Constraint-Based Program Debugging Using Data Structure Repair

Muhammad Zubair Malik Junaid Haroon Siddiqui Sarfraz Khurshid
Department of Electrical and Computer Engineering
The University of Texas at Austin
Austin, TX 78712
Email: {mzmalik,jsiddiqui,khurshid}@ece.utexas.edu

***Abstract*—Developers have used data structure repair over the last few decades as an effective means to recover on-the-fly from errors in program state. Traditional repair techniques were based on dedicated repair routines, whereas more recent techniques have used invariants that describe desired structural properties as the basis for repair. All repair techniques are designed with one primary goal: run-time error recovery. However, the actions that any such technique performs to repair an erroneous program state are meant to produce the effect of the actions of a (hypothetical) correct program. The key insight in this paper is that repair actions on the program state can guide debugging of code (when the erroneous program execution is due to a fault in the program and not an external event).**

This paper presents an approach that abstracts concrete repair actions that a routine performs to repair an erroneous state into a sequence of program statements that perform the same actions using variables visible in the scope of the faulty code. Thus, appending the generated statements to the original code is akin to performing the repair from within the program. Our implementation uses the Juzi data structure repair tool as an enabling technology. Experimental results using a library data structure as well as two applications demonstrate the effectiveness of our approach in enabling repair of faulty code

I. Introduction

As software pervades our societies and lives, the need for software reliability grows more and more. With computers flying planes and running financial markets, the cost of even a single software failure can be quite high. Failures already cost the Nation's economy $59.5 billion every year [1]. To meet the ever-increasing demand for reliability, we must create methodologies that deliver more reliable software at a lower cost.

Traditional methodologies for increasing software reliability fall in two basic categories: compile-time checking, e.g., using software testing [2], [3] and debugging [4], model checking [5], or static analysis [6], and runtime monitoring or error recovery [7], [8], e.g., using data structure repair [9], [10].

This paper presents a novel methodology based on a runtime technique — data structure repair — and a compile-time technique — software testing and debugging — which results in a synergy that can substantially increase software reliability as well as decrease the cost of software development.

Our insight is that since the goal of repair is to transform an erroneous state into an acceptable state, the state mutations performed by repair provide a basis of debugging faults in code (assuming the erroneous states are due to bugs and not external events, say cosmic radiation). A key challenge to embodying our insight into a mechanical technique arises due to the difference in the concrete level where the program states exist and the abstract level where the program code exists: repair actions apply to concrete data structures that exist at runtime and have a dynamic structure (i.e., may get mutated), whereas debugging applies to code that has a static structure.

Given a Java method that takes as input structurally complex data, the structural invariants that the method must preserve, and an input that leads to an invariant violation by the method, our technique performs three basic steps. (1) It uses data structure repair to transform the erroneous output into a program state that satisfies the structural invariants. (2) It abstracts the set of concrete repair actions by generating a sequence of Java statements using variables visible within the scope of the method. (3) It determines (heuristically) where in the method the generated sequence fits.

As an enabling technology, we use the Juzi framework for data structure repair [10]–[13]. To increase our confidence in the correctness of the generated code, we use the Korat framework for systematic testing [14], [15]. Juzi and Korat use the same structural invariants for their analysis that our methodology uses for generating debugging suggestions. When these invariants already exist, say because of systematic testing, debugging using our technique comes for free.

The algorithms developed in this paper are inspired by our previous study [16] of possible faults in Java Collection `LinkedList`. However, that work only described the various types of faults and conjectured the existence of possible algorithms that may leverage concrete repair actions to generate program repairs but did not provide any such algorithms to automatically fix the faults. This paper presents the algorithms and evaluates their implementation. We make the following contributions:

- **Abstract repair.** We introduce the idea of abstracting concrete repair actions into a sequence of program statements that represent those actions;
- **Repair for debugging assistance.** We introduce the idea of using concrete repair actions for debugging assistance and show how our methodology assists with locating faults and fixing them;
- **Algorithms.** We present two algorithms that form the basis of our methodology: one algorithm performs the abstraction of concrete repair actions and the other algorithm uses abstract repair to generate debugging suggestions.
- **Evaluation.** We evaluate our approach for its success rate on faulty mutants of many programs and demonstrate its effectiveness on Java library code as well as two large scale open source applications: ANTLR and RayTracer. Experimental results show our approach generates highly accurate debugging suggestions.

II. Example

Consider the following declaration of a class implementing doubly-linked circular lists based on the `java.util.LinkedList` class from Java libraries:

```
 1. public class LinkedList {
 2.     private Entry header=new Entry(null, null, null);
 3.     private int size = 0;
 4.
 5.     public LinkedList() {
 6.         header.next = header.previous = header;
 7.     }
 8.
 9.     private static class Entry {
10.         Object element;
11.         Entry next;
12.         Entry previous;
13.
14.         Entry(Object element, Entry next,
15.               Entry previous) {
16.             this.element = element;
17.             this.next = next;
18.             this.previous = previous;
19.         }
20.     }
```

Each list object has a sentinel `header` node and caches the number of nodes in the field `size`. The class `Entry` implements the list nodes. Each node has an `element`, and `next` and `previous` pointers to other nodes. Figure 1 (a) illustrates an empty list.

