The Dissertation Committee for Muhammad Zubair Malik
certifies that this is the approved version of the following dissertation:

Combining Data Structure Repair and Program Repair

Committee:

Sarfraz Khurshid, Supervisor

Dewayne Perry

Craig Chase

Christine Julien

Daniel Miranker

Combining Data Structure Repair and Program Repair

by

Muhammad Zubair Malik, B.S.; M.S.; M.S.E.

DISSERTATION

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

DOCTOR OF PHILOSOPHY

THE UNIVERSITY OF TEXAS AT AUSTIN

August 2014

Dedicated to my family

&

Melanie Gulick

(for going beyond the call of duty)

Acknowledgments

This thesis has been made possible by the help of many people. I do not think I will be able to thank them all of them but I am truly grateful to everyone who helped.

First of all my heartfelt gratitude goes to my adviser Dr. Sarfraz Khurshid for helping in all aspects of this work. He is an excellent teacher, prolific in providing research ideas and involved in guiding the research process. He is a sound adviser and helped me achieve my objectives.

Dr. Craig Chase has been a constant source of inspiration, knowledge and guidance over the years. I cannot express my gratitude towards him in words. I have spent more time in his company than any other professor at UT and enjoyed every second of it.

I am deeply indebted to Prof. Dewayne Perry for his kindness and guidance throughout my stay at UT. He always has an interesting story to instruct as well as lighten up the mood. I find his empirical style of research fascinating but had no idea how hard it is to find support for projects in empirical setting until I collaborated on Megan's course project with him.

Dr. Christine Julien is always cheerful and welcoming. I really enjoyed attending her courses and discussions in the hallway.

I am thankful to Prof. Daniel Miranker for being gracious and accepting

to be the external member in on my PhD committee. His questions and comments helped me improve this work considerably.

My special thanks goes to Dr. Suzanne Barber for mentoring me and allowing me to apply my research to novel applications.

I have been proud to be a part of SVVAT research group. I was fortunate to work with brilliant researchers including: Daryl Shannon, Danhua Shao, Bassem Elkrablieh, Engin Uzuncaova, Shadi Abdul Khalek, Junaid Siddquie, Peter Kim, Divya Gopinath, Razieh Zaeem, Guowei Yang, Sam Harwell, Shiyu Dong, Lingming Zhang, and Khalid Ghori. The idea of using data structure repair for program repair was first introduced in Khalid Ghoris Masters thesis.

I must give the strongest acknowledgement to Melanie Gulick and the rest of ECE staff who made my life easier and provided outstanding help in completing my work.

I have had many roommates and friends over the years who made my stay at UT a fun experience. Thank you for being there guys: Khubaib, Owais Khan, Amber Hassan, Umar Farooq, Rashid Kaleem, Faisal Iqbal, Megan Ruthven, Junaid Siddique, Tauseef Rab, Ahmad Sheikh, Ansab Ali, Muqeet Ali, Younas Sajjad, Aman Pervaiz, Malik Saleh and Omar Shakeel.

Finally, I am thankful to my family for their unconditional love and support.

The work presented in this dissertation is partially supported by the National Science Foundation under Grant Nos. CCF-0845628, IIS-0438967, and CCF-0702680, and AFOSR grant FA9550-09-1-0351.

Combining Data Structure Repair and Program Repair

Publication No. _____

Muhammad Zubair Malik, Ph.D.
The University of Texas at Austin, 2014

Supervisor: Sarfraz Khurshid

Bugs in code continue to pose a fundamental problem for software reliability and cause expensive failures. The process of removing known bugs is termed *debugging*, which is a classic methodology commonly performed before code is deployed. Traditionally, debugging is tedious, often requiring much manual effort. A more recent technique that complements debugging is *data structure repair*, which handles bugs that make it to deployed systems and lead to erroneous behavior at runtime by modifying erroneous program states to recover from errors. While data structure repair presents a promising basis for dealing with bugs at runtime, it remains computationally expensive.

Our thesis is that debugging and data structure repair can be integrated to provide the basis of an effective approach for removing bugs before code is deployed and handling them after it is deployed. We present a *bi-directional* integration where ideas at the basis of data structure repair assist in automating debugging

and vice versa. Our key insight is two-fold: (1) a repair *action* performed to mutate an erroneous object field value to repair it can be abstracted into a program statement that performs that update correctly; and (2) repair actions that are performed repeatedly to fix the same error can be *memoized* and re-used.

We design, develop, and evaluate two techniques that embody our insight. One, we present an automated debugging technique that leverages a systematic constraint-based data structure repair technique developed in previous work and provides suggestions on how to fix a faulty program. Two, we present *repair abstractions* that are based on the same central ideas as in our automated debugging technique and memoize *how* an erroneous state was repaired, which enables prioritizing and re-using repair actions when the same error occurs again.

The focus of our work is programs that operate on structurally complex data, e.g., heap-allocated data structures that have complex structural integrity constraints, such as acyclicity. Checking such constraints plays a central role in the techniques that lay at the foundation of our work. These techniques require the user to provide the constraints, which poses a burden on the user. To facilitate the use of constraint-based techniques, we present a third technique to check constraint violations at runtime using *graph spectra*, which have been studied extensively by mathematicians to capture properties of graphs. We view the heap of an object-oriented program as an edge-labeled graph, which allows us to apply results from graph spectra theory. Experimental results show the effectiveness of using graph spectra as a basis of capturing structural properties of a class of commonly used data structures.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Traditional methodologies for increasing software reliability using analysis fall in two basic categories: checking before code is deployed [1, 10, 17, 81, 115], e.g., using software testing and debugging, model checking, or static analysis; and runtime monitoring or error recovery after code is deployed [14, 18, 26, 30], e.g., using data structure repair. We introduce a novel methodology that integrates ideas from a pre-deployment technique — debugging — and a post-deployment technique — data structure repair — to enable a synergy that holds potential to significantly increase software reliability.

1.1 Our thesis

Our thesis is that debugging and data structure repair can be integrated to provide the basis of an effective specification-based approach for removing bugs at compile-time and handling them at runtime. We propose a *bi-directional* integration where ideas at the basis of data structure repair assist in automating debugging and vice versa. Our key insight is two-fold: (1) a *repair action* performed to mutate an erroneous object field value to repair it can be abstracted into a program statement that performs that update correctly; and (2) *abstract repairs* can capture the essence

of how to fix specific kinds of errors in concrete data structures and help optimize fixing the same error in future through memoization and re-use. Moreover, to ease the burden on the user to write specifications, we propose a technique based on graph theoretic foundations to detect and check invariants of data structures.

1.2 This dissertation

This dissertation presents the design, implementation and evaluation of three techniques for program repair, data structure repair and structural invariant detection, which embody our thesis (Figure 1.1). We next describe the background for each technique and summarize its key ideas.

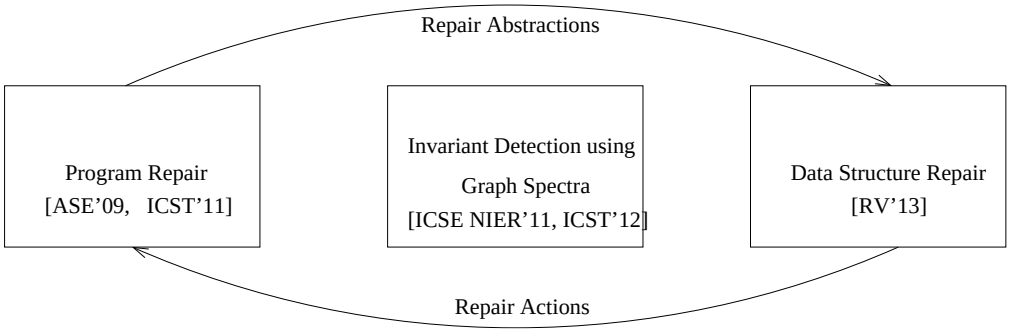


Figure 1.1: Dissertation overview. Data structure repair provides the basis of program repair [67]. Repair actions are concrete mutations to fix errors in program state and provide hints for likely fault location and repair. We implemented [72] a program repair framework that abstracts repair actions using heuristics, dataflow and control-flow analysis and presented experimental evaluation. The idea of abstracting repair actions enables more efficient data structure repair [113]. Both our approaches for program repair and data structure repair require the user to provide a given specification of data structure properties. We introduced the idea of using graph spectra to capture structural constraints of data structures using dynamic analysis [68].

1.2.1 Debugging

Debugging faults in code is tedious and can itself be error-prone. Using a traditional debugging environment, a programmer has to manually trace the execution of the program. On finding a corrupted program state the programmer has to make assumptions about fault location(s) and create possible fix(es). This can be quite time-intensive; moreover the fix may introduce some new bugs. Sometimes it is hard to trace the root fault as the fault may seem to propagate from one place to another. A variety of tools and techniques have been developed to help with localizing bugs in programs [43, 115]. Sometimes localizing bugs and fixing code manually can create more bugs or different types of bugs that might not have been present earlier.

A key element of debugging is *program repair*, which is the problem of transforming faulty lines of code into correct code. We present a novel technique for automated repair of buggy programs, which eases the burden of debugging by suggesting likely fixes to faulty code [67, 72]. Our technique first uses the buggy program to generate corrupt program states, next it repairs these states by invoking an off-the-shelf *data structure repair* tool [32, 60], and then it abstracts the repair actions and synthesizes code that represents a likely fix. Our technique performs *specification-based* repair using given data structure invariants. Such invariants have been used in previous work on systematic testing [13], which can be applied in conjunction with our program repair approach.

1.2.2 Data structure repair

A variety of techniques [26, 27] have been developed during the last decade to *repair* structurally complex data that do not satisfy the desired structural integrity constraints at run-time. Conventional use of these techniques has been to enable continued execution of programs in case of otherwise fatal data structure corruption. One such technique is Juzi, which provides an enabling technology for our work [30,32,33,60]. To fix corrupt data structures, Juzi generates and applies repair actions that represent mutations to the structures so that the transformed structures satisfy the desired constraints.

While data structure repair presents a promising basis for dealing with bugs at runtime, scaling it remains a key challenge. We introduce *repair abstractions* for more efficient data structure repair. Our key insight is that if an error in the program state is due to a bug in code (or a fault in hardware), a similar error may occur again, say when the same buggy code segment is executed again (or when the same faulty memory location is accessed again). Conceptually, repair abstractions capture *how* erroneous program executions are repaired using concrete repair actions to allow faster repair of similar errors in future executions.

1.2.3 Structural constraints

The focus of our work is programs that operate on structurally complex data, e.g., heap-allocated data structures that have complex structural integrity constraints, such as acyclicity. Checking such constraints plays a central role in the techniques that lay at the foundation of our work. These techniques typically re-

quire the user to provide the constraints. Writing complex constraints manually poses a burden on the user. To facilitate the use of constraint-based techniques, we present a novel technique to check constraint violations at runtime using graph *spectra*, which have been studied extensively by mathematicians to capture properties of graphs [68, 69]. Viewing the heap of an object-oriented program as an edge-labeled graph allows us to apply results from graph spectra theory [22] to perform dynamic program analysis.

1.3 Illustration

This section gives an illustrative overview of our techniques using a faulty implementation of a doubly-linked list data structure. Consider the following declaration of a class `List`:

```
1   class List{
2       static class Node{
3           int data;
4           Node next;
5           Node prev;
6       }
7       Node head, last;
```

The class `Node` implements the list nodes. Each node has an integer `data`, as well as `next` and `prev` pointers to other nodes. Figure 1.2(a) illustrates a linked list with four nodes.

The structural invariants (called *class invariants* in object oriented programs) of doubly-linked lists in this example are *acyclicity* along `next` fields as well as along `prev` fields, and *transpose* relation between `next` and `prev` fields. Any

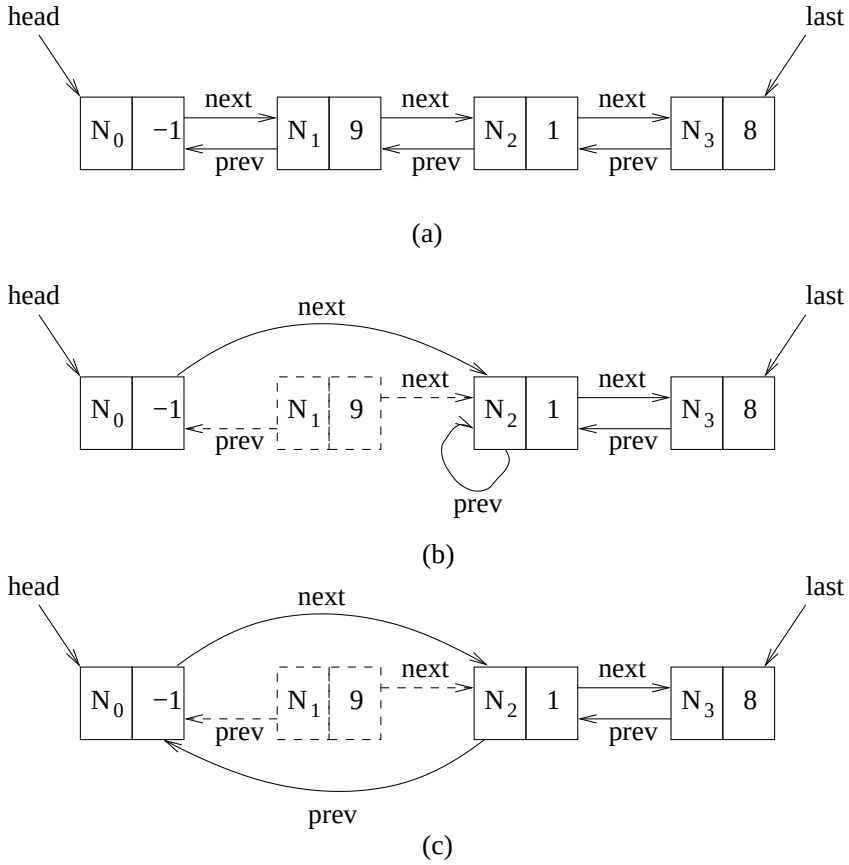


Figure 1.2: (a) A doubly linked list with four nodes. (b) Erroneous output (post-state) of the `remove` method where the value 9 is removed from the list in part a (c) Correct output of the repaired `remove` method.

valid list must satisfy these invariants (in all publicly visible states). These invariants can be represented using a `repOk` [66] method that traverses its input structure and returns true if and only if the input satisfies all the invariants:

```
33   public boolean repOK() {
34     if (head == null || last == null)
35       return head == last;
36     if (head.prev != null)
37       return false;
```

```
38      if (last.next != null)
39        return false;
40      HashSet<Node> hs = new HashSet<Node>();
41      Node curr = head;
42      while (curr != null) {
43        if (!hs.add(curr))
44          return false;
45        if (curr.next != null && curr.next.prev != curr)
46          return false;
47        curr = curr.next;
48      }
49      return true;
50    }
```

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.

1.3.1 Program Repair

Consider the following implementation of the method `remove`:

```
8   public void remove(int n) {
9       Node curr = head;
10      while (curr != null && curr.data != n) {
11          curr = curr.next;
12      }
13      if (curr == null) //Data not found, nothing to delete
14          return;
15      if(curr.next == null && curr.prev == null){
16          head = last = null;
17          return;
18      }
19      if (curr.next == null){ //last element in the list
20          last = curr.prev;
21          last.next = null;
22          return;
```

```
23        }
24        if (curr.prev == null){
25            head = curr.next;
26            head.prev = null;
27            return;
28        }
29        curr.prev.next = curr.next;
30        //curr.next.prev = curr.next; // Error
31        curr.next.prev = curr.prev;   // Fix
32    }
```

The method has a fault at the assignment statement on line 30. The line erroneously sets the `prev` pointer of the `next` of `curr` to point back to itself, thus violating the transpose relationship. Figure 1.2(b) illustrates the state of the list once the method is executed on the input list from Figure 1.2(a). Figure 1.2(c) shows the repaired output.

