		`}`
		
		`pred add_post(l: List, x: Int, s, s': State) {`
		` RepOk[l, s']`
		` l.(header.s').*(next.s').(elem.s')`
		`  = l.(header.s).*(next.s).(elem.s) + x`
		`}`
		
		`pred size_pre(l: List, s: State) {`
		` RepOk[l, s]`
		`}`
		
		`pred size_post(l: List, result: Int, s: State) {`
		` result = #l.(header.s).*(next.s)`
		`}`

③	**New Spec**:
	acyclic, sorted lists that allow repetitions
	
	Modified Spec:
	"RepOk",
	"add_post"
	
	Unchanged Spec:
	"add_pre",
	"size_pre",
	"size_post"
		`pred RepOk(l: List, s: State) {`
		` all n: l.(header.s).*(next.s) {`
		`  n not in n.^(next.s) // list is acyclic`
		`  // list is sorted, while allowing repetitions`
		`  some n.(next.s) => n.(elem.s) <= n.(next.s).(elem.s)`
		` }`
		`}`
		
		`pred add_pre(l: List, x: Int, s: State) {...}`
		
		`pred add_post(l: List, x: Int, s, s': State) {`
		` RepOk[l, s']`
		` all i: Int {`
		`  i != x =>`
		`  #{n: l.(header.s).*(next.s) | n.(elem.s)=i}`
		`  = #{n: l.(header.s').*(next.s') | n.(elem.s')=i}`
		`  else`
		`  #{n: l.(header.s).*(next.s) | n.(elem.s)=i} + 1`
		`  = #{n: l.(header.s').*(next.s') | n.(elem.s')=i}`
		` }`
		`}`
		
		`pred size_pre(l: List, s: State) {...}`
		
		`pred size_post(l: List, result: Int, s: State) {...}`

④	**Example Test Repair**:
	assertion updated
		(a)
		`@Test public void test() {`
		` List l = new List();`
		` l.add(0);`
		` l.add(0);`
		` assertEquals(1, l.size());`
		`}`
		
		(b)
		`@Test public void test() {`
		` List l = new List();`
		` l.add(0);`
		` l.add(0);`
		` assertEquals(2, l.size());`
		`}`

Fig. 1. Example program evolution & test repair

defines the list class invariant (`RepOk`) and methods `add` and `size`; ③ a new Alloy specification that defines the modified list class invariant and method `add`; and ④ an example test repair performed by SPECTR.

The Java code declares that each list has a `header` node, and each node has an integer `elem` and has a `next` node. The method `size` returns the number of elements in the list. The method `add` inserts a given integer into the list.

The Alloy specification in Fig. 1-② has five *predicates*; each predicate (`pred`) defines a parameterized formula. The predicate `RepOk` states the class invariant and has two parameters: a list `l` and a state `s`. This universally quantified (`all`) formula expresses acyclicity and sortedness of unique elements. The operator '`.`' represents relational join. An expression `o.(f.s)` represents dereferencing of field `f` of object `o` in state `s`. '`*`' represents reflexive transitive closure. For example, `header.*next` denotes all nodes reachable from `header`. '`^`' represents transitive closure. The first sub-formula states directed acyclicity by ensuring that a traversal that starts node `x` cannot revisit the same node. The second sub-formula ensures that the list is sorted and contains no repetitions. The predicate `add_post` states the post-condition of method `add`. States `s` and `s'` represent a pre-state and a post-state after invoking `add` respectively. The first formula states the class invariant holds in the post-state. The second formula states the elements in the list is a union of the elements in the pre-state and an added element `x`. The predicate `size_post` states the post-condition of method `size`. The operator '`#`' denotes the cardinality of a set. The parameter `result` represents the return value. Since `size` is a *pure* method, i.e., the execution of the method does not change the state of its inputs, its predicate does not need a post-state. The predicates `add_pre` and `size_pre` state the pre-conditions of `add` and `size` respectively; both the predicates state that the class invariant holds in the pre-state `s`.