The structural invariants (called *class invariants* in object-oriented programs) of doubly-linked lists are sentinel `header` nodes, circularity along `next` fields, transpose relation between `next` and `previous` fields, and correct value for `size`. Any valid list must satisfy these invariants (in all publicly visible states). These invariants can be represented using a `repOk` [17] method that traverses its input structure and returns true if and only if the input satisfies all the invariants:

```
21.     public boolean repOk() {
22.         if (header == null) return false;
23.         if (header.element != null) return false;
```

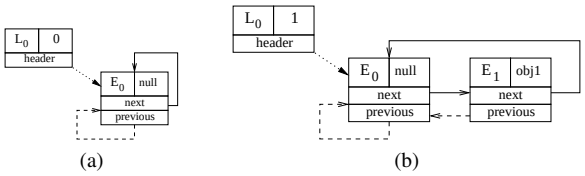

Figure 1. Doubly-linked circular list with sentinel `header`. (a) An empty list (size 0). (b) An erroneous list of size 1 containing element 0. A small box-pair represents a list object and is labeled with the object's identity and the value of its `size`. Large box-pairs represent entry objects and are labeled with object identity and value of `element`.

```
24.         Set visited = new HashSet();
25.         visited.add(header);
26.         Entry current = header;
27.         while (true) {
28.             Entry next = current.next;
29.             if (next == null) return false;
30.             if (next.previous!=current) return false;
31.             current = next;
32.             if (!visited.add(next)) break;
33.         }
34.         if (current != header) return false;
35.         if (visited.size() - 1 != size) return false;
36.         return true;
37.     }
```

Class invariants implicitly form a part of the preconditions and postconditions of public methods. Thus, all executions of a public method are expected to terminate in a state where the class invariants hold.

Consider the following implementation of the method `addFirst`, which uses the (erroneous) helper method `addBefore`:

```
38.     public void addFirst(Object e) {
39.         addBefore(e, header.next);
40.     }
41.
42.     private Entry addBefore(Object e, Entry entry) {
43.         Entry newEntry =
44.                 new Entry(e, entry, entry.previous);
45.         newEntry.previous.next = newEntry;
46.         newEntry.next.previous = entry; // fault
47.         size++;
48.         return newEntry;
49.     }
```

The method `addBefore` has a fault in its third assignment statement (Line 46), which erroneously sets a `previous` field to `entry` instead of `newEntry` (as correctly implemented by `addBefore` method of `java.util.LinkedList`). To illustrate the effect of this fault, consider the following code snippet:

```
50.         LinkedList l = new LinkedList();
51.         assert l.repOk(); // pass
52.         l.addFirst(0);
53.         assert l.repOk(); // fail
```

The second assertion (Line 53) fails. Figure 1 (b) illustrates the erroneous list in the post-state of `addFirst`. The `previous` field of the `header` node (E_0) is erroneously set to the node itself (instead of E_1).

Given the erroneous list and the `repOk` method, Juzi — a data structure repair tool discussed in Section III — repairs

the list by performing the following repair action: $\langle E_0$, previous, $E_1\rangle$, i.e., by setting the previous field of E_0 to E_1, thereby generating a valid list of size 1 containing the element 0—the list a correct implementation of addBefore would generate.

Based on Juzi's concrete repair action, our repair abstraction algorithm using conservative reasoning finds the correct mapping from E_0 to newEntry.next and E_1 to newEntry that enables it to generate the following Java code:

```
newEntry.next.previous = newEntry;
```

Our debugging advisor algorithm suggests this fix for Line 46—this is exactly how the Java libraries (correctly) implement addBefore.

III. Background: Juzi

Juzi is an automated framework for on-the-fly repair of data structures [10]–[12], [18]. Given a corrupt data structure, as well as a repOk method that describes the structural integrity constraints, Juzi systematically mutates the fields of the corrupt data structure so that it satisfies the given constraints. In addition to repairing the given structure, Juzi reports the *repair actions* it performed on the corrupt structure in a log-file that holds a sequence of tuples $\langle o$, f, $o'\rangle$, i.e., an assignment to field f of object o the value o'—each tuple represents a repair action.

To illustrate Juzi and its repair mechanism, consider the example of repairing corrupt doubly linked lists (Section II). Consider the list in Figure 1 (b). The list has one corruption in the previous field on node E_0. Given the corrupt structure and the repOk method (Section II), Juzi first invokes repOk on the corrupt structure and monitors the fields accessed by repOk during its execution. When repOk returns false due to a constraint violation, Juzi systematically mutates the *last* field accessed by repOk by non-deterministically setting it to : (1) null, (2) nodes that have already been visited during repOk's execution, and (3) one node that has not yet been visited.

To illustrate, monitoring the execution of repOk, Juzi detects the fault in the previous field of node E_0, and mutates its value first to null, which does not repair the fault, and then to node(s) that have been previously encountered during the execution of repOk. Since E_0 is the original value of the field, Juzi does not need to try it again (unless some other fields are modified first). Therefore, Juzi tries node E_1 next, which repairs the fault in the structure.

In addition to repairing the corrupt structure, Juzi reports the tuple $\langle E_0$, previous, $E_1\rangle$ to indicate the repair action that fixed the corruption. Note that although Juzi tries several mutations on corrupt fields, only the repair actions that result in repairing the fields are reported.

To provide more effective repair, Juzi tries to preserve the reachability of the data in the given structure. In particular, if a sequence of repair actions generates a valid structure that has fewer data than the original structure, Juzi performs further repair actions to preserve the reachability if possible.

The next section describes how to translate these repair actions into code statements that can be used as effective suggestions for debugging faulty code.

IV. Algorithms

Figure 2 illustrates key components of our technique. Given an erroneous output of a faulty method and the structural invariants (repOk) expected of a correct output, data structure repair generates concrete repair actions, i.e., a sequence of field mutations, that repair the corrupt structure. The repair abstraction algorithm takes as input the faulty method, the valid input for which the method gives the erroneous output, and the concrete repair actions of the data structure repair routine, and generates abstract repair code that represents the concrete actions using a sequence of Java statements. The debugging advisor determines (heuristically) where the abstract repair code provides a bug fix in the faulty method and generates a repaired method which has been validated over bounded exhaustive test cases generated by Korat [14].