A faulty assignment of `curr.next.prev` to `curr.next` breaks the transpose relationship between `next` and `prev` and violates the structural integrity of the list. A data structure repair routine such as Juzi [32] is able to restore the structural integrity by repairing this data structure corruption. Juzi uses the `repOk` method to check the structural constraints and when it detects a violation it systematically mutates the structure by performing *repair actions* to transform it into one that does not violate the constraints. A Juzi repair action is a triple $< s, f, d >$ where s is the source node, f is the field mutated and d is the destination node. In this example, Juzi first attempts $< N2, prev, null >$ which fails to fix the structure, followed by $< N2, prev, N0 >$ which brings the list to a valid structural state.

Our program repair approach translates the concrete repair action performed by Juzi into a Java statement, i.e., `curr.next.prev = curr.prev;`, using program variables visible in the scope, e.g., `head`, `last`, `curr` and `this`, which hold references to objects on the heap. Our approach records the heap locations referenced by each of these variable and uses bounded *path expressions* over recursive fields to determine statements that capture the state modifications suggested by Juzi.

1.3.2 Data Structure Repair

To illustrate our approach to data structure repair, consider a scenario where the `remove` method is executed periodically and it continually produces incorrect outputs, which are repaired using a data structure repair routine, such as Juzi. Each invocation of Juzi requires a systematic exploration of a space of candidate structures. However, it is the *same* fault in `remove` that is the cause of data structure corruption.

Our data structure repair approach creates a *repair abstraction* for a concrete repair action to memoize *how* the corruption was fixed. When a similar corruption is encountered during a future run, the abstract repair action is first performed as a heuristic; if the heuristic leads to a successful repair, data structure repair completes, and otherwise, the default Juzi algorithm applies. For this example, the Juzi repair action $< N2, prev, N0 >$ is abtracted to $< prev, Neighbor >$, where the keyword $Neighbor$ indicates that a mutation of the `prev` field should be (heuristically) prioritized to first point to a neighboring node, i.e., a node that is directly connected to the source node along one field access. Thus, the value `null` has a

lower priority among the set of possible repairs.

1.3.3 Graph Spectra

Data structure repair routines, such as Juzi, that provide the foundation of our work use given structural invariants as a basis of performing repair. While user-provided `repOk` methods enable such routines to apply, we envision new approaches that do not require the users to provide detailed invariants. Specifically, here we illustrate our approach for capturing structural invariants using *graph spectra* [22].

Recall that our example lists are acyclic along `prev` fields. Note also that the faulty `remove` method violates this property. Our technique represents object graphs using adjacency matrices and uses their *spectra*, i.e., eigenvalues, to classify them. The eigenvalues for lists in Figure 1.2 (a) and (c) are $(0, 0, 0, 0)$ and $(0, 0, 0)$ respectively, i.e., they are all zeroes. However, the eigenvalues for the list in Figure 1.2(b) are $(0, 0, 0, 1)$, i.e., they contain a non-zero element. Since all eigenvalues of an acyclic structure are zero [22], the list in part (b) violates the acyclicity invariant. Thus, this violation is detected by using results from graph spectra, without using a user-provided `repOk` method.

1.4 Contributions

The results in this thesis are based on work published at: ASE 2009 [67], ICST 2011 [72], ICSE NIER 2011 [68], ICST 2012 [69] and RV 2013 [113]. We make the following contributions:

1. **Program Repair using Data Structure Repair**

   We present an approach to repair programs using data structure specifications with the following key contributions:

   - **Algorithms.** We present two algorithms that form the basis of our approach: one algorithm performs the abstraction of concrete repair actions and the other algorithm uses abstract repair to generate debugging suggestions. We argue the correctness of our approach.
   - **Evaluation.** We evaluate our approach for its success rate on faulty mutants of a suite of programs, including some benchmark data structures as well as parts of the ANTLR and RayTrace applications. Experimental results show our approach generates accurate debugging suggestions.

2. **Dynamic Analysis using Graph Spectra**

   We present an approach for dynamic analysis of data structure implementations with the following key contributions:

   - **Graph spectra in dynamic analysis.** We introduce the use of graph spectra in dynamic analysis of programs that manipulate structurally complex data.
   - **Technique.** We present a technique for detecting likely properties of object graphs in Java programs.
   - **Evaluation.** We use a suite of subject programs that implement complex data structures to evaluate our technique. Experimental results

show that our technique holds much promise in accurately identifying structural properties as well as detecting likely erroneous executions.

3. **Prioritizing Data Structure Repair using Repair Abstractions**

   We present a technique to prioritize repairs based on previously known errors and repairs.

   - **Memoizing repair actions.** We introduce the idea of memoizing repair actions from previous runs of a data structure routine in order to prioritize future repairs and optimize performance.
   - **Repair abstractions.** We define repair abstractions that generalize concrete data structure repair actions into *concepts* based on rooted, edge-labeled graphs, which capture the essence of concrete repair actions and enable their re-use in future.
   - **Evaluation.** We experimentally evaluate our technique and compare it with previous work on invariant-based repair.

1.5 Organization

The rest of this document is organized as follows. Chapter 2 presents our work on program repair. Chapter 3 describes our technique to capture data structure invariants using graph spectra. Chapter 4 describes our technique to optimize data structure repair using repair abstractions. Chapter 5 presents the related work and Chapter 6 concludes the thesis.

Chapter 2

Program Repair Using Data Structure Repair

This chapter describes our approach to program repair using data structure repair. The basic problem our approach addresses is to modify a given faulty program p into another program p' such that p' is correct with respect to a given bounded correctness criteria. We support two forms of criteria: (1) a given test suite where each test case consists of an input and the corresponding expected output; and (2) a given specification, which allows enumerating test inputs and provides the correctness properties of p, as well as a bound on the input size to check p. Thus, the transformed program p' is correct with respect to a bounded number of possible program behaviors, which are either presented explicitly in the form of the given test suite or implicitly in the form of the specification and the bound on input size. This chapter is based on our work presented in ASE 2009 [67][1] that makes a case for state repair for program repair, and our ICST 2011 [72][2] paper that presents the details and evaluation of the work.

[1]Muhammad Zubair Malik, Khalid Ghori, Bassem Elkarablieh, Sarfraz Khurshid. A Case for Automated Debugging Using Data Structure Repair. *ASE 2009*. (Ghori and Elkarablieh are former UT students supervised by Khurshid.)

[2]Muhammad Zubair Malik, Junaid Haroon Siddiqui, Sarfraz Khurshid. Constraint-Based Program Debugging Using Data Structure Repair. *ICST 2011*. (Siddiqui is a former UT student supervised by Khurshid.)

The key insight of the work presented in this chapter is to leverage the information available in program state to repair the program code. We show that for a large class of programs that operate over complex data structures using the data structure repair is efficient and effective for program repair. Traditional approaches of automated debugging such as delta debugging [114], statistical debugging [15], and spectral techniques [56] focus on isolating cause of failure in code but do not provide repair suggestions on how to *modify* code. More recent work addresses automated program repair [40], which we discuss in detail in Chapter 5. To our knowledge, our work is the first to introduce the idea of modifying program states as a basis of synthesizing code that repairs a faulty program.

We present a novel methodology for developing reliable software: data structure repair for automated debugging. A technique embodying the methodology is 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.

The idea of using data structure repair for program repair was first introduced in Ghori's Masters thesis [39]. This chapter refines the original ideas into two core algorithms, argues the correctness of the approach, and presents a rigorous experimental evaluation using textbook data structure implementations as well

as structures derived from parts of the ANTLR [84] and RayTrace [11] applications.

2.1 Debugging Using Data Structure Repair Overview

Systems with high reliability and availability requirements have used data structure repair over the last few decades as an effective means to recover on-the-fly from errors in program state [26,32]. 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, using dataflow and heuristics, the place to put the generated sequence in the method.

2.2 Motivating 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 2.1 (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` [66] 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;
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 has been modified from its original implementation:

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

This method has an injected fault in its third assignment statement (Line 44), which erroneously sets a `previous` field to `entry` instead of `newEntry`[3]. 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 2.1 (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 2.3 — repairs the list by performing the following repair action: $\langle E_0, \texttt{previous}, E_1 \rangle$, 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 `addFirst` would generate.

At every control point in the program we record all objects that are reachable from any reference field. Based on Juzi's concrete repair action, our repair abstraction algorithm using semantics of Java reference fields 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;
```

[3]The `addFirst` method is correctly implemented in `java.util.LinkedList` and uses the helper method `addBefore`, which we inline here to make it accessible for our tool

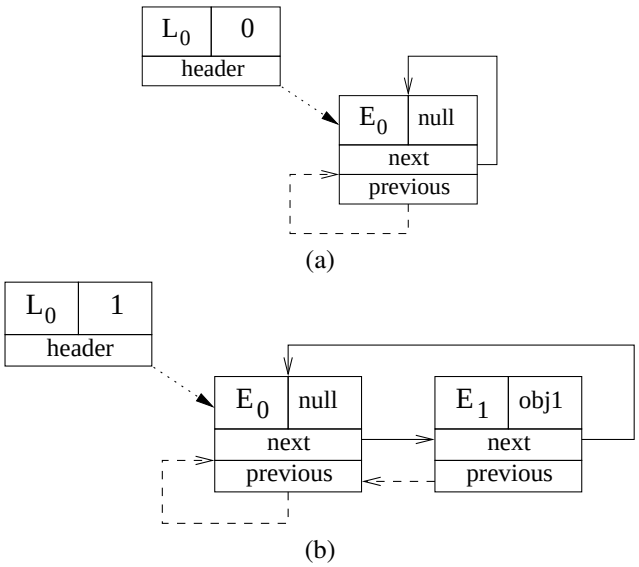


Figure 2.1: Doubly-linked circular list with sentinel `header`. (a) An empty list (size 0). (b) An erroneous list of size 1 containing element 0. A small box-pair represents a list object and is labeled with the object's identity and the value of its `size`. Large box-pairs represent entry objects and are labeled with object identity and value of `element`.

Our debugging advisor algorithm based on the dataflow suggests this fix for Line 46, which corrects the fault.

2.3 Background: Data Structure Repair using Juzi

Juzi is an automated framework for on-the-fly repair of data structures [30, 32,33,60]. 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. Consider the list in Figure 2.1 (b). The list has one corruption in the `previous` field on node E_0. Given the corrupt structure and the `repOk` method, 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

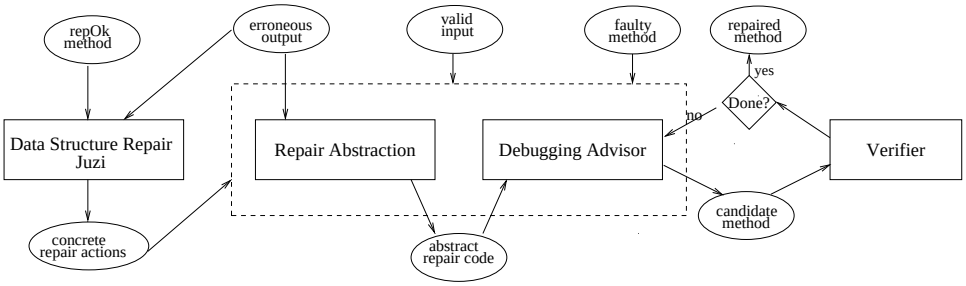


Figure 2.2: Automated program repair using data structure repair. Solid boxes represent computation modules, the ovals represent data, and the arrows show the flow of information. The dashed box represents the two algorithms that embody our technique. The arrows reaching only the dashed box imply that the information is available to both the algorithms.

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.

2.4 Program Repair

Figure 2.2 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 er-

roneous 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 is validated over the test cases that represent the bounded correctness criteria.

2.4.1 Repair Abstraction Algorithm

This algorithm abstracts concrete repairs suggested by Juzi into the actual program code. Figure 2.3 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 initialize the code handles to correct value. For all Juzi repair actions it map objects in Juzi repair action to code handles using conservative reasoning, and translate repair action to build a code statement. The algorithm updates the code and applies repair actions on the program state. And finally it updates 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.

To illustrate, consider executing the method `addFirst` on the empty input list shown in Figure 2.1 (a). The map at Line 46 for this execution is:

```
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 2.3: 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 sub-sequence 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 $\langle E_0, \texttt{previous}, E_1 \rangle$ (Section 3.2). 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 [13]).

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.