An example JUnit test with respect to this specification is shown in Fig. 1-④-(a). The test allocates a new list and makes two invocations of `add` followed by a correctness check using `assertEquals`. Since the class invariant does not allow repetitions, the assertion checks that the size of the list after the two `add` operations is 1. This test passes only if method `add` correctly avoids repetitions.

To demonstrate SPECTR's test repair process, consider the following modification to the list specification: a list may now contain repeated elements, and must still be acyclic and sorted (Fig. 1-③). Note the comparison operator in `RepOk` is now '`<=`' instead of '`<`'. The post-condition of `add` is updated correspondingly. This modified post-condition states that the number of times each integer other than `x` appears in the list in the pre-state is the same as the number of times that they appear in the post-state, whereas the number of times that `x` appears in the list is increased by 1.

With respect to this modified specification, the JUnit test in Fig. 1-④-(a) is no longer correct, since the list size is expected to be 2 instead of 1. SPECTR transforms the old test to the repaired test shown in Fig. 1-④-(b). The repaired test is now correct with respect to the modified specification in the sense that every test failure now represents a violated specification. We emphasize that

SPECTR does not require method implementations to be correct with respect to the modified specification. Moreover, none of the previous test repair techniques [5, 6, 8, 18] can repair the above test since they use the updated code as opposed to the updated specification as a basis for test repair.

4 SPECTR

SPECTR repairs JUnit tests using method-level specifications written in Alloy. SPECTR takes as input an existing test and the modified specifications of the methods invoked by the test, and corrects the test's expected output. Section 4.1 describes our test repair algorithm. Section 4.2 describes how we leverage the Alloy tool set as an enabling technology for automated test repair.

4.1 Algorithm Overview

Given a set of tests that need to be repaired with respect to the modified specifications, SPECTR repairs the tests one at a time. SPECTR assumes that each test case consists of three components in the following style:

```
@Test public void testcase() {
    // 1. initialization: code to create pre-state (inputs)
    ...
    // 2. execution: code to execute sequence of methods under test
    ...
    // 3. assertion: code to check post-state (output)
}
```

In general, JUnit methods can contain arbitrary Java code and may not follow this structure, e.g., have no assertion to check the output. Such non-conforming tests are not handled by SPECTR. However, our approach can, in principle, leverage the JForge framework [7] to handle more general JUnit tests, including those with loops, conditional statements, or even multiple test assertions.

Fig. 2 describes our test repair algorithm. Given an old test to repair (`oldTest`) and a modified specification (`newSpec`), it returns a repaired test (`newTest`) conforming to `newSpec`. The resulting repaired test must have the same initialization and method execution code as the old version followed by *updated assertion checks*, conforming to modified specifications.

Identification of Expected Test Behavior. SPECTR emulates the execution of a test case by executing corresponding modified specifications using Alloy. The execution of a JUnit test essentially makes several state transitions starting from the initial state and checks certain properties at certain states. Given an initial state, a sequence of method invocations, and the specifications of invoked methods, the Alloy Analyzer generates the pre-states and post-states of those method invocations to identify the expected behavior of the test.

Consider a sequence of method invocations and state transitions in a test: $\langle\sigma_0\rangle m_1()$; $\langle\sigma_1\rangle m_2()$;$\langle\sigma_2\rangle$ ...; $\langle\sigma_{k-1}\rangle m_k()$; $\langle\sigma_k\rangle$. m_1 is invoked on a pre-state σ_0 (initial state abstracted from test initialization code). If σ_0 satisfies m_1's pre-condition, its expected post-state σ_1 is generated by the Alloy Analyzer using the pre-state σ_0 and m_1's post-condition. For the invocation of m_i, where $1 < i \leq k$, the post-state of m_{i-1}, σ_{i-1} is the pre-state for m_i. Assuming σ_{i-1} satisfies the

```
1TestCase repair(TestCase oldTest, Spec newSpec) {
2    // extract the three elements of the given test
3    Code testInit = oldTest.getTestInitCode();
4    Code methodExec = oldTest.getMethodExecution();
5    AssertEquals assertion = oldTest.getAssertEquals();
6
7    TestCase newTest = new Test(); // output
8    newTest.append(testInit);
9    newTest.append(methodExec);
10
11   // compute expected post-state
12   Instance post; // expected post-state w.r.t spec
13   Instance pre = abstract(Java.execute(testInit));
14   post = Alloy.solve(createModel(pre, methodExec, newSpec));
15
16   // synthesize new correctness check(s)
17   Expression actual = assertion.getActual();
18   newTest.append(new AssertEquals(
19     concretize(Alloy.solve(createModel(post, actual, newSpec))), actual);
20   return newTest;
21}
```