A. Repair Abstraction Algorithm

This algorithm abstracts concrete repairs suggested by Juzi into the actual program code. Figure 3 presents our repair abstraction algorithm. Given a faulty method, an input state of the program, and a list of repair actions along with output state suggested by Juzi, this algorithms performs the following actions:

- Initialize the code handles to correct value
- For all Juzi repair actions
  - Map objects in Juzi repair action to code handles using conservative reasoning, and translate repair action to build a code statement
  - Update the code and apply repair action on the program state
  - Update the handles to reflect the changes in program state

A key auxiliary data structure the algorithm maintains for every control point during a method execution is a map, Map<Variable, Object>, from statically declared variables that are visible at the control point (including the input parameters, such as this) to their values at that control point for the current execution. This map is similar to *Interesting Value Mapping Pair (IVMP)* used by Jeffrey et al. [19] for fault localization by value replacement, but we use it for reachability analysis.

To illustrate, recall the methods addFirst and addBefore from Section II. Consider executing the method addFirst on the empty input list shown in Figure 1 (a). The map at Line 46 for this execution is:

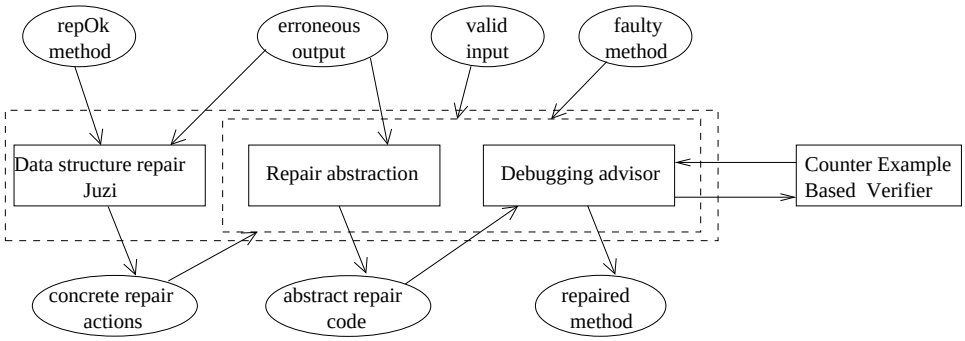

Figure 2. Automated program repair using data structure repair. Solid boxes represent computation modules, and ovals represent data. The dashed box represents the two algorithms that embody our technique.

```
Vector<AssignmentStatement> repairAbstraction(Vector<RepairAction> ras,
                                              Method faulty, Object input, Object output) {
    Vector<AssignmentStatement> abstractRepair = new Vector<AssignmentStatement>();
    Map<Variable, Object> variableValueMap = buildVariableValueMap(faulty, input);
    for (RepairAction ra: ras) {
        Expression source = abstraction(ra.source(), variableValueMap, output);
        Expression target = abstraction(ra.target(), variableValueMap, output);
        abstractRepair.add(new AssignmentStatement(source, ra.field(), target));
        performRepairAction(output, ra);
        updateVariableValueMap(variableValueMap, output);
    }
    return abstractRepair;
}
```

Figure 3. Repair abstraction algorithm.

variable	value
`newEntry`	E_1
`this`	L_0
`e`	0
`entry`	E_0

Note the map is with respect to the variables that are visible in the context of the method that contains the control point. Thus, if a method invokes a helper method, the map is updated to reflect the invocation.

The method `buildVariableValueMap` initializes the map with respect to the last control point that performs a mutation on an object field of the method input when the faulty method is executed on that input. The state of the map reflects that last mutation. Therefore, if the input method calls a helper method that performs all the mutations, the map is built with respect to the variables of the helper method, and the abstract repair code applies to the helper method.

The method `abstraction` has the signature:

```
Expression abstraction(Object o,
                       Map<Variable, Object> v,
                       Object root) { ... }
```

It outputs a path expression that starts traversal at an object pointed to by a variable in the variable-value map, and terminates at the desired object `o`, which is reachable from `root` along some sequence of field dereferences. The output path expression is *loop-free*, i.e., it does not include a subsequence of field dereferences starting at an object and evaluating to the same object. More formally, if for variable v, and fields $f_1, \ldots, f_k$ ($k \geq 1$), $v.f_1.\cdots.f_k$ evaluates to v, the generated expression will not take the form $e.f_1.\cdots.f_k.e'$ for any expression e that evaluates to v, rather it will take the form $v.e'$. This allows `abstraction` to consider a bounded number of path expressions. Moreover, an expression that is not loop-free is likely to represent a programming error.

Among the set of loop-free path expressions that provide the desired handle, `abstraction` prioritizes expressions that start with a local variable declared by the method, since methods that manipulate input object graphs often use local variables as pointers into the input graphs for traversing them and accessing their desired components.

To illustrate, consider the repair action $\langle E_0$, `previous`, $E_1\rangle$ (Section II). Applying `abstraction` to the action's source object E_0 using the variable-value map at the method exit point generates two loop-free expressions "`newEntry.next`" and "`entry`"—both expressions evaluate to E_0. Since priority is given to local variables, `abstraction` outputs the expression "`newEntry.next`".

The method `performRepairAction` updates the object graph that is reachable from a given root object with respect

to the given repair action by mutating the object graph. After that, the method `updateVariableValueMap` modifies the map with respect to the updated object graph.