2.4.2 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

```
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 2.4: Debugging advisor algorithm.

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 2.4) 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 `addFirst` (Section 2.2). 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.

Table 2.1: Success Rate Evaluation

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

2.4.3 Possible Integration with Bounded Exhaustive Testing

To increase confidence in the correctness of the repaired method, our technique allows a direct application of the Korat framework for systematic testing [13, 73] to automatically generate valid inputs and check outputs using `repOk` when the correctness criteria includes the `repOk` method. Moreover, any bugs discovered by Korat can feedback into our technique to use it to iteratively debug a faulty program that has multiple faults along different control-flow paths.

The validation by Korat can be implemented within the method `checkFixFeasibility`, which allows us to use a counterexample-driven refinement of fixes proposed by our algorithm. Korat is a structural constraint solver that can be used to generate non-isomorphic inputs for bounded exhaustive testing. The inputs generated by Korat can be used to check the proposed fixes.

2.5 Evaluation

We report experiments and case studies designed to evaluate success rate of our approach and its effectiveness. We use fault injection in sourcecode to create

faulty versions of our subject programs. We apply our technique to repair the faulty versions and compare the repaired code to original correct code. We use five textbook data structures to compute the success rate of our approach in repairing faults. Moreover, we evaluate the applicability and effectiveness of our approach by case studies on programs based on parts of two real-world applications.

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

2.5.1 Success Rate

We report the overall *Success Rate* of our approach as the ratio of the number of correct fixes generated by our approach with respect to the total number of faulty versions created by our fault injection methodology. Since our approach relies fundamentally on the ability of Juzi to successfully repair program state and Juzi may or may not be able to repair each erroneous state, we also report *Repairable Success Rate*, which is the success rate ratio computed only with respect to errors that actually can be repaired by Juzi.

We consider faults of *omission*, where the programmer forgets to write the necessary code, and faults of *commission*, where the programmer writes incorrect code. Our focus is on injecting semantic faults, which are not detected in the compilation process. We assume that the starting code is correct. To inject faults that mimic omission, we remove one or more lines of code; the faults of commission are mimicked by using standard mutation operators [1].

Table 2.1 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 methods 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 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*. *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 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.

2.5.2 Applicability and Effectiveness

To evaluate the applicability and effectiveness of our approach, we performed two case studies on code derived from parts of real programs. Our studies show that program repair approach holds potential to work on real programs and fix non-trivial bugs. Specifically, we consider (1) ANTLR [84] — a compiler

generator, and 2) RayTrace [11] — a program for tracing lights paths through an image.

2.5.2.1 Experiment Design

Debugging and repairing large scale open source software is as much art as much it is a systematic processes. The well defined application programming interfaces promote component based development and allow unit tests to be performed using stubs. The stubs present pre-defined correct behaviour of interacting components and allow the debugging process to focus on the behavior of the piece of code under test. This simplifies the test input generation and makes unit testing more effective. This processes does not exclude the need of system-level testing but does give confidence on the features tested for the current piece of code.

In the design of this study we have focused our approach only on the relevant piece of code, assuming that the source of error has already been localized. The tests used were also generated to specifically reach the faulty method and hit the faulty code block. The various method calls in these methods were stubbed to return only the correct and desired output when called with the provided test.Using this setup we were able to focus our approach on the desired functionality and error. This behavior is inline with the actual debugging process followed by human developers and does not undermine the soundness of our approach. It does reflect our belief that we expect every other part of the code to behave according to their specifications.

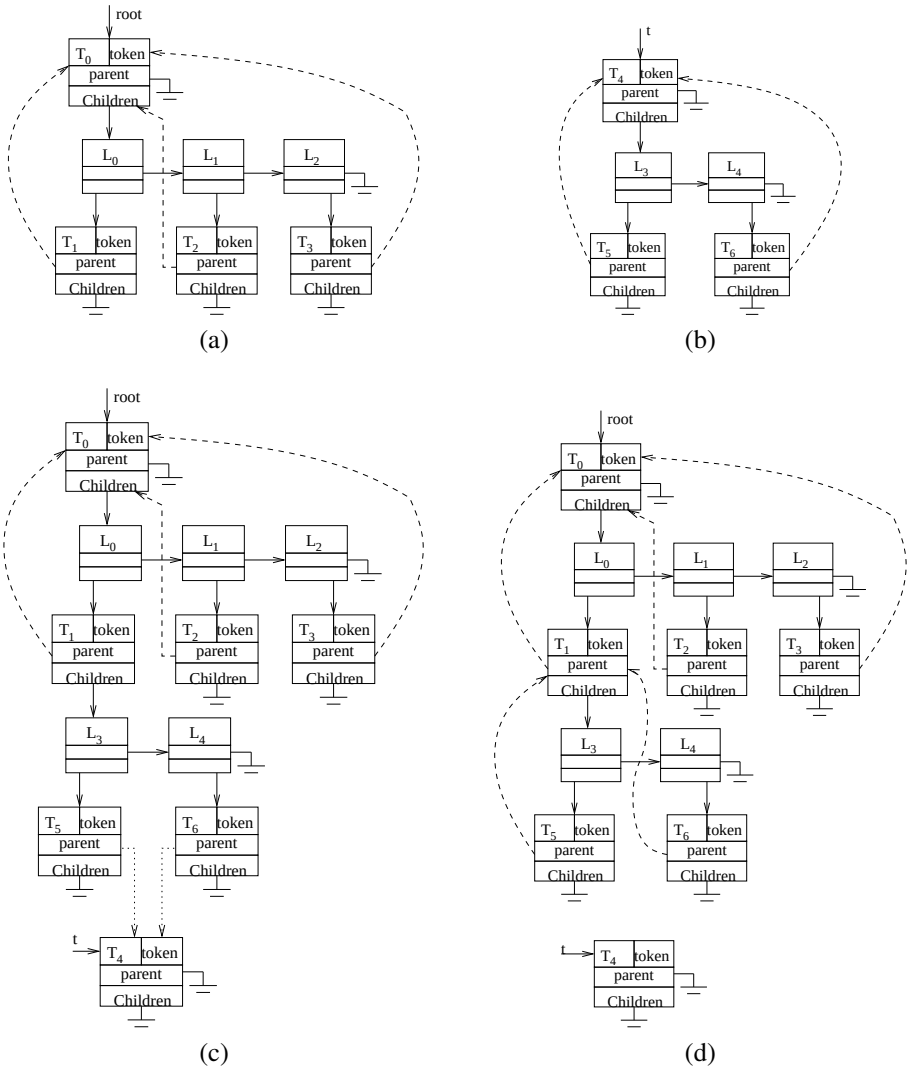


Figure 2.5: `CommonTree` accessible from `root`.(a) A bug free tree.(b) Tree T4 that has no payload but has children. (c) Erroneous tree state with two structural violations resulting from faulty method `addChild`. (d) Resulting structure after applying Juzi, this step also generated two repair actions that are translated to valid working Java statement.

2.5.2.2 ANTLR

ANother Tool for Language Recognition (ANTLR) [84] 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. It is one of the most widely used parser generators, language translator and interpreter. 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. 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).

Our proof of concept implementation does not handle all intricacies of class hierarchy which is fairly complicated in large scale software like ANTLR. We adapt the ANTLR code by squashing the class hierarchy and bringing all declarations in the instantiable class `CommonTree`.

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 `CommonTree` 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 (){
31  CommonTree root = this;
32  if (root==null) return true;
33    Set<CommonTree> visited = new HashSet<CommonTree>();
34    visited.add(root);
35    LinkedList<CommonTree> workList =
36                            new LinkedList<CommonTree>();
37    workList.add(root);
38    while (!workList.isEmpty()) {
39      CommonTree current = workList.removeFirst();
40      if(!visited.add(current))
41        return false;
42      for(int i= 0;i<current.children.size();i++){
43        if((current.children.get(i)).parent != current)
44          return false;
45        workList.add((Tree)current.children.get(i));
46      }
47    }
48    return true;
49  }
```

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

```
81 public void addChild(CommonTree childTree) {
82     if ( childTree==null ) {
83         return;
84     }
85     if ( childTree.isNil() ) {
86         if ( this.children!=null &&
87                 this.children == childTree.children ) {
88             throw new RuntimeException(
89                     "attempt to add child list to itself");
90         }
91         if ( childTree.children!=null ) {
92             if ( this.children!=null ) {
93                 int n = childTree.children.size();
94                 for (int i = 0; i < n; i++) {
95                     CommonTree c = childTree.children.get(i);
```

```
96                      this.children.add(c);
97                      c.setParent(this);
98                      c.setChildIndex(children.size()-1);
99                  }
100             } else {
101                 this.children = childTree.children;
102                 for (int c = 0; c < children.size(); c++) {
103                     CommonTree child = getChild(c);
104                     child.setChildIndex(c);
105                     child.setParent(this);
106                 }
107             }
108         }
109     }else {
110         if ( children==null ) {
111             children = createChildrenList();
112         }
113         children.add(childTree);
114         //childTree.setParent(this);   //Injected Error
115         childTree.parent = this;       //Fix
116         childTree.setChildIndex(children.size()-1);
117     }
118 }
```

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 2.5 (a) shows a valid tree accessible from root and Figure 2.5 (b) shows another tree accessible from `t`. The `addChild` method is called on T_1 and Figure 2.5 (c) 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 `childTree.parent = this;` at line 115 to fix the problem. The fix is verified by bounded exhaustive testing.

2.5.2.3 RayTrace

Ray tracing is a technique to produce images with realistic graphics by tracing paths of light through pixels in an image. RayTrace 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. We simplified the original code for our study to enable our tool to handle it; specifically, we remove the abstract classes and interfaces in the class hierarchy and focus on the concrete class `ObjNode`:

```
1 public class ObjNode{
2     private Object theObject;
3     private ObjNode NextLink;
4     public  ObjNode(Object newObj) {
5         theObject=newObj;
6     }
```

The workhorse class of RayTrace is `OctNode` that operates on these structures to maintain the space of the image it is representing. 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 2.6a illustrates an empty OctNode.

The structural invariants of `OctNode`, which are relevant to its list nodes, are acyclicity along `NextLink` fields and correct value for `NumObj`:

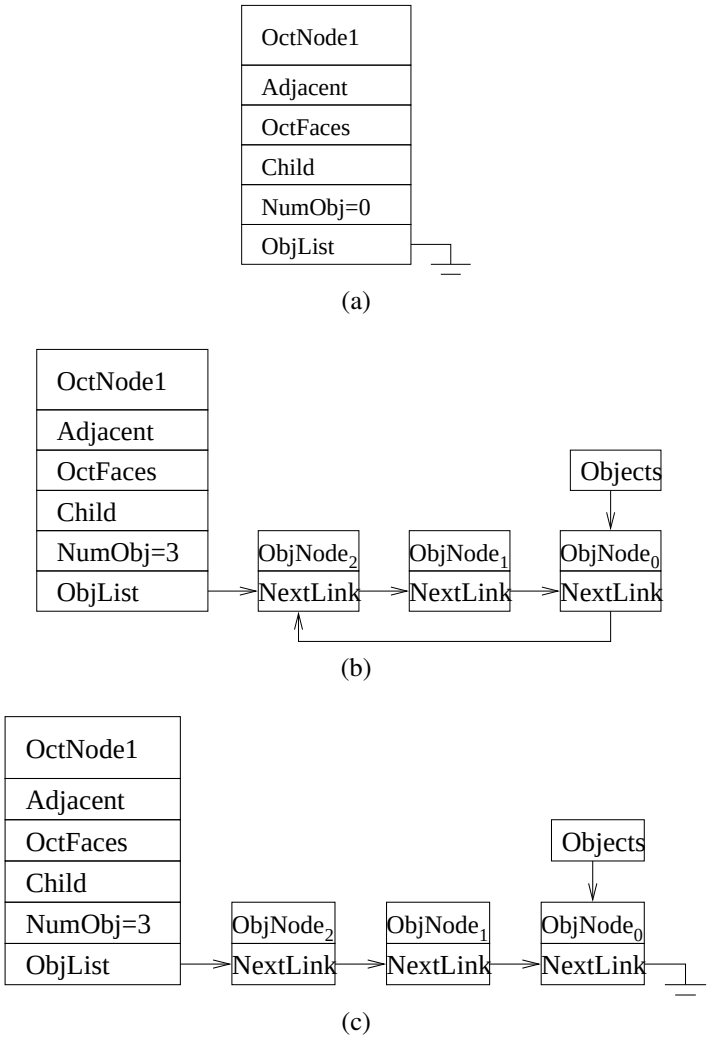


Figure 2.6: `OctNode` structure in RayTrace. (a) An empty ObjList (ObjNum 0).(b) An erroneous list of size 3 containing a cycle introduced by faulty code. (c) Fixed structure after applying Juzi.

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

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 void CreateTree(ObjNode objects, int numObjects) {
67         ObjNode newnode = new ObjNode();
68
69         if (objects != null) {
70         if (numObjects > MaxObj) CreateChildren(objects, 1);
71         else {
72             ObjNode currentObj = objects;
73             ObjNode last = ObjList;
74             while (currentObj != null) {
75                 newnode = new ObjNode(currentObj.GetObj());
76                 if(ObjList == null){
77                         ObjList = newnode;
78                         last = ObjList;
79                 }else{
80                         last.SetNext(newnode);
81                         last = newnode;
82                 }
83                 currentObj = currentObj.Next();
84             }
85             //newnode.SetNext(ObjList); //injected error
```

```
86                  newnode.NextLink = null; //Fix
87                  NumObj = numObjects;
88
89              }
90      }
91  }
```

The method `CreateTree` has a fault in its line 85 that erroneously sets the last `ObjNode` is `NextLink` to the `ObjNode` pointed to by `ObjList`. Figure 2.6(b) 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 2.6(c).