Fig. 2. Test repair algorithm

pre-condition of m_i, the Alloy Analyzer computes a corresponding post-state σ_i based on the post-condition of m_i. If any method's pre-condition is not satisfied by the method invocation's pre-state, it means that the inputs of the method invocation do not meet a pre-condition, and thus the test is broken and cannot be repaired. These tests need to be removed from the test suite.

Replacement of Expected Values in JUnit Assertions. JUnit provides several assert methods to write correctness properties, which can be de-sugared into the `assertEquals` method. Each test is repaired by using the post-state Alloy instance after the invocation of the sequence of methods under test to compute the expected value for the assertion check.

4.2 Using Alloy for Test Repair

The initialization code of the old test is used to generate the pre-state Alloy instance using an abstraction translation [16], which traverses the Java data structures and initializes a corresponding Alloy instance.

Each method of a class has its corresponding specification, i.e., a pre-condition and a post-condition. Consider a method `m` in class `C`:

```
class C{T m (T1 p1, T2 p2, ..., Tk pk){...}}
```

The Alloy pre-condition for `m` has the following declaration:

```
pred m_pre(c:C, p1:T1, p2:T2, ..., pk:Tk, s_pre:State){...}
```

If the return type `T` is not void, and method `m` is not a pure method, the post-condition for method m has the following declaration:

```
pred m_post(c:C, p1:T1, p2:T2, ..., pk:Tk, result:T,
     s_pre:State, s_post:State){...}
```

If the return type `T` is void, then the parameter `result:T` in `m_post` does not exist. If method `m` is a pure method, the parameter `s_post:State` does not exist, since the values other than the return value don't change between the pre-state and post-state of the method invocation.

If the method `m` is static, for both pre-condition and post-condition, the parameter `c:C` does not exist in the parameter list.

To illustrate, in our running example (Section 3), the class `List` has methods `add` and `size`. The method `add` is not pure, its return type is void, and it has an `int` type parameter; while the method `size` is pure, its return type is `int`, and it has no parameters. Their pre-conditions and post-conditions have the following declarations:

```
pred add_pre(l:List, p1:Int, s_pre:State){...}
pred add_post(l:List, p1:Int, s_pre:State, s_post:State){...}
pred size_pre(l:List, s_pre:State){...}
pred size_post(l:List, result:Int, s_pre:State){...}
```

Alloy directly supports primitive integers. Support for other primitive types can be provided through Alloy libraries, e.g., the standard Alloy library includes a model for Boolean.

Given the specifications for each method, the method invocations are translated to an Alloy model using four steps:

1. Model the receiver object and include it as the first parameter for the pre/post-conditions;
2. Model the formal parameters and append them to the parameter list for the pre/post-conditions;
3. For post-condition specification, if there's a return value, create an Alloy signature with the corresponding return type, and append it to the parameter list; and
4. Append the current state to the parameter list. If the list is for the post-condition and the method is not pure, create a new Alloy State and append it to the parameter list, and update the current state to the newly created one.

For example, consider the following method invocation in state `S1`:

```
l.add(2);
```

This invocation is translated to the following Alloy code:

```
add_pre[l, 2, S1]
add_post[1, 2, S1, S2]
```

and the current state is updated to `S2`.