Correctness. We argue that the repair abstraction algorithm generates a sequence of program statements that represent the given sequence of repair actions. In other words, appending the generated code at the tail of the current execution path (just before the `return` statement) in the control-flow graph results in a modified program that (1) compiles and (2) when executed on the original input, generates the repaired output (up to isomorphism [14]).

Central to our correctness argument is a property of the Juzi data structure repair framework. Juzi performs a systematic search of a neighborhood of the given corrupt structure using backtracking. The basis of the search is an iterative process for mutating object fields and re-executing `repOk` after each mutation to check the validity of the resulting structure. Juzi keeps no explicit pointers into the given object graph other than the given root pointer (`this` of `repOk`). Therefore, the final sequence of repair actions, say $r_1, \ldots, r_n$, where $r_i = \langle o_{i,s}, f_i, o_{i,t} \rangle$ for $1 \leq i \leq n$, performed by Juzi is such that for any repair action r_j $(1 \leq j < n)$ and for any repair action r_k $(j < k \leq n)$, the objects $o_{k,s}$ and $o_{k,t}$ are still reachable from the given root pointer. Thus, the method `abstraction` can always generate a legal path. Hence, the generated sequence of assignment statements compiles and each repair action is abstracted into one assignment statement that represents that action. Thus, executing the sequence of statements performs the same mutations in the same order as Juzi. Therefore, the resulting structure is the same (up to isomorphism) as the repaired structure generated by Juzi.

B. Debugging Advisor Algorithm

The abstract repair code can directly serve as a debugging suggestion: append the sequence at the tail of the execution path (just before the `return` statement). While this suggestion is likely to fix the specific erroneous execution, it does so by undoing any erroneous field mutations of the execution—technically, this may qualify as a bug fix, however, the user may have to go through a tedious process of determining what fault each assignment statement is fixing. Ideally, we would like to mechanically determine where the erroneous mutations are located in the faulty code and to replace them with repaired mutations. The debugging advisor algorithm (Figure 4) uses a heuristic approach to provide this functionality.

The algorithm takes as inputs the faulty method (`faulty`), the input (`input`) that exhibits an erroneous output, the sequence of assignment statements (`stats`) that represent repair actions, termed *repair statements*, and the corresponding sequence of concrete repair actions (`ras`). Intuitively, the algorithm determines for each repair statement where to place it in the faulty method. There are two placement possibilities: (1) replace an existing statement with it, or (2) insert it in the execution path as a new statement. The debugging advisor first tries to find an existing statement for replacement, but if it fails to find such a statement, it inserts it as a new statement.

The class `MethodGen` represents mutable method objects. The method `tracePath` builds an explicit representation of the execution path of the faulty method on the given input.

The method `locateLastRelevantAssignment` provides the key functionality of suggesting an effective bug fix: it traverses the execution path to find an assignment statement that assigns the same object field as the repair statement, heuristically treating the original assignment as erroneous. If it does not find such a statement, it returns `null`, and `debuggingAdvisor` simply appends the repair statement to the execution path. Otherwise, the function `debuggingAdvisor` checks the feasibility of swapping the statements subject to the repair actions that have yet to be integrated into the faulty code.

Recall the faulty method `addBefore` (Section II). For the repair statement "`newEntry.next.previous=newEntry;`" and for the corresponding repair action $\langle E_0,$ `previous`, $E_1 \rangle$, `locateLastRelevantAssignment` returns the statement on Line 46, since it assigns to the same object field as the repair action.

The method `checkFixFeasibility` returns true if swapping the variable `last` with `stat` permits the application of remaining repair actions as a sequence of operations at the tail of the (modified) execution path to generate the repaired output (up to isomorphism). If the replacement is determined infeasible, the function `debuggingAdvisor` appends the repair statement to the path.

Correctness. We argue that the debugging advisor generates a method that, for the given input, outputs a structure isomorphic to the repaired structure generated by Juzi. If `debuggingAdvisor` performs no statement replacements, it simply appends the repair statements to the execution path, and hence the correctness argument for `repairAbstraction` establishes the correctness argument for `debuggingAdvisor`. Consider next the case when the `debuggingAdvisor` replaces an existing statement. By construction, such a replacement is only performed if there exists an integration of the remaining repair actions such that the repaired method generates the repaired output. Thus, the replacements are safe with respect to generating the repaired output by the repaired method.

C. Counter Example Based Verifier

To increase confidence in the correctness of the repaired method, our technique allows a direct application of the Korat framework for systematic testing [14], [15] to automatically generate valid inputs and check outputs using `repOk`. Moreover, any bugs discovered by Korat can feedback into

```
Method debuggingAdvisor(Method faulty, Object input, Object repairedOutput,
                        Vector<AssignmentStatement> stats, Vector<RepairAction> ras) {
    MethodGen repairedMethod = new MethodGen(faulty);
    for (int i = 0; i < stats.size(); i++) {
        AssignmentStatement stat = stats.elementAt(i);
        ExecutionPath path = tracePath(repairedMethod, input);
        AssignmentStatement last = locateLastRelevantAssignment(path, input, stat,
                                                                ras.elementAt(i));
        if (last != null &&
            checkFixFeasibility(repairedMethod, last, stat, stats, ras, repairedOutput)) {
            repairedMethod.replace(last, stat);
        } else repairedMethod.append(path, stat);
    }
    return repairedMethod.method();
}
```

Figure 4. Debugging advisor algorithm.

our technique to use it to iteratively debug a faulty program that has multiple faults along different control-flow paths.