```
newnode.NextLink = null; //Fix
```

2.6 Limitations

The current embodiment of our approach does not fix faults that alter reachability in a data structure since we use Juzi, which only repairs data structures with respect structural invariants that specify properties of the structure reachable from the given root of the erroneous structure. We believe our technique can use specifications richer than structural invariants to generate debugging suggestions for a larger class of faults. For example, in more recent joint-work, we investigated the use of postconditions that relate method pre-state with post-state to correct erroneous implementations [40]; this work leverages a SAT backend. Moreover, if the underlying data structure repair routine (Juzi) can be modified to generate the repair actions to correct other classes of functional errors, our technique will be able to handle more complex program bugs.

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. Our technique addresses faults along one execution path. A method that has multiple independent 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.

2.7 Discussion

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.

2.7.1 Efficient Data Structure Repair

The technique developed in this chapter focuses on automated debugging, but 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. This insight forms the basis of Chapter 4.

2.7.2 Programming by Sketching

Another application is for programming by sketching [96]. 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.

2.8 Summary

This chapter 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 show 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.

Chapter 3

Representing Data Structure Properties using Graph Spectra

This chapter introduces a novel dynamic technique for identifying properties of the program's key data structures. This work addresses a main hindrance in our goal to automate the debugging process: programs do not always provide specifications like `repOk` methods to describe the key structural properties, which are required by both program repair and data structure repair (Figure 2.2); moreover, writing these properties by hand poses a challenge in itself. We borrow classical results from graph theory [22] to characterize the shape of the program's dynamic data structures. Specifically, spectral graph theory, a field that studies the properties of a graph in relation to the properties of matrices based on the graph provides the foundational ideas. The work presented here is based on our ICSE NIER 2011 [68][1] paper, which introduces the idea of using graph spectra for representing data structure properties, and our ICST 2012 [69][2] paper that presents the detailed results.

This chapter first gives an overview, which is followed by an illustrative example to describe the basic idea of our approach. Then, we describe its details as

[1]Muhammad Zubair Malik. Dynamic shape analysis of program heap using graph spectra. *ICSE 2011*.

[2]Muhammad Zubair Malik, Sarfraz Khurshid. Dynamic Shape Analysis Using Spectral Graph Properties. *ICST 2012*.

well as its application to finding bugs using runtime checking. Finally we describe our experimental evaluation.

3.1 Overview

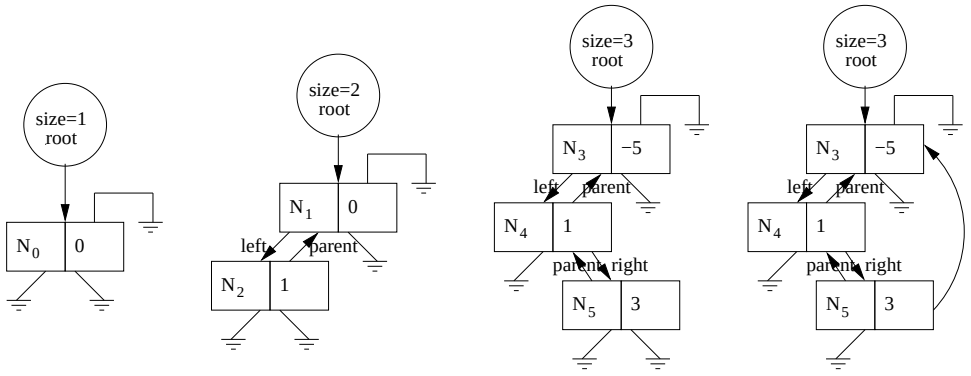
Automated analysis and testing of programs with dynamic data structures requires reasoning about these structures that may have complex structural properties (as discussed before). A number of existing tools can systematically check such programs for *given* structural properties. Shape analysis [77, 89] is a class of techniques that address reasoning about such programs. Traditionally, shape analysis is performed using *static* analysis of the program code. A key motivation behind the use of static analysis is to determine the properties at desired control points for *all* program executions, say for program verification. Shape analysis techniques and other specification-based techniques, e.g., our program repair technique from Chapter 2, require the user to provide structural properties. Recent work introduced *dynamic* techniques for shape analysis, which inspect actual program states to identify key data structure properties without requiring the user to provide them [58,70]. While these techniques do not enable verification for all executions, they enable detecting likely erroneous executions at runtime and promise to be more scalable for finding bugs than techniques based on static analysis.

This chapter introduces a novel dynamic technique, which adapts well-studied results from graph theory to determine the shape of the program's key data structures. We view the object graph that represents a program heap as a mathematical object – an edge-labeled graph, where graph vertices correspond to objects

allocated on the heap and graph edges correspond to fields of these objects [51]. We leverage results from *spectral graph theory* [22] – a field that studies the properties of a graph in relation to the properties of matrices based on it, such as its adjacency matrix or its Laplacian matrix. Specifically, we define properties of recursive data structures using properties of eigenvalues of the associated matrices as well as other graph properties, such as *in-degree* of a vertex.

Our technique builds on our previous work on the Deryaft framework [61, 70] for generating likely representation invariants. Deryaft takes its inspiration from the Daikon invariant detector [35]. In contrast to Daikon, which is a general purpose invariant detection engine, Deryaft focuses on structural properties and as such generates more accurate structural invariants. We follow the general approach introduced by Deryaft for structural invariants: first, identify *core* and *derived* fields of a data structure; and then, check which properties from a pre-defined collection of properties hold for the field values for a given set of program states. The properties that hold for a given set of states are used in two ways: (1) to directly check if a new program state satisfies them; and (2) to generate a representation of the properties as an executable Java predicate, which can be used in a number of ways, e.g., as a runtime assertion or to perform data structure repair [31].

A key advantage of using graph spectra over Deryaft's approach is that, in principle, they allow checking for (violation of) properties that may not be predefined and computed only based on the program states once they are encountered. Thus, graph spectra not only introduce a novel abstraction for properties of program state, but they also enhance our ability to dynamically detect a larger class of errors



(a) (b) (c) (d)

Figure 3.1: This example shows four binary trees with parent pointers. The first three trees (parts (a), (b) and (c)) are valid but the fourth tree (part (d)) has a cycle along the `right` field which breaks the representation invariant.

without requiring the user to provide detailed specifications. As a first step to enable detecting properties that are not directly characterized in spectral graph theory, we conjecture that an invariant learning mechanism using support vector machines [76] may provide a viable solution.

Experimental results using a suite of data structures demonstrate the potential the technique holds in identifying data structure properties and detecting likely erroneous program states.

3.2 Illustrative Example

This section illustrates the working of our invariant generation technique using an example binary tree data structure that additionally maintains min-heap property [20]. We use this example as our running example for the rest of the

(a)	(b)	(c)
left:		
$N_0 \begin{pmatrix} N_0 \\ 0 \end{pmatrix}$	$\begin{matrix} \\ N_1 \\ N_2 \end{matrix} \begin{pmatrix} N_1 & N_2 \\ 0 & 1 \\ 0 & 0 \end{pmatrix}$	$\begin{matrix} \\ N_3 \\ N_4 \\ N_5 \end{matrix} \begin{pmatrix} N_3 & N_4 & N_5 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix}$
right:		
$N_0 \begin{pmatrix} N_0 \\ 0 \end{pmatrix}$	$\begin{matrix} \\ N_1 \\ N_2 \end{matrix} \begin{pmatrix} N_1 & N_2 \\ 0 & 0 \\ 0 & 0 \end{pmatrix}$	$\begin{matrix} \\ N_3 \\ N_4 \\ N_5 \end{matrix} \begin{pmatrix} N_3 & N_4 & N_5 \\ 0 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 0 \end{pmatrix}$
parent:		
$N_0 \begin{pmatrix} N_0 \\ 0 \end{pmatrix}$	$\begin{matrix} \\ N_1 \\ N_2 \end{matrix} \begin{pmatrix} N_1 & N_2 \\ 0 & 0 \\ 1 & 0 \end{pmatrix}$	$\begin{matrix} \\ N_3 \\ N_4 \\ N_5 \end{matrix} \begin{pmatrix} N_3 & N_4 & N_5 \\ 0 & 0 & 0 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix}$
left+right:		
$N_0 \begin{pmatrix} N_0 \\ 0 \end{pmatrix}$	$\begin{matrix} \\ N_1 \\ N_2 \end{matrix} \begin{pmatrix} N_1 & N_2 \\ 0 & 1 \\ 0 & 0 \end{pmatrix}$	$\begin{matrix} \\ N_3 \\ N_4 \\ N_5 \end{matrix} \begin{pmatrix} N_3 & N_4 & N_5 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 0 \end{pmatrix}$
parent+left+right:		
$N_0 \begin{pmatrix} N_0 \\ 0 \end{pmatrix}$	$\begin{matrix} \\ N_1 \\ N_2 \end{matrix} \begin{pmatrix} N_1 & N_2 \\ 0 & 1 \\ 1 & 0 \end{pmatrix}$	$\begin{matrix} \\ N_3 \\ N_4 \\ N_5 \end{matrix} \begin{pmatrix} N_3 & N_4 & N_5 \\ 0 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 1 & 0 \end{pmatrix}$
'$<$' along left+right:		
$N_0 \begin{pmatrix} N_0 \\ 0 \end{pmatrix}$	$\begin{matrix} \\ N_1 \\ N_2 \end{matrix} \begin{pmatrix} N_1 & N_2 \\ 0 & 1 \\ 0 & 0 \end{pmatrix}$	$\begin{matrix} \\ N_3 \\ N_4 \\ N_5 \end{matrix} \begin{pmatrix} N_3 & N_4 & N_5 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 0 \end{pmatrix}$
Spectra of left+right directed adjacency matrix:		
$\lambda_1 \begin{pmatrix} N_0 \\ 0 \end{pmatrix}$	$\begin{matrix} \lambda_1 \\ \lambda_2 \end{matrix} \begin{pmatrix} N_1 \\ 0 \\ 0 \end{pmatrix}$	$\begin{matrix} \lambda_1 \\ \lambda_2 \\ \lambda_3 \end{matrix} \begin{pmatrix} N_3 \\ 0 \\ 0 \\ 0 \end{pmatrix}$

Figure 3.2: The matrix representations computed along various fields for the three valid trees from Figure 3.1 (a),(b) and (c). The matrix along `left` and `right` is the one that ensures reachability and is used for detecting most of the global properties. The matrices computed along derived fields are used primarily to check local properties of the structures. Spectrum for acyclic trees is a zero vector – however, this property more generally holds for directed acyclic graphs, not just trees. To identify the sub-class of trees, we additionally use the property that in-degree of any vertex is ≤ 1.

chapter. Consider the following class declaration:

```
1     class BinaryTree {
2         Node root;
3         int size; // number of nodes in the tree
4
5         static class Node {
6             int key;
7             Node left;
8             Node right;
9             Node parent;
10        }
```

A binary tree object has a `root` node; each node has a `left` and a `right` child node, a `parent` node, and an integer `key`. The *structural integrity constraints*, which are also termed *representation invariants*, are: acyclicity along `left` and `right`, and correctness of parent-child relationship and of the `size` value, as well as the min-heap property, i.e., the `key` in a node is smaller than those in its children[3].

The rooted binary tree has three recursively declared fields. However, in all *positive instances* of the tree, i.e., valid trees, we can reach all connected parts of the structure from the `root` using only the two fields `left` and `right`. These are called the *core-fields* [70]; they are useful in detecting various global and local properties. For each reachable node, the algorithm builds a *directed* adjacency matrix along each reference field. The matrices formed using the core-fields are the basis of the structure, and the structure core is computed using matrix summation. This basic representation is used to derive other matrix representations (such as Laplacian) that are used in spectral graph theory to detect a number of properties.

[3]Our example structure is different from the *binary heap* data structure, which additionally maintains a *complete* binary tree

$$left + right : \begin{array}{c} \\ N_6 \\ N_7 \\ N_8 \end{array} \begin{array}{c} \begin{array}{ccc} N_6 & N_7 & N_8 \end{array} \\ \begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{pmatrix} \end{array} \quad spectra : \begin{array}{c} \\ \lambda_1 \\ \lambda_2 \\ \lambda_3 \end{array} \begin{array}{c} \begin{array}{c} N_6 \end{array} \\ \begin{pmatrix} 1 \\ -0.5 \\ -0.5 \end{pmatrix} \end{array}$$

Figure 3.3: This example shows the matrix representation of the invalid tree from Figure 3.1(d) that has a cycle. The spectra of this tree form a non-zero vector.

The input to our technique is concrete structures, such as the valid binary trees as shown in Figure 3.1 (a),(b) and (c). The technique takes a graph view of the heap, identifies the core-fields and abstracts the heap-state to matrix form. Figure 3.2 gives a traditional $|V|^2$ representation of the matrices, where V is the set of nodes in the structure. Each entry that is 1 in the matrix corresponds to an edge in the corresponding structure, while a 0 entry represents lack of an edge.

For each of the given structures, the property detection algorithm first checks top-level properties in the dictionary hierarchy, which narrows the search to relevant properties. For example, in this case, the algorithm will not check for *girth*, which requires *circularity*, since it is determined that the structure is acyclic. Figure 3.2 (bottom row) demonstrates that all acyclic structures (Figure 3.1 (a), (b) and (c)) along core fields have similar spectra – all their eigenvalues are zero when represented as directed adjacency matrices. When a cycle is introduced in one of the trees (Figure 3.1 (d)) the spectra form a non-zero vector, a property that is used to detect a cyclic structure. The other relevant tree-properties in the dictionary of rules include cardinality constraints for which integer values in the object holding root pointer are compared with cardinality of the set of nodes in the tree.

The algorithm next checks local properties. First, it checks for *symmetry* along various fields. The directed adjacency matrix along `parent+left` is not symmetric; other field combinations also fail except `parent+right+left`, which is symmetric. Note that this property is implied by the correctness of `parent` pointers, i.e., $parent$ is transpose of $left + right$ for structures that are trees along `left` and `right` fields.

Next, it checks for arithmetic relations among values in nodes: if a matrix generated by applying comparison relationships $<, \leq, =, \geq, >$ along a set of fields is equal to the directed adjacency matrix along the same set of fields then the comparison relationship holds along that fields. For the binary trees in Figure 3.2 '$<$' holds along `left` and `right`, because positive instances maintain a min-heap property.

We write the properties that hold for all positive instances as a Java predicate function, named `repOk`, which can, in principle, be used in a number of analyses (e.g., runtime checking using assertions and test input generation [13]). The following code shows parts of the `repOk` method for the binary tree example, which uses the matrix library JAMA [82] for basic matrix operations:

```
1   boolean repOk() {
2      Matrix m = Matrix.buildDirectedAdjacency (
3                     this,new String[]{"left","right"});
4      if (!acyclicCore(m))
5        return false;
6      if (!sizeOk(m,size))
7        return false;
8      if(!symmetric(root,
9                  new String[] {"parent","left","right"}))
10       return false;
```

```
11      if(!lessAlong(root,
12                    new String[] {"left","right"},"key"))
13        return false;
14      return true;
15    }
16    //Graph spectral rule for acyclicity
17    boolean acyclicCore(Matrix m){
18      return m.spectra().equals(Matrix.zeros(m.dim)) &&
19        maxInDegree(m) <= 1;
20    }
21    //Graph matrix property for size
22    boolean sizeOk(Matrix m,int cardinality){
23      return m.dim==cardinality;
24    }
25    //Graph matrix property for symmetry
26    boolean symmetric(
27                              Object root,String fields[]){
28      Matrix m = Matrix.buildDirectedAdjacency(
29                                              root,fields);
30      //check m[i][j] == m[j][i] for all i,j
31      return m.symmetric();
32    }
33    //Less than along a set of fields
34    boolean lessAlong(
35                          Object root, String[] f,String val){
36      Matrix m1 = Matrix.buildAdjacencyAlongFields(root,f);
37      Matrix m2 = Matrix.buildLessThanAlong(root,f,val);
38      return m1.equals(m2);
39    }
```

3.3 Technique

This section describes our spectra-based technique for detecting structural invariants. We take an abstract view of the program heap as an edge-labeled graph [51, 52, 59, 74]. Our technique uses a partitioning of the set of object fields into *core*

fields and *derived* fields [70]. Given a set of *positive instances* (i.e., structures whose properties are to be detected), our technique constructs the relevant matrices based on these structures to enable property detection based on graph spectra [22] using an iterative algorithm. To enable detecting properties that are not captured by spectra, we conjecture that an invariant learning mechanism using support vector machines [76] may provide a viable solution.

3.3.1 Program Heap as an Edge-Labeled Graph

We take a relational view of the program heap and view the heap of a Java program as an edge-labeled directed graph whose nodes represent objects and whose edges represent fields. For languages, such as C and C++, that allow pointer arithmetic and arbitrary conversions between integer values and memory addresses, a different view would be needed. However, for type-safe subsets of such languages, the relational view applies. The presence of an edge labeled f from node o to v says that the field f of the object o points to the object v or has the primitive value v. Mathematically, we treat this graph as a set of vertices and a collection of edges, one for each field. We partition the set of vertices according to the declared classes and partition the set of edges according to the declared fields. We represent `null` by the absence of the edge. A particular program state is represented by an assignment of values to these sets and relations. Since we model the heap at the concrete level, there is an isomorphism between program states and assignments of values to the corresponding sets and relations. The model for our `BinaryTree` example consists of three sets, each corresponding to a declared class or primitive type

`BinaryTree, Node, int` and six relations corresponding to a declared field:

```
root:BinaryTree x Node
size: BinaryTree x int
key : Node x int
left: Node x Node
right: Node x Node
parent: Node x Node
```

We assume (without the loss of generality) that each structure in the given set has a unique root pointer. Thus, the abstract view of a structure is a rooted edge-labeled directed graph, whose properties are detected based on its reachability.

3.3.2 Core and Derived Fields

Following our previous work on structural invariant generation [70], we partition the set of reference fields declared in the classes of objects in the given structures (i.e., positive instances) into two sets: core and derived. For a given set, S, of structures, let F be the set of all *reference* fields.

Definition 1. *A subset $C \subseteq F$ is a core set with respect to S if for all structures $s \in S$, the set of nodes reachable from the root r of s along the fields in C is the same as the set of nodes reachable from r along the fields in F.*

In other words, core set preserves reachability in terms of the set of reachable nodes. Indeed, the set of all fields is itself a core set. We aim to identify a *minimal* core set, i.e., a core set with the least number of fields.

To illustrate, the set containing both the reference fields `left` and `right` in

the example from Section 3.2 is a minimal core set with respect to the given set of trees.

Definition 2. *For a core set C the set of fields $F - C$ is the derived set.*

Our partitioning of the reference fields is inspired by the notion of back-bone in certain data structures.

3.3.3 Matrix Representation of Heap

Spectral graph theory [22] characterizes properties of graphs in terms of their *spectra*: the *spectrum* of a graph is based on the *eigenvalues* of its adjacency matrix. The properties are largely defined in terms of directed adjacency matrix, undirected adjacency matrix, and Laplacian matrix representations. Our technique primarily uses directed adjacency matrix representation.

In the following discussion, we denote a graph by $G = (V, E)$ where V is the set of nodes and $E \subseteq V \times V$ is the set of edges. The *degree* of a vertex u is the number of edges connected to u and is denoted by d_u. The *in-degree* of a vertex is the number of edges incident on the vertex.

The directed adjacency matrix representation of a graph is given by:

$$B(u,v) = \begin{cases} 1 & (u,v) \in E, \\ 0 & otherwise \end{cases}$$

Since we view the program heap as a directed graph, the adjacency matrix may not be symmetric.

The undirected adjacency matrix A, which is always symmetric, can be derived from the directed adjacency matrix representation:

$$A(u,v) = \begin{cases} 1 & B(u,v) = 1 \; or \; B(v,u) = 1, \\ 0 & otherwise \end{cases}$$

Note that adjacency matrices on recursive fields are always *square*, i.e., have the same number of rows and columns.

For a square matrix A, a non-zero vector v is an *eigenvector* if $Av = \lambda v$ for some scalar λ. The scalar λ is termed the *eigenvalue* corresponding to v. The eigenvalues are solutions to the equation $|A - \lambda I| = 0$, where I is identity matrix.

3.3.4 Properties of Interest

Following our previous work on structural invariant generation [70], we consider global as well as local properties of rooted edge-labeled directed graphs as likely representation invariants structurally complex data. The properties are divided into the following categories:

- Global reference field properties, which include properties on the shape of the structure reachable from the root along some set of reference fields.
- Global primitive field properties. In reasoning about graphs, the notion of a cardinality of a set of nodes occurs naturally, e.g., to cache the number of nodes reachable from a root pointer. We consider properties relating values of integer fields and cardinalities of sets of reachable objects.

Table 3.1: Properties identified using directed adjacency matrix representation. Let $A = (a_{i,j})$ for $1 \leq i, j \leq n$ be an $n \times n$ adjacency matrix. Let λ be an eigenvalue of A.

Property	*Rule*
Directed acyclic (DAG)	$\forall \lambda\ , \lambda = 0$
Tree	*DAG* $\wedge\ \forall j\ \sum_i a_{i,j} \leq 1$
Circular	*girth* = n
Symmetric	$\forall i, j\ a_{i,j} = a_{j,i}$
Min-heap	*less-than* = A
SizeOk	`size` = n

- Local reference field properties. We consider local properties that relate different types of edges, e.g., the transpose relationship.

- Local primitive field properties. We check for order (e.g, less than) rules for values in nodes connected by an edge. To enable the use of matrix algebra, we define a *relative ordering matrix* where an entry $m_{i,j} = 1$ iff there is an edge from node i to node j and the integer values in nodes i and j satisfy the corresponding ordering relation (e.g., $value(i) < value(j)$).

Table 3.1 presents a list of rules that we apply to detect properties of graphs using directed adjacency matrices that are built using our abstract view of the program heap. A *directed acyclic graph (DAG)* has all eigenvalues of its directed adjacency matrix equal to zero. A *tree* is a DAG where the *in-degree* of each vertex ≤ 1. For a *circular* structure (along one field), the *girth* of its graph, i.e., the length of the shortest cycle in the graph, equals the number of vertices in the graph. The *transpose* relationship between certain data structure fields, such as `previous` and

`next` for doubly-linked lists, or `parent` and `left + right` for binary trees, is detected based on symmetry of corresponding directed adjacency matrices. For a *min-heap*, the *less-than* matrix $H = (h_{i,j})$, which is defined as $h_{i,j}$ = 1 iff there is an edge from node i to node j and the integer value in node i is less than the integer value in node j, equals the directed adjacency matrix. A *max-heap* can similarly be characterized. Structures with a *top-level* integer field, such as `size` in binary tree are checked to see if the value of that field equals the number of reachable nodes.

3.3.5 Algorithm

Algorithm 1 gives the pseudo-code of our approach. It first computes the data structure backbone through core-field analysis. In the beginning it assumes that all properties are valid but gradually keeps narrowing its search to only properties of interest that hold on given structures. Once the algorithm has identified all valid properties, a `repOk`, which only checks for these properties is created.

Our dictionary is organized to minimize checks. We use a hierarchical design and utilize order between various properties to reduce checks. For example, it is wasteful to check girth of an acyclic structure. Similarly, there is no need to check for treeness in a structure that violates the DAG property.

3.3.5.1 Learning new Properties

To enable new properties to be detected and checked using graph spectra, we envision the use of machine learning techniques. We conducted an initial investigation into learning the *height balance* property in trees by training a support

input : $\mathscr{C}$ – Data structure declaration
input : $\mathscr{T}$ – Valid Structures
input : $\mathscr{D}$ – Dictionary of p, where $p \in$ graph properties
output: `repOk`, a Java predicate representing invariant properties

$\mathscr{F} \leftarrow coreFields(\mathscr{C}, \mathscr{T})$;
$\mathscr{V} \leftarrow \mathscr{D}.getAllProperties()$;
for $\forall t \in \mathscr{T}$ **do**
  for $\forall p \in \mathscr{V}$ **do**
    $Fields\ f[\,] \leftarrow p.getRequiredFields(\mathscr{F})$;
    $matrix \leftarrow Matrix.build(t.root, f)$;
    if $!matrix.statifies(p)$ **then**
      if $p.hasChildren()$ **then**
        $\mathscr{V}.removeSubHeirarchy(p)$;
      end
      $\mathscr{V}.remove(p)$;
    end
  end
end
$\mathscr{V}.minimizeRules()$;
`repOk` $\leftarrow \emptyset$;
for $\forall p \in \mathscr{V}$ **do**
  $code \leftarrow p.synthesize()$;
  `repOk`$.append(code)$;
end

Algorithm 1: Invariant generation

Table 3.2: Results for subject data structures (of size ≤ 5)

Benchmark	Structures Generated	TP	TN	FP	FN	`repOk` Time (ms)	
						Manual	Spectral
Singly-linked acyclic list	7776	24	7752	0	0	0.02	0.11
Singly-linked circular list	7776	24	7752	0	0	0.01	0.01
Doubly-linked circular list	60466176	24	60466152	0	0	0.01	0.01
Binary tree	60466176	1008	60462888	2280	0	0.01	0.02
DAG (binary)	60466176	32712	60433464	0	0	0.01	0.01

vector machine [76] using positive and negative instances. The accuracy of the rule learnt was better than a chance classifier, which is an encouraging result. We believe the *numeric* encoding of graph properties using spectra will enable future work to develop novel applications of machine learning techniques in more accurately detecting erroneous program executions.

3.4 Application: Dynamic Shape Analysis using Graph Spectra

Our work on using graph spectra to represent properties of dynamic data structures provides a new approach for dynamic shape analysis [58]: record the spectra at control points of interest for representative executions and then verify the spectra for future executions to check their validity.

3.5 Evaluation

In this section we present our experiments designed to address the following research questions:

1. Are the properties detected by our approach comparable to those written manually as predicates?

2. How well does our approach disambiguate *Trees*, *DAGs* and *Cycles* ?

3. Are the properties detected useful in finding bugs in software?

3.5.1 Comparing deteced properties with those written manually as predicates

We conduct this experiment as a basic sanity check and to investigate the sources of error in the properties that are detected using graph spectra. We compare properties generated by our approach with manually written predicates based on previous work [13]. Graph spectra are computed using operations on real numbers; to allow for errors in representing reals using oatingpoint number, we check values to lie within a small threshold (10^5) of the expected value.

Our experimental setup uses an exhaustive generator, which enumerates all (valid and invalid) structures of a class within a given size. Given the declaration of a recursive data structure consisting of nodes, our structure generator enumerates for a set of all recursive fields F , all possible field assignments for each field for a given set of node objects O to the same set of nodes O and the literal $null$. We fix the value of the $size$ field to the number of nodes; thus, a correct `repOk` will only accept structures with exactly n nodes. For this implementation all objects are uniquely labeled, which allows us to sequentially permute all possible field assignments.

We evaluate the validity of each of these structures against two `repOk` functions, one based on the properties detected by our approach using graph spectra and the other written manually (oracle). We define *true positive* (TP) to mean that if

oracle accepts a structure and so does the spectral `repOk`. Similarly, *true negative* (TN) means that when oracle rejects a structure and so does the spectral `repOk`. We define *false positive* (FP) when the oracle rejects a structure but the spectral `repOk` admits it. We define *false negative* (TN) when oracle accepts a structure but the spectral `repOk` rejects it. The accuracy of the approach is the proportion of true results in the population: $accuracy = \frac{TP+TN}{TP+FP+FN+TN}$.

A possible source of errors in the result of spectral repOk is lack of precision due to finite representation of numbers. For example, on a 32-bit machines addition rule valid for all $x \in \mathbb{Z}$: $x+1 > x$, does not hold for integers; similarly the addition rule for all $x \in \mathbb{R}$: $x+1 \neq x$, breaks down for IEEE 754 oating point number representation. In general, limited precision binary machines cannot precisely represent fractions if 2 is not a prime factor of the number. Spectral rules require many oating point computations and suffer from lack of precision in computation. In these experiments we have used an error bound of 10^5 for comparing equality of numbers.

Another source of possible errors in detecting properties using graph spectra is the encoding of data structures that reside on the program heap using adjacency matrices that may represent connectivity along a set of fields (e.g., `left` and `right`) and hence lose the distinction between `left` pointers and `right` pointers.

Table 3.2 tabulates the experimental results for five subject data structures: singly-linked acyclic lists, singly-linked circular lists, doubly-linked circular lists, binary trees, and directed acyclic graphs where each node has a left and a right child. We use our exhaustive structure generator to generate all possible structures

with up to 5 nodes.

For all chosen subjects except binary tree, the result of spectral `repOk` matches exactly the result of the oracle. For binary tree, we observe 2280 false positives. These are all due to the imprecision in the adjacency matrix representation where the distinction between `left` and `right` fields is lost. More specifically, if a node i points to another node j along `left` as well as along `right`, the adjacency matrix A will represent the two edges with a single edge: $a_{i,j} = 1$.

The running time of `repOk` using graph rules is comparable with the oracle for these structures with up to 5 nodes in majority of the cases. However, for larger structures, we expect the oracle to run substantially faster than spectral `repOk` due to the complexity of the underlying matrix operations. We expect an incremental approach for updating spectra to provide a practical basis for the use of graph spectra in real applications.

3.5.2 Is it a tree, DAG or a cycle?

A key question in traditional shape analysis is to see if the shape of heap-allocated data structure is a tree, DAG, or a cyclic graph [38]? While such analysis is traditionally done statically and is required to be safe, our approach provides an (unsafe) way to detect the shape dynamically based on the observed executions of the program.

The last two rows of Table 3.2 show how our approach performs empirically in correctly identifying trees, DAGs, and cyclic structures, where each node has two labeled children (`left` and `right`). If such a structure has a directed cycle, it is

always correctly identified (FP=0 for DAG). If the structure has an undirected cycle, it may not be correctly identified, and instead may be classified as a tree (FP>0 for binary tree) – thus a DAG, which is not a tree, may be classified as a tree. If the structure is a tree, it is always classified correctly.

3.5.3 Error detection

While many programming errors are discovered before deployment, some may only be encountered after deployment. These errors can cause run-time failures, resulting in security violations and increased down time. Many systems that have high security and availability requirements may need to perform run-time checking for timely detection of errors and applying possible corrective actions. We evaluate our approach for capturing run-time errors in code adapted from two case studies: `RayTrace` which is part of `SPECjvm` [21] and `ANTLR` which is taken from `DaCapo` [11] benchmark. In the setup of this experiment, first we detect properties of the valid heap structures generated by the selected code from each benchmark program. Then, we execute a faulty version of the code to generate invalid structures and check if spectral `repOk` identifies it. These errors in the code are manually injected based on our understanding of how the code works.

The errors detected and properties observed clearly depend on the capabilities of our approach to observe them and the fault injection model. For example if the fault injection model only mutates object reference fields it is less likely for our approach to detect error in object keys.

3.5.3.1 ANTLR

ANTLR [84] is a language tool that generates recognizers, compilers, and translators from grammatical descriptions. We used it for our case study in Chapter 2 and provided a detailed description of its key data structure `CommonTree` in Section 2.5.2.2. We use the same example to test the error detection ability of spectral `repOk`.

Recall that the relevant representation invariants of `CommonTree` are acyclicity along children and transpose relationship between parent and children. While these are standard properties of trees with parent pointers, an adjacency matrix representation cannot be built directly based on the given data declarations because `children` is a field of declared type `java.util.List`. Our implementation only considered recursively declared fields during matrix translation, it required us to write a dedicated matrix translator for the given non-parameterized lists. However, now with parameterized lists `List<CommonTree>` we allow adjacency lists of recursively declared data types as general graph representation.

We use the method `addChild` from Section2.5.2.2 with the same injected error. The erroneous example we consider adds another `CommonTree` that has its own children but has no payload to the given `CommonTree` instance by adopting all its children. The injected fault omits the update of parent pointers in the adopted nodes thus violating parent-child transpose relationship. The spectral `repOk` correctly detects this error.

3.5.3.2 RayTrace

RayTrace [21] is a Java program that produces realistic graphics by tracing path of light through pixels in an image. We introduced it in Section 2.5.2.3 of Chapter 2. As discussed earlier, RayTrace maintains the 3D world model of the image in a structure `OctNode`. The `OctNode` maintains object list `ObjList` composed of object nodes `ObjNode` to record the objects inside the space.

The `ObjList` implements an acyclic list whose cardinality is cached in `NumObj` field of `OctNode`. We revisit the `CreateTree` example from Section 2.5.2.3 where a cycle was introduced in the list by erroneous code. The spectral `repOk` successfully detects the bug in the state of RayTrace like the manual `repOk`.

3.6 Discussion

This section discusses the mathematical basis, possible implementation optimizations, as well as uses and limitations of our approach. We also briefly compare our approach to previous work on using dynamic analysis for detecting properties of recursive data structures.

3.6.1 Basis of our approach

We view the heap of recursively declared data structures as edge-labeled graphs represented using matrices. Our basic observation is that invariants of the heap that a program updates are related to the properties of matrices. We have used this simple yet elegant observation to build an invariant detector for programs that

operate on structurally complex data.

3.6.2 Fast Computation of Matrix Operations

Matrix operations required for computation of graph spectra can be relatively expensive for non-small data structures. For example, a traditional `repOk` might only perform one traversal of the structure to check for acyclicity whereas a spectral `repOk` may have to perform an operation of polynomial complexity in the size of structure. This problem can be addressed in two ways: 1) by using optimized libraries Basic Linear Algebra Subprograms (BLAS) [83] for various matrix computations; 2) many matrix operations that we perform are repeated over the same set of values, therefore we can also exploit memoization strategies to implement an incremental technique for computing spectra – one such possibility is to use Cuppen's divide-and-conquer algorithm for calculating eigenvalues [87].

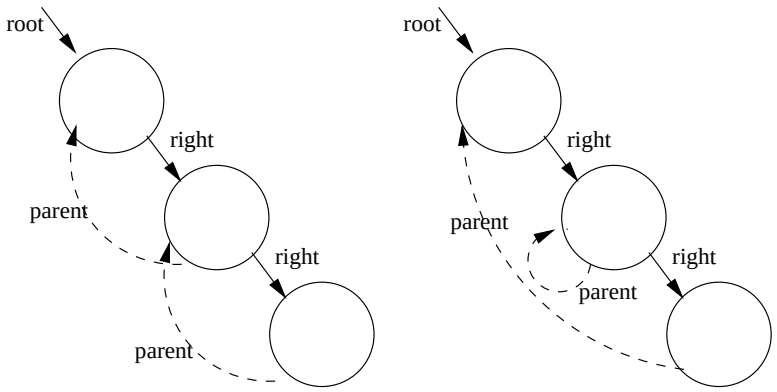
3.6.3 Uses

Two software checking techniques enabled by our approach in addition to runtime checking are test input generation [13] and error recovery using data structure repair [31]. The spectral `repOk` could be produced using a small number of manually created structures that satisfy desired properties. The `repOk` could then be used by a test generation tool, such as Korat [13] to enumerate desired test inputs. It could also be used by a data structure tool, such as Juzi [32], to mutate erroneous program states to satisfy structural constraints. How tools such as Korat and Juzi perform using spectral `repOk`'s requires further investigation.

We conjecture it may even be possible to apply our dynamic analysis in the context of parallelizing compilers where traditional shape analysis has been used [38]. Traditionally, the goal of shape analysis has been to statically determine the shape of data structures using formal reasoning by relating locally visible variables on the stack to dynamically allocated variables on the heap. An optimizing and parallelizing compiler can use this information to apply optimizing transformations such as loop unrolling, null pointer dereference detection, improved pipelining or dead code elimination. Basic structural information such as the knowledge that a local variable points to a *tree-like* structure can give compiler the clue that memory regions accessed through different fields of the variables are disjoint and may be processed in parallel. Similarly, for a *DAG* like structure traversing along different fields does not guarantee disjointness, however, such guarantees may be given for subsequences of the accessible links. Finally, for cyclic structures disjoint substructures are not possible. We conjecture that in contrast to the traditional use of static analysis, compilers may be able to use the statistics from the profile runs of a program to perform optimized recompilation of the program, which may allow shape information collected using our dynamic shape analysis technique to improve efficiency of compiled code.

3.6.4 Limitations

Detecting invariants dynamically has two inherent limitations [35]: not all invariants of observed structures can be detected; and invariants that are detected may be violated by a valid structure not observed thus far. However, from a practical



(a) (b)

Figure 3.4: (a) A valid tree and (b) an invalid tree (invalid parent pointers). The use of degree metrics alone [58] is unable to distinguish between the two structures.

perspective, the approach can lay the foundation of a useful tool that assists in writing correct programs.

3.6.5 Comparison with previous work on dynamic analysis for invariant detection

Our previous work on Deryaft [70] introduces a dynamic approach for detecting structural invariants by checking a (fixed but extensible) collection of invariants for a given set of program states. Similar to our approach in this paper, Deryaft enables runtime checking, test input generation, and runtime error recovery using data structure repair. The key novelty of our approach in this paper is the use of graph spectra to abstract data structure properties. Moreover, the use of spectra is more general than checking a fixed set of properties, since the values of eigenvalues could themselves be used as a check.

The ShapeUp framework [58] presents a dynamic approach to shape analysis using *degree metrics*, which summarize the in-degrees and out-degrees of nodes in recursive structures. While degree metrics provide a lighter-weight mechanism than graph spectra for dynamic shape analysis, degree metrics are not sufficient to identify certain errors, such as certain incorrect parent pointers in a binary tree. Figure 3.4 illustrates such an error. Each node has the same in-degree and out-degree in both the structures. Therefore, using degree metrics, the two structures have the same abstract representation and are indistinguishable. In contrast, our approach detects such errors in binary trees with parent pointers.

3.7 Summary

Spectral graph theory explores the properties of a graph in relation to the properties of the matrices representing the graph, e.g., eigenvalues of its adjacency matrix. In this chapter we viewed the program heap as an edge-labeled graph and defined the rules based on graph spectra to characterize data structure properties. Our experiments on a suite of text book data structures showed that graph spectra can characterize these data structures correctly and can detect violations of structural properties.

Chapter 4

Repair Abstractions

Systematic data structure repair techniques [32, 112] allow programs to repair from erroneous state by performing a bounded exhaustive search to find the correct structure. Our basic observation is that many program errors do not happen purely at random, rather they are caused by a specific source in the system, such as a faulty method, and in cases where such errors recur we can have more efficient techniques that re-use work performed during repair.

This chapter is based on our work published at RV 2013 [113][1].

4.1 Overview

We introduce *repair abstractions*, which capture the essence of how certain data structure corruptions are repaired by specific actions of a data structure repair routine, such as Juzi [32], and allow more efficient repair of errors that recur. Recall the data structure repair problem: given a structure s and a method $repOk$ that represents desired structural integrity constraints of the structure s such that s does not satisfy $repOk$, perform repair actions on s to transform it into a structure s

[1] Razieh Nokhbeh Zaeem, Muhammad Zubair Malik, Sarfraz Khurshid. Repair Abstractions for More Efficient Data Structure Repair. *RV 2013*. (Zaeem is a former UT student supervised by Khurshid.)

such that it satisfies $repOk$ [32]. Juzi performs repair actions to mutate a corrupt data structure into a valid one; each repair action is a triple $< o, f, v >$ that sets a field f of a object o to value v. Juzi performs a systematic search to compute repair actions. The goal of abstract repair is to avoid this search by abstracting and memoizing repair actions for future use. Conceptually, a repair abstraction is a tuple $(field, action)$ where $action$ is an *abstract* repair action performed when $field$ (of some object) in the program state needs repair.

Our approach to repair abstractions consists of two key steps: (1) building a repair abstraction based on a concrete repair action; and (2) applying the repair abstraction when the same error is encountered again. We describe the central idea of our approach by relating it to the search-based repair performed by Juzi. The basic Juzi algorithm [60] repeatedly invokes $repOk$ on structures that are candidates for the output (i.e., repaired) structure. During each invocation of $repOk$ the repair algorithm performs two key steps: (1) it monitors the order of field accesses, (2) and if $repOk$ returns false, non-deterministically updates the value of the last field accessed — if all values have been checked, it systematically backtracks to update the value of the second last field accessed and so forth. The algorithm terminates when the structure is fixed or the (bounded) search space is exhausted.

Our approach using repair abstractions integrates with the basic Juzi algorithm as follows. Every time Juzi finds a correct fix for a constraint violation, our approach computes an abstraction for the repair based on simple rules, for example, if Juzi repaired the structure by assigning a field f to `null` then repair abstraction records $(f, Null)$ (i.e., if f needs to be mutated, set it to `null`) as an abstract

repair action, thereby prioritizing it when a future execution encounters the same error – even if the underlying repair routine would have first tried a non-null value according to its default search.

Repair abstractions offer two key advantages. One, they allow summarizing concrete repair actions into intuitive descriptions of how certain errors in data structures were fixed, which helps developers understand and debug faulty program behaviors (when the errors in state were due to bugs in code). Two, they allow a *direct* re-use of repair actions *without* the need for a systematic exploration of a large number of candidate structures (as is performed by Juzi) when the same error appears in a future program execution, e.g, due to the execution of a faulty statement in code. We have implemented this approach in a tool that we call AbstractJuzi. Experimental results using a suite of complex data structures show that repair abstractions allow more efficient repair than previous techniques.

4.2 Motivating Example

Recall the doubly-linked list example from Chapter 1 with the faulty `addFirst` method. Consider the following code segment which shows two invocations of the faulty method such that after each invocation the structural constraints are checked and repair is performed using the `Repair.assertrepair` method, which checks the list class invariant (by invoking `repOk`) and repairs the list if the invariant is violated:

```
LinkedList l = new LinkedList();
l.addFirst(0); Repair.assertrepair(l);
l.addFirst(1); Repair.assertrepair(l);
```

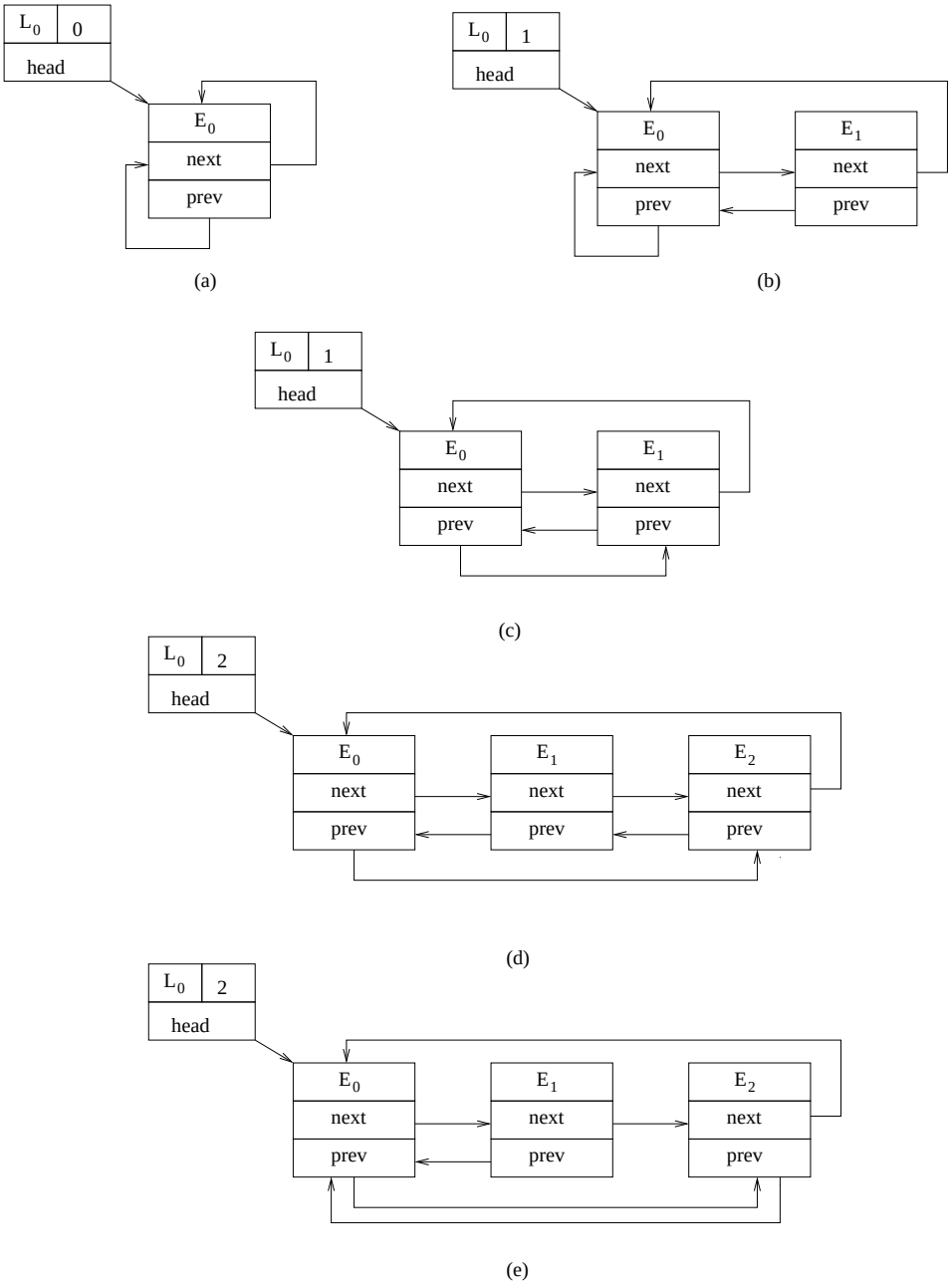


Figure 4.1: Doubly-linked circular list with sentinel `head`. (a) An empty list (size 0). (b) An erroneous list of size 1 containing element E1. (c) List in part-b is repaired by Juzi and the rule $(prev, Neighbor)$ is learnt (d) An erroneous list of size 2 containing elements E1 and E2.(e) List after applying repair action $< E1, prev, E2 >$ that is obtained directly from the rule $(prev, Neighbor)$.

Figure 4.1 shows the pre-state and post-state for each of the two invocations of `addFirst` as well as the repaired state after the final invocation. Note that the post-state for the each invocation is repaired to form the pre-state for the next invocation.

If we apply the standard Juzi algorithm [31] to repair the list after the first addition, it has two iterations to fix the value of `E0.prev`: `null` and `E1`. To repair the list again after the second addition, the standard Juzi algorithm would again systematically search for a valid value for the corrupt `prev` field.

Our approach captures the essence of successful repair action after the first addition using the repair abstraction $(prev, Neighbor)$, which indicates that if `prev` of a node n is mutated, try first as the destination a *neighbor*, i.e., a node that is connected along one edge to n. By building this abstraction after Juzi first performs its default search-based repair, we re-use the abstraction by prioritizing the repair actions to check setting `prev` to a neighbor. This significantly reduces the repair cost when similar errors are encountered in the data structure again.

4.3 Framework

Figure 4.2 gives an overview of *AbstractJuzi*, our repair abstraction framework, which leverages Juzi's systematic search together with repair abstractions. Given a corrupt data structure and a corresponding `repOk` method, AbstractJuzi performs repair using Juzi's core functionality modified in three key ways:

- *Repository* of repair abstractions. AbstractJuzi creates and maintains a repos-

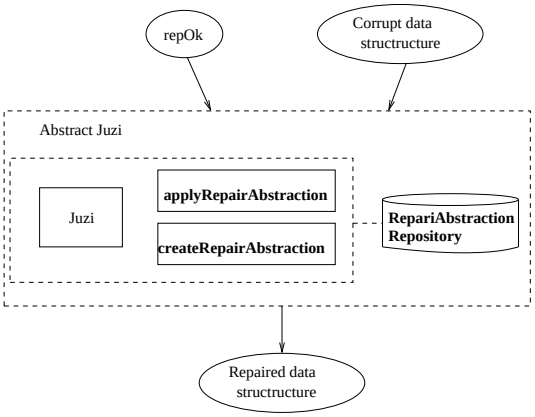


Figure 4.2: The AbstractJuzi repair framework.

itory of abstract repair actions, which are applied when needed. The repository grows as more concrete repairs are abstracted.

- *Creation* of repair abstractions. When Juzi performs a concrete repair that is not based on an existing repair abstraction, AbstractJuzi creates a new repair abstraction (if possible) and adds it to the repository. The algorithm `createRepairAbstraction` describes this process (Figure 4.3). AbstractJuzi supports the following *kinds* of abstractions:

  - *Self*: set the relevant object field to point to the object itself, e.g., `next` of node `n` is set to `n`;
  - *First*: set the relevant object field to point to the first object of the same type reachable from the given root pointer;
  - *Leaf*: set the relevant object field to point to the furthest object of the same type reachable from the given root pointer;

```
AbstractRepair createRepairAbstraction(Object root,
                  Object source, Field f, Object target) {
    AbstractPosition ap = AbstractPosition.UNDEFINED;
    if (target == null)
        ap = AbstractPosition.NULL;
    else if (source == target)
        ap = AbstractPosition.SELF;
    else if (target == getFirst(root))
        ap = AbstractPosition.FIRST;
    else if (isLeaf(target))
        ap = AbstractPosition.LEAF;
    else if (isNeighbor(source, target))
        ap = AbstractPosition.NEIGHBOR;
    return new AbstractRepair(f, ap);
}