Given the specifications for methods invoked, and an initial state abstracted from the execution result of test initialization code, the Alloy Analyzer checks the satisfiability of each method's pre-condition before the invocation of the method, and generates a post-state using the pre-state and post-condition. For example, for the test shown in Fig. 1-④-(a), SPECTR generates the following Alloy code to check whether the pre-state of the first `add` method invocation (the initial state `S0`) satisfies the method's pre-condition.

```
one sig S0 extends State {}
one sig l extends List {}
fact {
 no l.(header.S0)
 add_pre[l, 0, S0]
}

pred test() {}
run test
```

If the Alloy Analyzer finds no solution, which means that the pre-condition is not satisfied, SPECTR reports to the users that the inputs in the test are not as expected and that the test cannot be repaired and should be removed from the test suite; otherwise, the pre-condition is satisfied, and SPECTR can generate an expected post-state of the method invocation by constructing the following Alloy code and solving it with the Alloy Analyzer.

```
one sig S0, S1 extends State {}
one sig l extends List {}
fact {
 no l.(header.S0)
 add_pre[l, 0, S0] && add_post[l, 0, S0, S1]
}

pred test() {}
run test
```

The post-state of the method invocation, which is the Alloy instance at `S1`, is generated using the pre-state, which is the Alloy instance at `S0`, and the method's post-condition `add_post`. Similarly, all pre-states of other method invocations can be checked, and all post-states of those invocations can be generated. Thus, each method invocation triggers a state transition from a state to its next state. Except for the initial state `S0`, all other states are expected states resulting from reasoning on `S0` and specifications.

The Alloy instance at the state where the assertion is to be checked is used to compute the expected value. For the actual expression in the `assertEquals` method, SPECTR uses its corresponding value in the Alloy instance as expected value and replaces the old value with it for the updated test.

For the test example shown in Fig. 1-④-(a), SPECTR generates the following Alloy model and solves it using the Alloy Analyzer.

```
one sig S0, S1, S2 extends State {}
one sig l extends List {}
one sig Result {val: Int}
fact {
 no l.(header.S0)
 add_pre[l, 0, S0] && add_post[l, 0, S0, S1]
 add_pre[l, 0, S1] && add_post[l, 0, S1, S2]
 size_pre[l, S2] && size_post[l, Result.val, S2]
}

pred test() {}
run test
```

Given this Alloy model and a scope, the Alloy Analyzer finds an instance that shows at `S2 Result.val` is 2, which is the expected value with respect to the modified specification. SPECTR then replaces the expected value of the assertion with 2 to repair the test.

Our current SPECTR prototype repairs tests by updating primitive values. A more comprehensive tool would allow updating more complex data structures, which can be achieved by concretizing an output from SAT and using the `equals` method for checking the validity of the output from the program under test.

Table 1. Evolution Scenarios

Sce.	Subject	Old Spec	Modified Spec	Test Method Executions	Assertion Method
#1	Sorted Singly-Linked List	The comparison among list elements is "<"	The comparison among list elements is "<="	add(0), remove(0), add(1), remove(1)	size()
#2	Binary Heap	Min heap	Max heap	insert(0), insert(1), insert(2), insert(3)	peek()
#3	java.util. LinkedList	Method add(E e) appends e to the end of the list	Method add(E e) inserts e at the beginning of the list	add(0), add(1), add(2), add(3)	getFirst()
#4	java.util. TreeSet	All integer values are allowed in the set	Only positive integer values are allowed in the set	add(-1), add(0), add(1), add(2)	add(E e)

5 Experiments

This section describes experiments to evaluate test repair performed by our prototype implementation of Spectr. The goal of our study is to demonstrate Spectr's ability to repair tests using modified specifications for structurally complex subjects and to demonstrate its feasibility for repairing test suites with a few hundred tests.

5.1 Evolution Scenarios

Table 1 shows the four evolution scenarios used in our study. Each row in the table lists the subjects, specification changes, the methods under test, and the methods used in correctness check. Those subject programs have been previously used to evaluate various approaches in testing and verification [7, 9]. **Sorted singly-linked list** represents sorted acyclic lists as described in Section 3. **Binary heap** is a heap data structure based on a binary tree. The tree is a complete binary tree. Heaps can be of two kinds: max-heap and min-heap. In a max-heap, each node is greater than or equal to each of its children. In a min-heap, each node is less than or equal to each of its children. The subjects **java.util.LinkedList** and **java.util.TreeSet** are from the standard Java libraries. The implementations of the subjects remain the same during evolutions. Having the old specifications is not necessary for applying Spectr; however, if we also have access to the old specifications, we can reduce the number of tests we attempt to repair by identifying a subset of tests that invoke methods with modified specifications.