The validation by Korat, implemented by method `checkFixFeasibility`, allows us to use a counter example driven refinement of fixes proposed by our algorithm. Korat is a structural constraint solver that can be used to generated non-isomorphic structures in bounded exhaustive fashion. The structures generated by Korat are used to test correctness of the proposed fixes. A proposed fix is only considered valid if it is valid for all the structures generated by Korat. This step is similar to the use of regression tests for verification by [20].

D. Best-First Heuristic

Instead of ranking our results we use best first heuristic to output our results, a new result is only searched if the user is not satisfied with the best fix that our approach suggests. The debugging advisor algorithm has the goal to minimize the changes from the original code. It first tries to fix the program by field redefinitions, then by reordering the statements after which it attempts changes in flow structure. Since the fix generation processes is fully in our control we allow user to make the choice of extending her search.

V. Evaluation

We report experiments and case studies designed to evaluate success rate of our approach and its effectiveness. The success rate is evaluated by combinatorial fault injection up to a limited size in five text book data structures and comparing it with the fixes found by our approach. The applicability and effectiveness is evaluated by case studies on two large scale industrial applications and `LinkedList` class in Java collections library.

Our approach is implemented as a stand-alone command line application, using AST provided by Eclipse JDT library. All experiments were run on an Intel Dual Core 2.8GHz machine with 2GB of RAM.

A. Success Rate

We define the success rate of our approach as the ratio of failures removed by applying the patch generated by our approach to the total failures detected in the original program. The failure is detected by a runtime verification system by comparing the state of the program to its specifications. Once a failure is detected, Juzi is applied to fix the state of the program. If Juzi is able to fix the concrete structure, we attempt to fix the abstract program using our approach. We define *Repairable Success Rate* as the the ratio of failures removed by applying the patch generated by our approach to the failures detected in the original program that are repairable by Juzi.

Combinatorial fault injection is used to generate variants of a program. We consider faults of *omission*, where the programmer forgets to write the necessary code, and faults of *commission*, where the programmer writes incorrect code. The faults our approach corrects are semantic and not syntactic therefore only semantic faults are injected and we rely on Java debugging framework to find syntactic faults. We assume that the starting code is correct, and to inject faults that mimic omission we remove one or more lines of code; the faults of commission are mimicked by altering the order of program statements and by replacing variables by other variables of same type visible in the scope and also replacing all fields by other fields of same type. We call this approach combinatorial fault injection because all combinations of available variables and fields are used to generate faulty variant methods.

Table I shows our results on seven mutator methods from five different classes. All of these classes implement a predicate function `repOk` which is used to verify the state of the class objects. The method chosen for our experiments vary in size and complexity of control structure. Not all program variants result in failure. We consider a variant to be *Faulty* which causes `repOk` to return false when applied on a valid structure. Not all failures result in errors

Table I
SUCCESS RATE EVALUATION

Class	Class LOC	RepOk LOC	Method	Method LOC	Variants	Faulty	Repairable	Fixed	Success Rate	Repairable Succ Rate
Singly Linked List	58	10	addLast	6	7	7	2	2	28 %	100%
Doubly Linked List	317	21	addBefore	5	31	30	28	28	90%	100%
			remove	9	104	96	70	68	71 %	97%
Disjoint Set	182	32	insertFirst	17	179	164	79	79	44%	100%
			remove	16	23	23	10	9	39%	90%
Binary Search Tree	338	26	addIterative	27	12	12	5	3	25%	60%
Linked Priority Queue	250	31	insert	14	6	6	2	2	33%	100%

Table II
CASES OUR TECHNIQUE CAN HANDLE

Code Type	Single Fault	Independent Multiple Faults	Dependent Multiple Faults
Straight Line	√	√	√
With Branches	√	√	
With Loop	√		√

that are repairable. We consider an error *Repairable* if Juzi can find a fix for the structure produced by the program variant. An error is *Fixed* when we can modify the erroneous program into a new program which when applied to any valid structure (in the bounded exhaustive sense) results in valid structures output. *Rate of Success* is the ratio of *Fixed* programs to programs resulting in *Failure*. While *Repairable Rate of Success* is the ratio of *Fixed* programs to the programs that caused a *Repairable* error.

The success rate of our approach is surprisingly high (97% on average) if we consider the fixes generated only for the cases that are repairable. The success rate is directly affected by structural redundancy of data (that guards against reachability violations) and we observe that our approach works best for doubly linked list with the largest ratio of redundant links among all structures in our experiment.

B. Applicability and Effectiveness

To evaluate the applicability and effectiveness of our approach, we perform three case studies on real programs. Our study shows that our approach can work on real programs and can fix interesting program bugs. We illustrate the use of our approach on, (1) ANTLR — a compiler generator, and 2) RayTracer — a program for tracing lights paths through an image.

The findings of our case studies are summarized in Table II. It shows the relation of code complexity and our ability to accurately localize and correct bugs using our automated debugging advisor. The simplest kind of bugs are those that result in a *single fault* in program state because of a *single defect* in the program code. Slightly more complicated bugs are *multiple faults* in program state that are caused by *multiple defects* in code. We call them *independent multiple faults*. Our algorithm tries to fix them iteratively. The hardest bugs are *multiple dependent faults*. These are *multiple faults* that result from a *single defect* in the code. We employ heuristics to solve multiple dependent faults. Our algorithm considers all possible combinations of program control flow (within some bounds) to apply likely fixes. In the following sections, we take each of three case studies and discuss the bugs our approach fixed.