List<RepairAction> applyRepairAbstraction(Object root,
                                    Object source, Field f){
    List<RepairAction> ras = new ArrayList<RepairAction>();
    Iterator<AbstractRepairActions> itr =
                    AbstractRepairActions.getIterator();
    while(itr.hasNext()){
        AbstractRepair ar = itr.getNext();
        if (!ar.field().equals(f)) continue;
        Object target = ar.concretize(input, source);
        ras.add(new RepairAction(source, f, target));
    }
    return ras; // if ras is empty, apply default repair
}
```

Figure 4.3: Repair abstraction algorithms.

- *Neighbor*: set the relevant object field to point to a neighboring object, where two object o and o' are neighbors if a field of o points to o';
- *Null*: set the relevant object field to the value `null`; and

The default kind *Undefined* is used to indicate that the relevant object field needs to be set using the default Juzi algorithm. The predicate `isLeaf` checks that `target` has no non-null fields. The predicate `isNeighbor` checks whether any field of `source` points to `target` and vice versa.

- *Application* of repair abstractions. When Juzi identifies an object field to mutate, AbstractJuzi first checks if an existing repair abstraction can help with the repair. The algorithm `applyRepairAbstraction` describes this process (Figure 4.3). Given the `root` object that represents the corrupt structure, the `source` object (reachable from `root`) that will have a field mutation (for repair), and the field `f` (of `source`) that will be mutated, the algorithm iterates over existing repair abstractions to find the ones that are *applicable*, i.e., apply to the field `f`, concretizes them into the corresponding concrete repair actions with respect to `root`, and returns them in a list. Finding the `target` is straightforward for the cases self, first, and null. To find a leaf, we use a breadth-first search (BFS) from the `root` and terminate the search when the first leaf node is found. To find a neighbor, we again use a BFS until we find the first object that has some field that points to `source`.

4.4 Evaluation

This section presents the experimental evaluation. We compare AbstractJuzi with the original Juzi repair algorithm using the following data structure subjects:

- Singly-linked, circular list. The errors injected in this structure violated the circularity constraint.
- Doubly linked list. The errors injected violate the constraint that `next` is the transpose of `prev`.
- Binary tree. The errors injected in violates the acyclicity constraint.
- Binary tree with parent pointers. The errors injected violate the constraint that `child` (along `left` or `right`) is the transpose of `parent`.

Recall both Juzi and AbstractJuzi check the structure for validity by calling `repOk` after every object field mutation. Therefore, the number of calls made to `repOk` counts the number of candidate structures explored before repair completes. We report this number to compare the efficiency of the two algorithms. To demonstrate the potential repair abstractions hold in optimizing repair in cases when an error recurs, the tabulated number of `repOk` calls for AbstractJuzi excludes the number of calls to create the particular abstraction (when no applicable abstraction was found earlier during repair since it was the first occurrence of the particular error).

We report results for two error scenarios: (1) the corrupt data structure has exactly one erroneous object field (Section 4.4.1); and (2) the corrupt data structure

Table 4.1: The number of `repOk` calls made by AbstractJuzi vs Juzi for fixing the errors.

Structure	*size = 10*		*size = 500*	
	Juzi	*AbstractJuzi*	*Juzi*	*AbstractJuzi*
Circular List	3	2	3	2
Doubly Linked List	6	2	251	2
Binary Tree	2	2	2	2
Binary Tree with Parent Pointers	8	2	263	2

has a small number (> 1) of erroneous object fields and each field is the same (e.g., `parent`) (Section 4.4.2).

All the experiments used a 2.50GHz Core 2 Duo processor with 4.00GB RAM running 64 bit Windows 7 and Sun's Java SDK 1.7.0 JVM.

4.4.1 Single error

We compare Juzi and AbstractJuzi for repairing a single error in small structures (with 10 nodes) and medium structures (with 500 nodes). Table 4.1 summarizes the results of our experiments with the subject structures with one erroneous object field in each subject.

Overall, Juzi's number of `repOk` calls is linear in the size of the structure, which is as expected (since there is exactly one error). However, in two case Juzi makes a constant number of calls to `repOk` to complete the repair. To fix the circularity violation of circular lists, Juzi first tries `null` and then the first node of the list, which works. To fix the acyclicity violation in binary trees, Juzi first tries

`null`, which works.

For all cases, AbstractJuzi performs the repair using a constant number of `repOk` calls – independent of the size of the structure. AbstractJuzi uses the following abstraction kinds for the four subjects:

- Circularity violation in circular lists: $First$;
- Violation of the transpose relation (between `next` and `prev`) in doubly linked lists: $Neighbor$;
- Acyclicity constraint in binary tree: $Null$; and
- Violation of the transpose relation (between `left`/`right` and `parent`) in binary trees with parent pointers: $Neighbor$.

Thus, for the chosen subjects with single error, AbstractJuzi explores a much smaller space of candidate repaired structures than Juzi.

4.4.2 Multiple errors

For two of the subjects, namely doubly linked lists and binary trees with parent pointers, we compare Juzi with AbstractJuzi on repairing multiple errors. We do not consider singly-linked lists here since by construction only one `next` pointer can violate acyclicity (we do not consider nodes unreachable from the given root since we do not have a handle to them). We do not consider binary trees with acyclicity violation since AbstractJuzi reduces to Juzi for repairing cycles – both algorithms use `null` as the first choice.

Table 4.2: The number of `repOk` calls made by AbstractJuzi vs Juzi for multiple errors in Doubly Linked List (DLL) and Binary Tree with Parent Pointers (BTPP) structures of size 500 nodes.

# Erros	*doubly linked lists*		*binary trees with parent pointers*	
	AbstractJuzi	*Juzi*	*AbstractJuzi*	*Juzi*
2	3	319	3	231
3	4	576	4	378
4	5	680	5	567
5	6	769	6	743

Table 4.2 tabulates the results. We fixed the structure size to 500 nodes for both the structures and injected 2,3,4 and 5 random errors – for doubly linked lists, the errors were injected in the `prev` fields; and for binary trees with parent pointers, the errors were injected in the `parent` fields. As before, we use the number of calls made to `repOk` as our metric for comparison.

For all cases, Juzi's number of `repOk` calls is proportional to the product of the number of faults and the size of the structure, which is as expected (since these faults can be fixed independently by Juzi).

For all cases, AbstractJuzi's number of `repOk` calls is proportional to the number of faults – independent of the size of the structure. AbstractJuzi uses the same abstraction kinds for the two subjects as described in Section 4.4.1.

Thus, for the chosen subjects with multiple errors, AbstractJuzi explores a much smaller space of candidate repaired structures than Juzi.

4.5 Summary

This chapter presented repair abstractions to enhance the efficiency and scalability of data structure repair. Our insight is that if an error is due to a fault in software or hardware, it is likely to recur. Therefore, we can abstract the concrete repair actions taken to fix a particular erroneous state and reuse them when a similar error is detected in future. Our embodiment of repair abstractions piggybacks on the existing repair framework Juzi and enables data structure repair using abstractions for Java programs. Experimental results show that repair abstractions can substantially reduce the space of candidate structures to explore in systematic techniques for data structure repair.

Chapter 5

Related Work

This chapter describes the projects most closely related to our work.

5.1 Program Repair

A number of program repair techniques have been introduced in the recent years to repair real-world programs. This sections gives an overview of these techniques. The key difference between our work and these techniques is our use of systematic data structure repair for program repair, which is not the basis of these techniques.

5.1.1 Genetic Programming

Genetic programming (GP) is a variant of genetic algorithms with variable-length string encoding [64]. Arcuri et al. [4] used GP to automate the task of fixing bugs. Their approach is based on co-evolution, in which programs and test cases co-evolve, influencing each other with the aim of fixing the bugs of the programs. Their approach requires formal specification along with the buggy program and tests to work.

Weimer et al. [106] also use genetic programming for program repair; they

generate program variants until one is found that both retains required functionality and also avoids the defects found in the original program. Their technique takes as input a program, a set of successful positive test cases that encode required program behavior, and a failing negative test case that demonstrates a defect. They use GP to maintain a population of variants of that program. Each variant is represented as an abstract syntax tree (AST) paired with a weighted program path. They modify program variants using two genetic algorithm operations, crossover and mutation. Each modification produces a new abstract syntax tree and weighted program path. The fitness of each variant is evaluated by running it on the test cases, and it is assigned a value based on a weighted sum of the positive and negative test cases it passes. The approach stops when it has evolved a program variant that passes all of the test cases. To restrict the search space for the GP authors use two insights: first, they limit the possible variations in the program to changes based on code existing in program elsewhere; and second, the mutation and cross-over work only over faulty part of the code.

5.1.2 Enforcing Contracts

Perkins et al. [85] developed a system for automatically patching errors in deployed software called ClearView. It works on stripped Windows x86 binaries without any need for source code, debugging information, or other external information, and without human intervention. ClearView (1) observes normal executions to learn invariants that characterize the applications normal behavior, (2) uses error detectors to distinguish normal executions from erroneous executions, (3) identifies

violations of learned invariants that occur during erroneous executions, (4) generates candidate repair patches that enforce selected invariants by changing the state or flow of control to make the invariant true, and (5) observes the continued execution of patched applications to select the most successful patch.

Wei et al. [103] developed Auto-FixE. Their tool takes an Eiffel class and, using their earlier work, generates test cases with AutoTest. From the execution runs, they extract object states using boolean queries (similar to repOk). By comparing the states of passing and failing runs, they generate a fault profile which is an indication of what went wrong in terms of abstract object state. From the state transitions in passing runs, they generate a finite-state behavioral model, capturing the normal behavior in terms of control. Both control and state guide the generation of fix candidates. Only those fixes passing the regression test suite remain.

5.1.3 Specification Based Repair

Gopinath's on-going dissertation work explores the problem of repairing programs using more general forms of specifications, such as rich behavioral specifications given in the form of preconditions and postconditions. In a recent paper, we collaborated on a SAT-based approach to generating likely bug fixes [40]. The key insight was to replace a faulty statement that has deterministic behavior with one that has nondeterministic behavior, and to use the specification constraints to prune the ensuing nondeterminism and repair the faulty statement. The SAT-based Alloy tool-set provided the enabling technology for writing specification constraints as well as for solving them. While this approach supports richer forms of specifi-

cations than the repOk methods, it also requires the use of SAT technology, which has not yet been shown to scale to real applications for data structure repair.

5.1.4 Repairing Boolean Programs

Boolean programs are similar to a high-level imperative language programs except that the only variables permissible are boolean variables. Roopsha et al. [90] present an approach based on local Hoare-triple to fix boolean programs. Their algorithm has two main steps. In the first step, they annotate the program by propagating pre and post conditions through the program statements. In the second step, they choose specified order to target statements for repair. For every chosen statement, they synthesize a local repair using the propagated pre and post conditions. Once a synthesis establishes the post condition for the entire function, a repair is extracted and the algorithm halts. If all fix suggestions fail, the algorithm reports that the program is irreparable within the constraints imposed by the repair model. Their approach is promising but cannot be used for data manipulating programs.

5.1.5 Repair as a game

Jobstmann et al. [55] presented a technique that automatically fixes bugs in finite state programs by considering repair as a game. Their approach requires specifications in linear time logic (LTL) against which a program is verified. They limit the faults they can handle to an incorrect left-hand side value of an assignment statement, which they transform to an "unknown" variable. The winning strategy for the system is able to replace the "unknown" variable with one that satisfies the

specifications. The game is played between the LTL model which generates acceptable inputs and the repair tool which provides values for the "unknown" variable. They prove that the problem of program repair as a game is NP-complete but their heuristic behaves well. The approach by Gopinath et al. [40] uses a similar insight for heap manipulating programs with specifications.

5.1.6 Programming by Sketching

Solar-lezama et al. introduced the concept of programming by sketching [97]. Their goal is not program repair but rather code synthesis from a reference program which is very similar to repair. The application of their technique is limited to programs that deal with manipulating streams of data at the bit level. Such manipulations have several properties that make them a hard domain for program developer to produce error free code. The approach requires the programmer to first write a full behavioral specification of a particular bit manipulation task, called as reference program. The reference program is written in a specialized dataflow language, and represents an un-optimized version describing the task at bit level. Having a reference program, the programmer sketches an efficient implementation. The sketching provides only a loosely constrained template of the implementation, with the compiler filling in the remaining details. The details are obtained by ensuring that the resolved sketch is behaviorally equivalent to the reference program. Again the application of this tool is limited to programs that deal with bit manipulation. However, an extension to heap-manipulating programs was recently presented [94].

5.2 Invariant Generation

Invariant generation is a classic research topic with a number of different approaches. Our work is most closely related to dynamic invariant generation [35]. The key difference between our work and previous work is our support for structural invariants of dynamic data structures as well as our idea of using graph spectra as representing structural invariants, which has not been used in previous work.

5.2.1 Daikon

Daikon [35] pioneered dynamic invariant detection. Daikon demonstrated how invariants can be dynamically detected from program traces that capture variable values at program points of interest. The user runs the target program over a test suite to create the traces, and an invariant detector determines which properties and relationships hold over both explicit variables and other expressions. Properties that hold over the traces and also satisfy other tests, such as being statistically justified, not being over unrelated variables, and not being implied by other reported invariants, are reported as likely invariants. Like other dynamic techniques such as testing, the quality of the output depends in part on the comprehensiveness of the test suite. Daikon does not detect high-level data structure properties.

5.2.2 DIDUCE

Like Daikon, DIDUCE (Dynamic Invariant Detection N Checking Engine) [43] tries to extract invariants dynamically from program executions. However, instead of presenting the user with numerous invariants found after a programs ex-

ecution, DIDUCE continually checks the programs behavior against the invariants hypothesized up to that point in the programs run(s) and reports all detected violations. When a dynamic invariant violation is detected, the invariant is relaxed to allow for the new behavior and program execution is resumed. This results in a fully automatic tool that checks a program against a model it creates without requiring any human intervention. However, similar to Daikon, DIDUCE does not detect data structure properties.

5.2.3 Deryaft

We previously implemented Deryaft [70, 71] algorithm for dynamic invariant detection, which is the closest approach to spectral invariant learning. Given a small set of structure representations in heap as examples, Deryaft analyzes them to formulate local and global properties that the structures exhibit. Deryaft focuses on graph properties for effective formulation of structural invariants, including reachability, and views the program heap as an edge-labeled graph. Given a set of concrete structures Deryaft inspects them to formulate a set of hypotheses on the underlying structural as well as data constraints that are likely to hold. Next, it checks which hypotheses actually hold for the structures. Finally, it translates the valid hypotheses into a Java predicate that represents the structural invariants of the given structures. The resulting predicate takes an input structure, traverses it, and returns true if and only if the input satisfies the invariants. We also adapted Deryaft for declarative language Alloy in a tool aDeryaft [61].