Test case generation using Java PathFinder. Spectr assumes a regression test suite exists—developers may have already written test cases for the old program version or generated them using an automated test generation tool. In our evaluation study, we use the Java PathFinder (JPF) model checker [22] to automatically generate a test suite for the old program version following a variant of an earlier approach [23].

We use JPF's non-deterministic choice operator to enumerate JUnit tests, where each test starts with a default constructor call, executes methods under test, and checks a correctness property. Note that the correctness check in each test reflects the actual behavior of the old version, but not the expected behavior

Table 2. Test repair by SPECTR.

Sce.	Old Tests	Affected Tests	Sucessfully Repaired	Modified Tests	Unchanged Tests	Total (seconds)	Average (seconds)
#1	340	100%	100%	112	228	38	0.11
#2	340	100%	100%	340	0	53	0.16
#3	340	100%	100%	252	88	15	0.04
#4	340	100%	100%	99	241	12	0.03

Table 3. Test repair by ReAssert.

Sce.	Old Tests	Passing Tests	Failing Tests	Repaired Tests
#1	340	340	0	0
#2	340	340	0	0
#3	340	340	0	0
#4	340	340	0	0

according to a given specification. Fig. 4 in Appendix A shows an example test generator for singly-linked lists.

This JPF-based driver generates 340 tests in total for each data structure: 4 tests with one method execution, 16 tests (4*4) with two method executions, 64 tests (4*4*4) with three method executions, and 256 tests (4*4*4*4) with four method executions. Repetition is allowed in each sequence of method execution. Tests for `TreeSet` execute the last add(E e) in the sequence of method executions in the test assertion.

5.2 Test Repair Results

We compiled all our subject programs and JUnit tests using Java version 6 and JUnit 4.4. We used the Alloy Analyzer version 4 as a back-end for solving Alloy specifications. The study was performed on a Dell Desktop running at 2.8 GHz Intel Core i7 CPU with 8 GB of memory and running Windows 7 Professional.

Table 2 shows SPECTR's repairing results. `Old Tests` column indicates the number of tests in the old test suite, and `Affected Tests` column shows the percentage of tests, which invoke some method with a specification change. Column `Successfully Repaired` shows the percentage of tests successfully repaired out of all affected tests. Column `Modified Tests` indicates the number of tests that are modified after the repair, while column `Unchanged Tests` shows the number of tests that are not changed by SPECTR. Column `Total` is the total time taken to repair all the tests, while column `Average` is the average time taken for a single test repair.

For all program evolutions considered in this study, all tests are affected, and SPECTR successfully repaired all of them. However, the number of modified tests, unchanged tests, and the time cost vary for different evolution scenarios. For instance, all tests were modified in scenario #2, while only 112 tests, less than one third of the total, were modified in scenario #1. Moreover, 53 seconds were spent on repairing the 340 tests in scenario #2, while only 12 seconds were spent on repairing the same number of tests in scenario #4. The cost of repair depends on the complexity of the modified specification and the length of the test execution.

```
@Test public void testcase47() {
    LinkedList l = new LinkedList();
    l.add(1);
    l.add(2);
    l.add(3);
 - assertEquals(1, l.getFirst());
 + assertEquals(3, l.getFirst());
}
```

(a) Modified Test

```
@Test public void testcase311() {
    LinkedList l = new LinkedList();
    l.add(3);
    l.add(2);
    l.add(0);
    l.add(3);
    assertEquals(3, l.getFirst());
}
```

(b) Unchanged Test

Fig. 3. Two example test repairs performed by SPECTR.

Note that a test repair technique that does not take into account specifications and is driven purely by implementation would not repair any of the old tests. We applied ReAssert [6], a recent test repair technique, in these four scenarios. ReAssert did not repair or modify any tests since all tests passed and ReAssert only repairs failing tests (Table 3).