1) ANTLR: (ANother Tool for Language Recognition) is a language tool that generates recognizers, compilers, and translators from grammatical descriptions. Using a formal grammar `ANTLR` automates the construction of language recognizers and generates a program that determines whether sentences conform to that language. With about 5000 monthly downloads, it is one of the most widely used programs that write other programs. By adding code snippets to the grammar, the recognizer becomes a translator or interpreter. ANTLR has 29710 line of code. The heap of ANTLR at run time consists mainly of custom data structures, errors in which have known to cause major bugs in the program. This makes ANTLR an excellent case study for verifying the scalability and usefulness of our approach. To generate lexical analyzer and parser for a given grammar, ANTLR represents the grammar internally in an `n-ary tree` structure. The interface `Tree` declares its public behavior that is implemented by abstract class `BaseTree` which maintains a list of pointers to its children:

```
1. public abstract class BaseTree implements Tree {
2.   protected List children; }
```

The `BaseTree` class is extended by class `CommonTree`that provides a reference for `parent` and adds the `token` payload to the structure. The actual construction of the structure is done by the `TreeAdaptor` class, and traversals are done by `TreeVisitors`. Since this structure is at the core of `ANTLR`, any error can have far reaching impact. All bugs in this structure are considered Major priority bugs in ANTLR (such as bug 15 and 133 in ANTLR version 3 bug repository).

The representation invariants of this structure are acyclicity along Children and transpose relationship between parent and child. Unlike data structures in JAVA Collections, (1) the Tree in ANTLR does not contain a sentinel root and (2) information about the size of structure is not kept within the

structure. The `repOk` for `CommonTree` is:

```
30. public static boolean repOk (Tree root){
31. if (root==null) return true;
32.   Set<Tree> visited = new HashSet<Tree>();
33.   visited.add(root);
34.   LinkedList<Tree> workList = new LinkedList<Tree>();
35.   workList.add(root);
36.   while (!workList.isEmpty()) {
37.     CommonTree current =
            (CommonTree)workList.removeFirst();
38.     if(!visited.add(current))
39.       return false;
40.     for(int i= 0;i<current.children.size();i++){
41.       if(((CommonTree)current.children.get(i)).parent
              !=current)
42.         return false;
43.       workList.add((Tree)current.children.get(i));
44.       }
45.   }
46.   return true;
47. }
```

Lets consider a variant of `addChild` method that adds another `Tree` by adopting all its children:

```
81. public void addChild(Tree t) {
82.   BaseTree childTree = (BaseTree)t;
83.   if ( childTree.isNil() ) {
84.     if ( childTree.children!=null ) {
85.       int n = childTree.children.size();
86.       for (int i = 0; i < n; i++) {
87.         Tree c = (Tree)childTree.children.get(i);
88.         this.children.add(c);
89.         c.setChildIndex(children.size()-1);
90.       }
91.     }
92.   }
93. }
```

This method has a bug that it does not update parent relationship of adopted node. This bug can go undetected during the construction of the tree but can result in a faulty grammar later.

Figure 5a shows a valid tree accessible from root and Figure 5b shows another tree accessible from `t`. The `addChild` method is called on T_1 and Figure 5c shows the resulting erroneous structure with multiple missing parent assignments. Juzi returns $\langle T_5$, `parent`, $T_1\rangle$ and $\langle T_6$, `parent`, $T_1\rangle$ and repairAbstraction and debugAdviser suggest adding `c.parent = this;` at line 88 to fix the problem. The fix is verified by bounded exhaustive testing.

2) RayTracer: Ray tracing is a technique to produce images with realistic graphics by tracing paths of light through pixels in an image. This program has 1953 non-comment non-blank lines of code. RayTracer maintains the 3D model of the image in a structure `OctNode`. This structure divides the 3D space into eight subspaces hierarchically. Since most of the space is empty, `OctNode` avoids dividing empty subspaces. It maintains an Object List `ObjList` of type `ObjNode` and only constructs deeper tree in subspaces that contain objects. This design saves both memory and search time in the tree.

We inject a bug in the construction of `ObjList` and show how our approach can detect and fix it. The declaration of the class `LinkNode` in RayTracer implements a singly linked list. It has only one pointer `NextLink` to the next `LinkNode`. Its constructor receives another `LinkNode`, constructs a new

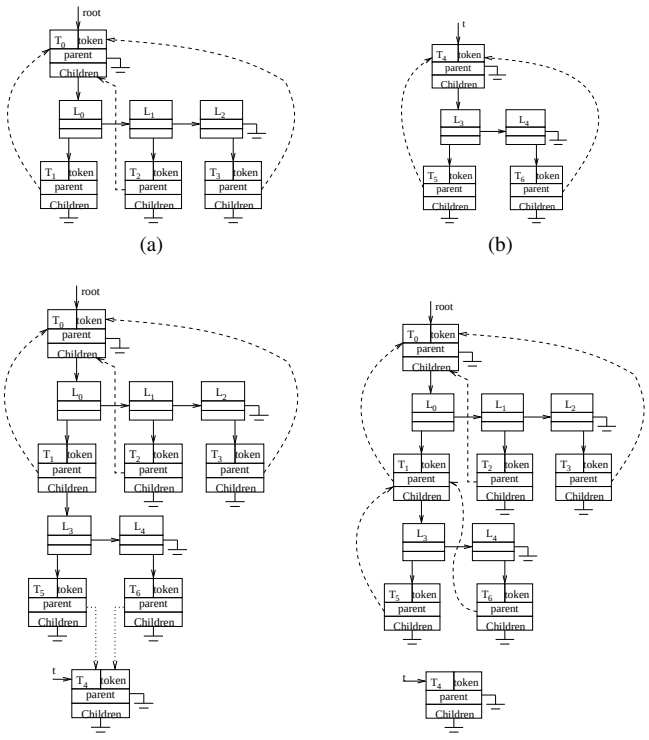

Figure 5. `CommonTree` accessible from `root`.(a) A bug free tree.(b) Tree T4 that has no payload but has children. (c) Erroneous tree state with two structural violations resulting from faulty method `addChild`. (d) Resulting structure after applying Juzi, this step also generated two repair actions that are translated to valid working java statement.

`LinkNode`, and sets its `NextLink` pointer to the passed `LinkNode`:

```
43. public class LinkNode  {
44.   private LinkNode NextLink;
45.   public LinkNode(LinkNode nextlink) {
46.     NextLink=nextlink;
47.   }
48. }
```

The payload of an `ObjectType` is added to `LinkNode` by the class `ObjNode`:

```
1. public class ObjNode extends LinkNode {
2.   private ObjectType theObject;
3.   public ObjNode(ObjectType newObj,LinkNode nextlink){
4.     super(nextlink);
5.     theObject=newObj;
6.   }
7. }
```

The workhorse class of RayTracer is `OctNode` that operates on these structures to maintain the space of the image it is representing:

```
1. public class OctNode {
2.   private OctNode[] Adjacent;
3.   private Face[] OctFaces;
4.   private OctNode[] Child;
5.   private ObjNode ObjList;
6.   private int NumObj;
}
```

Every `OctNode` points to two `OctNode` arrays of length eight; one `Adjacent` for neighboring OctNodes and the

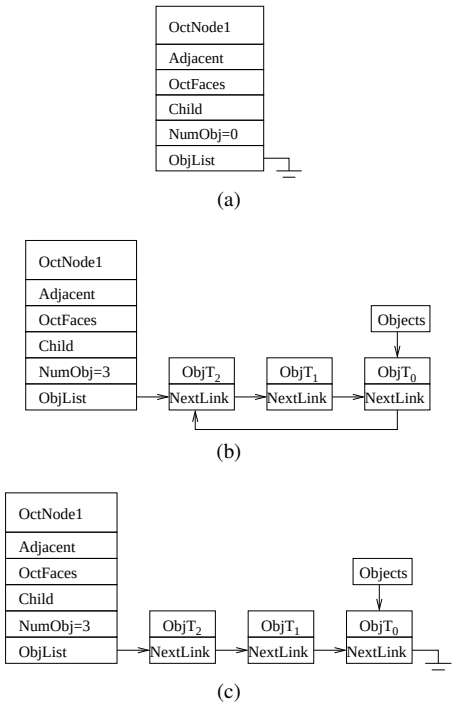

Figure 6. `OctNode` structure in RayTrace. (a) An empty ObjList (ObjNum 0).(b) An erroneous list of size 3 containing a cycle introduced by faulty code. (c) Fixed structure after applying Juzi.

other `Child` for Children OctNodes, also it has an array of length six for Faces. Each OctNode object has a sentinel header `ObjList` pointing to a list of objects contained in the `OctNode` space, it caches the size of this list in the field `NumObj`. Figure 6a illustrates an empty octNode.

The structural invariants of `ObjList` are correctness of sentinel `ObjList` nodes, acyclicity along `NextLink` fields, and correct value for `NumObj`:

```
30. boolean repOk(){
31.   if (ObjList == null) return NumObj==0;
32.   java.util.Set<ObjNode> visited =
33.     new java.util.HashSet<ObjNode>();
34.     visited.add(ObjList);
35.     ObjNode current = ObjList;
36.     while (current != null) {
37.       current = current.Next();
38.       if(!visited.add(current)){
39.         return false;
40.       }
41.    }
42.    return visited.size()== NumObj;
43. }
```

Consider the implementation of the method `CreateTree` below, that adds objects to `ObjList` when the objects are not `null` and less than maximum number of objects allowed for this `OctNode`:

```
66. private void CreateTree(ObjNode objects,
        int numObjects) {
67.   if (objects != null) {
68.     if (numObjects>MaxObj) CreateChildren(objects,1);
69.     else {
70.       ObjNode currentObj = objects;
71.       while (currentObj != null) {
72.         ObjNode newnode =
73.           new ObjNode(currentObj.GetObj(), ObjList);
74.         ObjList = newnode;
75.         currentObj = currentObj.Next();
76.       }
77.       objects.SetNext(ObjList); //Injected Fault
78.       NumObj = numObjects;
79.       }
80.    }
81. }
```

The method `CreateTree` has a fault in its line 77 that erroneously sets the last `ObjNode` is `NextLink` to the `ObjNode` pointed to by `ObjList`. Figure 6b demonstrate one such fault.

Based on Juzi's concrete repair action, our repair abstraction algorithm generates the following Java code, which results in correct `ObjList` in Figure 6c.

```
objects.SetNext(null);
```

C. Discussion

While this paper develops a technique for automated debugging, the algorithms that embody the technique have other novel applications, e.g., for highly optimized data structure repair. Abstract repair code could be injected into the faulty method to allow it to repair its own output on-the-fly without having to repeatedly run Juzi to repair the output. This approach has the potential of providing a substantial speed-up since Juzi performs a systematic search and requires repeated executions of `repOk` on each candidate repair action. Injecting abstract repair code would replace the search and perform repair in a negligible amount of time.

Another application is for programming by sketching [21]. The user could annotate the right-hand-side of a field assignment statement as unspecified, which can be treated initially as `null` and then repaired using our technique. We plan to build on our core technique to handle a larger class of faults and explore various novel applications in future work.

1) Threats to Validity: Our approach cannot handle faults in reachbality. That is, if the program is broken in such a way that Juzi cannot fix it, our approach cannot fix it as well.