To validate repairs made by SPECTR, we manually inspected all repaired tests and found that all of them correctly reflect the modified specifications.

Fig. 3-(a) shows an example of the repair done by SPECTR for scenario #3. In the modified specification, `add(E e)` inserts `e` at the beginning instead of appending `e` to the end, thus the expected result of `l.getFirst()` in `testcase47` is modified from 1 to 3 to reflect the modified specification.

Note that some tests remain unchanged after repair, since the test inputs result in the same outputs according to the old specification as well as the modified specification. Fig. 3-(b) shows such a case for scenario #3, where the first element and the last element added to the list are the same.

We ran the repaired tests against the implementation, and found that all the *modified* tests failed. Those failing tests reflect the errors in implementations which have not yet undergone modifications.

Our study demonstrates that for the subject programs and the selected types of evolution used in the study, SPECTR effectively repairs existing tests to reflect the modified specifications. SPECTR automatically updates the expected test outcomes. *The cost of test repair using* SPECTR *is reasonable, with a range from* 0.03 *to* 0.16 *seconds per test.* Our SPECTR prototype is not optimized, e.g., it uses several file-I/O operations to read each old test and write each modified test. We plan to optimize SPECTR is future work.

6 Conclusions and Future Work

This paper presents SPECTR, a novel specification-based technique for test repair. Given behavioral specifications for the modified program and an existing test suite, SPECTR repairs each test that exercises modified behaviors. It leverages the existing test inputs and updates the test assertions to reflect the modified specification.

The experiments conducted on a suite of subject programs with modified specifications show that SPECTR can effectively repair tests with respect to modified specifications. Moreover, SPECTR is efficient in terms of test repair performance, and the time spent on each repair is less than a half second on average for the subject programs used in our experiments.

SPECTR leverages the Alloy tool-set as an enabling technology and hence requires the use of first-order logic and SAT. While properties of a diverse class of programs can conveniently be expressed in Alloy and checked using SAT, for some programs, e.g., those that perform complex numeric calculations, effective test repair would need an alternative enabling technology. However, our basic approach for test repair would still be applicable, for example, to enable the Pex framework [20] to repair C# tests using Spec# specifications.

As future work, we plan to conduct more extensive evaluation of SPECTR, especially using more complex subjects, such as open source programs.

Acknowledgments. This work is supported in part by NSF grants under CCF-0845628, CCF-1043810, CCF-1117902, and CCF-1149391, AFOSR grant FA9550-09-1-0351, and 2011 Microsoft SEIF Award.

References

1. JUnit website. http://www.junit.org.
2. L. Chen and A. Avizienis. N-version programming : a fault-tolerance approach to reliability. In *FTCS-8*, pages 3–9, 1978.
3. Y.-F. Chen, D. S. Rosenblum, and K.-P. Vo. Testtube: a system for selective regression testing. In *ICSE*, pages 211–220, 1994.
4. H. Cleve and A. Zeller. Locating causes of program failures. In *ICSE*, pages 342–351, 2005.
5. B. Daniel, T. Gvero, and D. Marinov. On test repair using symbolic execution. In *ISSTA*, pages 207–218, 2010.
6. B. Daniel, V. Jagannath, D. Dig, and D. Marinov. Reassert: Suggesting repairs for broken unit tests. In *ASE*, pages 433–444, 2009.
7. G. Dennis, F. S.-H. Chang, and D. Jackson. Modular verification of code with SAT. In *ISSTA*, pages 109–120, 2006.
8. A. V. Deursen, L. Moonen, A. Bergh, and G. Kok. Refactoring test code. In *XP2001*, pages 92–95, 2001.
9. J. P. Galeotti, N. Rosner, C. G. López Pombo, and M. F. Frias. Analysis of invariants for efficient bounded verification. In *ISSTA*, pages 25–36, 2010.
10. D. Gopinath, M. Z. Malik, and S. Khurshid. Specification-based program repair using SAT. In *TACAS*, pages 173–188, 2011.
11. M. Grechanik, Q. Xie, and C. Fu. Maintaining and evolving GUI-directed test scripts. In *ICSE*, pages 408–418, 2009.
12. S. Hangal and M. S. Lam. Tracking down software bugs using automatic anomaly detection. In *ICSE*, pages 291–301, 2002.
13. D. Jackson. *Software Abstractions: Logic, Language, and Analysis*. The MIT Press, 2006.
14. D. Jeffrey, M. Feng, N. Gupta, and R. Gupta. Bugfix: A learning-based tool to assist developers in fixing bugs. In *ICPC*, pages 70–79, 2009.
15. L. Jiang and Z. Su. Context-aware statistical debugging: from bug predictors to faulty control flow paths. In *ASE*, pages 184–193, 2007.
16. S. Khurshid and D. Marinov. Testera: Specification-based testing of Java programs using SAT. *ASE*, 11(4):403–434, 2004.
17. A. M. Memon. Automatically repairing event sequence-based GUI test suites for regression testing. *TOSEM*, 18(2):1–36, 2008.