In general, programs can have several different kinds of faults, e.g., a fault in a loop condition that performs an incorrect check or an incorrect overriding of `equals` method. We believe our technique can use specifications richer than structural invariants to generate debugging suggestions for a larger class of faults. Specifically, we are investigating the use of postconditions that relate method pre-state with post-state to correct erroneous implementations.

Our technique addresses faults only along one execution path. A method that has multiple faults along different execution paths can be handled by an iterative application of our technique using inputs that execute the different paths and augmenting bug fixes generated by the debugging advisor.

VI. Conclusion

There is a large body of research on automated debugging and fault localization, e.g., using delta debugging [22] and statistical debugging [23]. More recently Weimer et al. [24] have shown promising results in automatic program repair by finding patches using genetic programming; and Dallmeier et al. [20] have used differences in passing and failing runs of a program to locate bugs; While Wei et al. [25] have used program contracts to repair Eiffle programs. However, to the best of our knowledge, our work is the first to use data structure repair for repairing faulty code and generating debugging suggestions for complex data structures.

This paper introduced a novel methodology for developing reliable software: data structure repair for automated debugging. A technique embodying the methodology was developed based on two algorithms: (1) repair abstraction algorithm, which translates concrete repair actions of a data structure repair tool into Java code that represents the actions using variables visible in the scope of the faulty code; and (2) debugging advisor algorithm, which (heuristically) computes where to apply the fix. Demonstration of the technique using the Juzi repair tool as an enabling technology on subject programs from standard benchmarks and Java libraries shows the effectiveness and versatility of the technique.

We believe our methodology holds much promise, and is likely to provide a basis for developing new techniques that systematically test and debug erroneous programs and result in a synergy that significantly enhances software reliability and reduces the cost of software development.

Acknowledgment

This material is based upon work partially supported by the NSF under Grant Nos. IIS-0438967 and CCF-0845628, and AFOSR grant FA9550-09-1-0351.

References

[1] National Institute of Standards and Technology, "The economic impacts of inadequate infrastructure for software testing," Planning report 02-3, May 2002.

[2] B. Beizer, *Software Testing Techniques*. International Thomson Computer Press, 1990.

[3] P. Ammann and J. Offutt, *Introduction to Software Testing*. Cambridge University Press, 2007.

[4] A. Zeller, *Why Programs Fail: A Guide to Systematic Debugging*. Morgan Kaufmann, 2005.

[5] E. M. Clarke, O. Grumberg, and D. A. Peled, *Model Checking*, 1999.

[6] F. Nielson, H. R. Nielson, and C. L. Hankin, *Principles of Program Analysis*. Springer, 1999.

[7] L. A. Clarke and D. S. Rosenblum, "A historical perspective on runtime assertion checking in software development," *SIGSOFT Software Engineering Notes*, vol. 31, no. 3, 2006.

[8] F. Chen and G. Roşu, "Java-MOP: A monitoring oriented programming environment for Java," in *TACAS*, 2005.

[9] B. Demsky, "Data structure repair using goal-directed reasoning," Ph.D. dissertation, Massachusetts Institute of Technology, 2006.

[10] B. Elkarablieh, "Assertion-based repair of complex data structures," Ph.D. dissertation, University of Texas at Austin, 2009.

[11] S. Khurshid, I. García, and Y. L. Suen, "Repairing structurally complex data," in *12th SPIN Workshop on Model Checking of Software*, 2005.

[12] B. Elkarablieh and S. Khurshid, "Juzi: A tool for repairing complex data structures," in *ICSE*, 2008.

[13] B. Elkarablieh, I. Garcia, Y. L. Suen, and S. Khurshid, "Assertion-based repair of complex data structures," in *ASE*, 2007.

[14] C. Boyapati, S. Khurshid, and D. Marinov, "Korat: Automated testing based on Java predicates," in *ISSTA*, 2002.

[15] D. Marinov, "Automatic testing of software with structurally complex inputs," Ph.D. dissertation, Computer Science and Artificial Intelligence Laboratory, Massachusetts Institute of Technology, 2004.

[16] M. Z. Malik, K. Ghori, B. Elkarablieh, and S. Khurshid, "A case for automated debugging using data structure repair," in *ASE*, 2009.

[17] B. Liskov and J. Guttag, *Program Development in Java: Abstraction, Specification, and Object-Oriented Design*, 2000.

[18] B. Elkarablieh, Y. Zayour, and S. Khurshid, "Efficiently generating structurally complex inputs with thousands of objects," in *ECOOP*, 2007.

[19] D. Jeffrey, N. Gupta, and R. Gupta, "Fault localization using value replacement," in *ISSTA*, 2008.

[20] V. Dallmeier, A. Zeller, and B. Meyer, "Generating fixes from object behavior anomalies," in *ASE*, 2009.

[21] A. Solar-Lezama, R. Rabbah, R. Bodik, and K. Ebcioglu, "Programming by sketching for bit-streaming programs," in *PLDI*, 2005.

[22] A. Zeller, "Isolating cause-effect chains from computer programs," in *FSE*, 2002.

[23] T. Chilimbi, B. Liblit, K. Mehra, A. Nori, and K. Vaswani, "Holmes: Effective statistical debugging via efficient path profiling," in *ICSE*, 2009.

[24] W. Weimer *et al.*, "Automatically finding patches using genetic programming," in *ICSE*, 2009.

[25] Y. Wei, Y. Pei, C. A. Furia, L. S. Silva, S. Buchholz, B. Meyer, and A. Zeller, "Automated fixing of programs with contracts," in *ISSTA*, 2010.