18. M. Mirzaaghaei, F. Pastore, and M. Pezze. Automatically repairing test cases for evolving method declarations. In *ICSM*, pages 1–5, 2010.
19. G. Rothermel and M. J. Harrold. A safe, efficient regression test selection technique. *TOSEM*, 6(2):173–210, 1997.
20. N. Tillmann and J. De Halleux. Pex: white box test generation for .net. In *TAP*, pages 134–153, 2008.
21. M. Vaziri. *Finding Bugs Using a Constraint Solver*. PhD thesis, CSAIL, MIT, 2003.
22. W. Visser, K. Havelund, G. P. Brat, S. Park, and F. Lerda. Model checking programs. *ASE*, 10(2):203–232, 2003.
23. W. Visser, C. S. Păsăreanu, and S. Khurshid. Test input generation with Java Pathfinder. In *ISSTA*, pages 97–107, 2004.
24. W. Weimer, T. Nguyen, C. Le Goues, and S. Forrest. Automatically finding patches using genetic programming. In *ICSE*, pages 364–374, 2009.
25. R. N. Zaeem and S. Khurshid. Contract-based data structure repair using Alloy. In *ECOOP*, pages 577–598, 2010.
26. A. Zaidman, B. V. Rompaey, A. van Deursen, and S. Demeyer. Studying the co-evolution of production and test code in open source and industrial developer test processes through repository mining. *ESE*, 16(3):325–364, 2011.
27. A. Zeller. Automated debugging: Are we close? *Computer*, 34:26–31, 2001.

A JPF-based test generator

Fig. 4 shows an example test generator, which we use for generating singly-linked lists using Java PathFinder in our experiments (Section 5).

```
1 static void testGenerator() {
2   Verify.resetCounter(0); // test ID
3   final int SEQ_LENGTH = Verify.getInt(1, 4);
4   StringBuilder tc = new StringBuilder(); // test case
5   tc.append("    List l = new List();\n");
6   List l = new List();
7   for (int i = 0; i < SEQ_LENGTH; i++) {
8     int arg = Verify.getInt(0, 1);
9     if (Verify.getBoolean()) {
10      tc.append("    l.add(" + arg + ");\n");
11      l.add(arg);
12    } else {
13      tc.append("    l.remove(" + arg + ");\n");
14      l.remove(arg);
15    }
16  }
17  int expected = l.size();
18  tc.append("    assertEquals(" + expected + ", l.size());\n" + "}");
19  tc.insert(0, "@Test public void testcase" + Verify.getCounter(0) + "() {\n");
20  System.out.println(tc + "\n");
21  Verify.incrementCounter(0);
22 }
```

Fig. 4. JPF-based test generator. It generates tests that represent all possible sequences involving one to four method executions on a list `l`: `l.add(0)`, `l.add(1)`, `l.remove(0)`, and `l.remove(1)`. Line 3 non-deterministically chooses the length of the sequence between 1 to 4. The for-loop from line 7 to 16 non-deterministically chooses one of the four possible method executions. In addition to generating method sequences, JPF also runs them on the old program implementation (Lines 6, 11, 14, and 17) and computes the value of the expressions (`l.size()`) in assertion checks.