5.2.4 Dynamic Shape Analysis

Dynamic shape analysis, as described in ShapeUp [58], dynamically checks recursive data structure invariants by summarizing data structures based on their in-degrees and out-degrees. ShapeUp computes a class-field summary graph (CFSG) which summarizes the dynamic object graph based on class definitions. The CFSG records the number of objects and their recursive degree metrics as in- and out-degree invariants. When a specific number of nodes of a data structure exhibit a particular degree, they start using it as a stable matric to detect the shape. HeapMD [16] is forerunner of ShapeUp and examines simple heap properties in C programs.

5.3 Data Structure Repair

Dynamic repair techniques that aim to counteract the effects of faults at runtime and prolong the uptime of a system have been in existence for a long time. File system utilities such as `fsck` and `chkdsk`, database check-pointing, and rollback techniques are standard repair routines used to monitor and correct the state of system at runtime. More recent techniques have used various forms of specifications for data structure repair. The key difference between our work and previous work is our idea to introduce repair abstractions to memoize and reuse repair actions, which has not been performed in previous work.

5.3.1 Constraint-based Repair

Demsky and Rinard [26] pioneered the idea of constraint-based repair of data structures. Constraints are written in a declarative language similar to Alloy

and repair is performed by translating the constraints to disjunctive normal form and solving them using an ad hoc search.

The Juzi framework [32, 60] that provides the basis of our work on program repair presents an *assertion*-based technique for data structure repair, where assertion violations as indicated by invocations of repOk methods that return false are used to mutate and repair erroneous program states. Symbolic execution of the `repOK` method combined with systematic search of the object space based on last field access aids in efficiently restoring the data structure to a state satisfying the invariants.

Dynamic Symbolic Data Structure Repair [49] (DSDR) extends Juzi's technique by producing a symbolic representation of fields and objects along the path executed in repOK. DSDR builds the path constraint required to take the current path in repOK. When repOK returns false, DSDR uses the conjunction of the negation of the path constraint with the other path conditions and solves them, directly generating a fix irrespective of the exact location of the corrupted object references or fields in the repOK method.

A limitation of constraint-based techniques is that class invariants hold at the entry and exit points of all public methods. These techniques alter the faulty data structure to produce an arbitrary state which satisfies the integrity constraints but can, in the worst case, be very different from the intended output of the method.

5.3.2 Contract-based Repair

Tarmeem [112], Plan B [91], and Cobbler [111] are more recent frameworks that support *contract*-based repair, where corrupt data structures can be repaired with respect to *rich* contracts that include method pre-conditions and post-conditions. These frameworks are based on Alloy toolset [50] and use its SAT-based backend for data structure repair. While contract-based repair allows handling a wider class of errors than constraint-based repair, the cost of repair for using rich contracts at runtime and repairing with respect to them is higher. Nonetheless, our approach to repair abstractions is equally applicable to both constraint-based repair (as we show in this dissertation) and contract-based repair (Section 5.4).

5.4 Repair abstraction using Alloy

The idea of abstracting concrete repair actions is orthogonal to the underlying repair framework used. We can plug in a different backend repair framework (instead of Juzi) and benefit from repair abstractions. Our joint work recently integrated repair abstractions with Cobbler [111] (which uses the Alloy language [50] for writing specifications and its SAT based backend for data structure repair). Experimental results show repair abstractions offer performance benefits in the context of Cobbler similar to those in the context of Juzi.

Chapter 6

Conclusion

We conclude this dissertation by providing a summary of our work on automated debugging, dynamic invariant generation, and efficient run time error recovery.

6.1 Summary

Automated debugging is becoming increasingly important as the size and complexity of software increases. We make a case for using constraint-based data structure repair, a recently developed technique for fault recovery, as a basis for automated debugging. Data structure repair uses given structural integrity constraints for key data structures to monitor their correctness during the execution of a program. If a constraint violation is detected, repair performs mutations on the data structures, i.e., corrupt program state, and transforms it into another state, which satises the desired constraints.

The primary goal of data structure repair is to transform an erroneous state into an acceptable state. The key insight of this thesis is that the mutations performed by repair actions provide a basis of debugging faults in code (assuming the errors are due to bugs). A key challenge to embodying this insight into a mechanical

technique arises due to the difference in the concrete level of the program states and the abstract level of the program code: repair actions apply to concrete data structures that exist at runtime, whereas debugging applies to code. This thesis addresses this challenge by relating static structures (program variables) that hold handles to dynamic structures (heap-allocated data), and performing a guided search.

This thesis focuses on programs that operate on structurally complex data, e.g., heap-allocated data structures that have complex structural integrity constraints, such as acyclicity. Checking such constraints is critical for our techniques and the user must provide them. However, writing the constraints poses a burden on the users. To facilitate the use of constraint-based techniques, we presented a technique to check constraint violations at runtime using graph spectra, which have been studied extensively by mathematicians to capture properties of graphs. We view the heap of an object-oriented program as an edge-labeled graph, which allows us to apply results from graph spectra theory. Experimental results show the effectiveness of using graph spectra as a basis of capturing structural properties of a class of commonly used data structures.

Finally, the thesis presents abstractions for more efficient data structure repair. When an error in the program state is due to a fault in software or hardware, a similar error may occur again. This thesis presents a set of graph-based abstractions that capture how erroneous program executions are repaired using concrete mutations to enable faster repair of similar errors in the future.

Bibliography

[1] Paul Ammann and Jeff Offutt. *Introduction to Software Testing*. Cambridge University Press, 2007.

[2] A. Andoni, D. Daniliuc, S. Khurshid, and D. Marinov. Evaluating the "Small Scope Hypothesis". Technical report, MIT CSAIL, 2003.

[3] John Anvik, Lyndon Hiew, and Gail C. Murphy. Who should fix this bug? In *ICSE*, 2006.

[4] Andrea Arcuri and Xin Yao. A novel co-evolutionary approach to automatic software bug fixing. In *IEEE Congress on Evolutionary Computation*, 2008.

[5] Ittai Balaban, Amir Pnueli, and Lenore D. Zuck. Shape analysis by predicate abstraction. In *VMCAI*, 2005.

[6] Gogul Balakrishnan and Malay Ganai. Ped: Proof-guided error diagnosis by triangulation of program error causes. In *SEFM*, pages 268–278, 2008.

[7] Thomas Ball and James R. Larus. Efficient path profiling. In *Proceedings of the 29th annual ACM/IEEE international symposium on Microarchitecture*, MICRO 29, 1996.

[8] Thomas Ball and Sriram K. Rajamani. The SLAM toolkit. In *CAV*, pages 260–264, 2001.

[9] Kent Beck and Erich Gamma. Test-infected: programmers love writing tests. 2000.

[10] Boris Beizer. *Software Testing Techniques*. International Thomson Computer Press, 1990.

[11] S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The DaCapo benchmarks: Java benchmarking development and analysis. In *OOPSLA*, 2006.

[12] Joshua Bloch. *Effective Java: A Programming Language Guide (Java Series)*. Addison-Wesley Longman, Amsterdam, 2008.

[13] Chandrasekhar Boyapati, Sarfraz Khurshid, and Darko Marinov. Korat: Automated testing based on Java predicates. In *ISSTA*, 2002.

[14] Feng Chen and Grigore Roşu. Java-MOP: A monitoring oriented programming environment for Java. In *TACAS*, 2005.

[15] Trishul Chilimbi, Ben Liblit, Krishna Mehra, Aditya Nori, and Kapil Vaswani. Holmes: Effective statistical debugging via efficient path profiling. In *ICSE*, 2009.

[16] Trishul M. Chilimbi and Vinod Ganapathy. Heapmd: identifying heap-based bugs using anomaly detection. In *ASPLOS*, pages 219–228, 2006.

[17] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. MIT, 1999.

[18] Lori A. Clarke and David S. Rosenblum. A historical perspective on runtime assertion checking in software development. *SIGSOFT Software Engineering Notes*, 31(3), 2006.

[19] James S. Collofello and Larry Cousins. Towards automatic software fault location through decision-to-decision path analysis.

[20] Thomas H. Cormen, Clifford Stein, Ronald L. Rivest, and Charles E. Leiserson. *Introduction to Algorithms*. McGraw-Hill Higher Education, 2nd edition, 2001.

[21] Standard Performance Evaluation Corporation.

[22] Dragos M. Cvetkovic, Michael Doob, and Horst Sachs. *Spectra of Graphs: Theory and Applications*. John Wiley & Sons Inc, 1998.

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

[24] Valentin Dallmeier. *Mining and Checking Object Behavior*. PhD thesis, Department of Computer Science, Saarland University, 2010.

[25] E.R. van Dam and W.H. Haemers. Which graphs are determined by their spectrum? Technical report, Tilburg University, Center for Economic Research, 2002.

[26] Brian Demsky. *Data Structure Repair Using Goal-Directed Reasoning*. PhD thesis, Massachusetts Institute of Technology, 2006.

[27] Brian Demsky and Martin C. Rinard. Automatic detection and repair of errors in data structures. In *OOPSLA*, pages 78–95, 2003.

[28] Greg Dennis, Felix Sheng-Ho Chang, and Daniel Jackson. Modular verification of code with SAT. In *ISSTA*, 2006.

[29] J. Durães and H. Madeira. Emulation of software faults: A field data study and a practical approach. *IEEE Trans. Software Eng.*, 32(11):849–867, 2006.

[30] Bassem Elkarablieh. *Assertion-based Repair of Complex Data Structures*. PhD thesis, University of Texas at Austin, 2009.

[31] Bassem Elkarablieh, Ivan Garcia, Yuk Lai Suen, and Sarfraz Khurshid. Assertion-based repair of complex data structures. In *ASE*, 2007.

[32] Bassem Elkarablieh and Sarfraz Khurshid. Juzi: A tool for repairing complex data structures. In *ICSE*, 2008.

[33] Bassem Elkarablieh, Yehia Zayour, and Sarfraz Khurshid. Efficiently generating structurally complex inputs with thousands of objects. In *ECOOP*, 2007.

[34] Albert Endres. An analysis of errors and their causes in system programs. *SIGPLAN Not.*, 10:327–336, April 1975.

[35] Michael D. Ernst. *Dynamically Discovering Likely Program Invariants*. PhD thesis, University of Washington, 2000.

[36] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended static checking for Java. In *PLDI*, 2002.

[37] Erich Gamma and Kent Beck. JUnit: A cook's tour. `http://www.junit.org`.

[38] Rakesh Ghiya and Laurie J. Hendren. Is it a tree, a dag, or a cyclic graph? a shape analysis for heap-directed pointers in C. In *POPL*, 1996.

[39] Khalid Ghori. Constraint-based program repair. MS Thesis, University of Texas at Austin, 2006.

[40] Divya Gopinath, Muhammad Zubair Malik, and Sarfraz Khurshid. Specification-based program repair using sat. In *TACAS*, pages 173–188, 2011.

[41] Kristen Grauman and Trevor Darrell. Unsupervised learning of categories from sets of partially matching image features. In *CVPR*, 2006.

[42] Neelam Gupta, Haifeng He, Xiangyu Zhang, and Rajiv Gupta. Locating faulty code using failure-inducing chops. In *ASE*, 2005.

[43] Sudheendra Hangal and Monica S. Lam. Tracking down software bugs using automatic anomaly detection. In *ICSE*, pages 291–301, 2002.

[44] G. Haugk, F. Lax, R. Royer, and J. Williams. The 5ESS(TM) switching system: Maintenance capabilities. *AT&T Technical Journal,*, 64.

[45] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, 1969.

[46] Pieter Hooimeijer and Westley Weimer. Modeling bug report quality. In *ASE*, 2007.

[47] David Hovemeyer and William Pugh. Finding bugs is easy. In *OOPSLA*, 2004.

[48] David Hovemeyer and William Pugh. Finding more null pointer bugs, but not too many. In *PASTE*, 2007.

[49] Ishtiaque Hussain and Christoph Csallner. Dynamic symbolic data structure repair. In *ICSE*, 2010.

[50] Daniel Jackson. *Software Abstractions: Logic, Language and Analysis*. MIT-P, 2006.

[51] Daniel Jackson and Alan Fekete. Lightweight analysis of object interactions. In *TACS*, 2001.

[52] Daniel Jackson and Mandana Vaziri. Finding bugs with a constraint solver. In *ISSTA*, 2000.

[53] Dennis Jeffrey, Min Feng, Neelam Gupta, and Rajiv Gupta. BugFix: A learning-based tool to assist developers in fixing bugs. In *ICPC*, 2009.

[54] Dennis Jeffrey, Neelam Gupta, and Rajiv Gupta. Fault localization using value replacement. In *ISSTA*, 2008.

[55] Barbara Jobstmann, Andreas Griesmayer, and Roderick Bloem. Program repair as a game. In *CAV*, 2005.

[56] James A. Jones and Mary Jean Harrold. Empirical evaluation of the tarantula automatic fault-localization technique. In *ASE '05*, pages 273–282, 2005.

[57] James Arthur Jones. *Semi-Automatic Fault Localization*. PhD thesis, Georgia Institute of Technology, 2008.

[58] Maria Jump and Kathryn S. McKinley. Dynamic shape analysis via degree metrics. In *ISMM*, pages 119–128, 2009.

[59] Sarfraz Khurshid. *Generating Structurally Complex Tests from Declarative Constraints*. PhD thesis, Massachusetts Institute of Technology, 2003.

[60] Sarfraz Khurshid, Iván García, and Yuk Lai Suen. Repairing structurally complex data. In *SPIN*, 2005.

[61] Sarfraz Khurshid, Muhammad Zubair Malik, and Engin Uzuncaova. An automated approach for writing Alloy specifications using instances. In *ISoLA*, 2006.

[62] Miryung Kim and David Notkin. Discovering and representing systematic code changes. In *ICSE*, pages 309–319, 2009.

[63] Konstantin Knizhnik and Cyrille Artho. Jlint. `http://jlint.sourceforge.net/`, 2012.

[64] John Koza. *Genetic programming: on the programming of computers by means of natural selection*. MIT Press, 1992.

[65] Ben Liblit, Alex Aiken, Alice X. Zheng, and Michael I. Jordan. Bug isolation via remote program sampling. In *PLDI*, 2003.

[66] Barbara Liskov and John Guttag. *Program Development in Java: Abstraction, Specification, and Object-Oriented Design*.

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

[68] Muhammad Zubair Malik. Dynamic shape analysis of program heap using graph spectra. In *ICSE*, pages 952–955, 2011.

[69] Muhammad Zubair Malik and Sarfraz Khurshid. Dynamic shape analysis using spectral graph properties. In *ICST*, pages 211–220, 2012.

[70] Muhammad Zubair Malik, Aman Pervaiz, and Sarfraz Khurshid. Generating representation invariants of structurally complex data. In *TACAS*, 2007.

[71] Muhammad Zubair Malik, Aman Pervaiz, Engin Uzuncaova, and Sarfraz Khurshid. Deryaft: a tool for generating representation invariants of structurally complex data. In *ICSE*, 2008.

[72] Muhammad Zubair Malik, Junaid Haroon Siddiqui, and Sarfraz Khurshid. Constraint-based program debugging using data structure repair. In *ICST*, pages 190–199, 2011.

[73] Darko Marinov. *Automatic Testing of Software with Structurally Complex Inputs*. PhD thesis, Massachusetts Institute of Technology, 2004.

[74] Darko Marinov and Sarfraz Khurshid. TestEra: A novel framework for automated testing of Java programs. 2001.

[75] David Meyer, Friedrich Leisch, and Kurt Hornik. The `libsvm` support vector machine under test. Presented at "Statistical Computing", Reisensburg, Germany, June 30 2003.

[76] T. M. Mitchell. *Machine learning*. McGraw Hill, 1997.

[77] Anders Møller and Michael I. Schwartzbach. The pointer assertion logic engine. In *PLDI*, 2001.

[78] Samiha Mourad and Dorothy Andrews. On the reliability of the ibm mvs/xa operating system. *IEEE Trans. Softw. Eng.*, 13(10):1135–1139, October 1987.

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

[80] A. Ng, M. Jordan, and Y. Weiss. On spectral clustering: Analysis and an algorithm. *Advances in Neural Information Processing Systems 14: Proceedings of the 2001.*, 2001.

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

[82] MathWorks Inc.and NIST. Jama–a Java matrix package, 2005.

[83] NSF. Basic linear algebra subprograms. `http://netlib.org/blas/`.

[84] Terence Parr and Others. Another tool for language recognition. `http://www.antlr.org/`.

[85] Jeff H. Perkins, Sunghun Kim, Samuel Larsen, Saman P. Amarasinghe, Jonathan Bachrach, Michael Carbin, Carlos Pacheco, Frank Sherwood, Stelios Sidiroglou, Greg Sullivan, Weng-Fai Wong, Yoav Zibin, Michael D. Ernst, and Martin C. Rinard. Automatically patching errors in deployed software. In *SOSP*, 2009.

[86] Manos Renieris and Steven P. Reiss. Fault localization with nearest neighbor queries. In *ASE*, 2003.

[87] Jeffery D. Rutter. A serial implementation of cuppen's divide and conquer algorithm. Technical report, Berkeley, CA, USA, 1991.

[88] Shmuel Sagiv, Thomas W. Reps, and Reinhard Wilhelm. Solving shape-analysis problems in languages with destructive updating. *TOPLAS*, 20(1), 1998.

[89] Shmuel Sagiv, Thomas W. Reps, and Reinhard Wilhelm. Parametric shape analysis via 3-valued logic. *ACM Trans. Program. Lang. Syst.*, 24(3), 2002.

[90] Roopsha Samanta, Jyotirmoy V. Deshmukh, and E. Allen Emerson. Automatic generation of local repairs for boolean programs. In *FMCAD*, 2008.

[91] H. Samimi, E.D. Aung, and T. Millstein. Falling Back on Executable Specifications. In *ECOOP*, 2010.

[92] Koushik Sen, Darko Marinov, and Gul Agha. CUTE: a concolic unit testing engine for C. In *FSE*, 2005.

[93] Jianbo Shi and Jitendra Malik. Normalized cuts and image segmentation. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 22:888–905, 2000.

[94] Rishabh Singh and Armando Solar-Lezama. Synthesizing data structure manipulations from storyboards. In *SIGSOFT FSE*, 2011.

[95] S. Sinha et al. Fault localization and repair for Java runtime exceptions. In *ISSTA*, pages 153–164, 2009.

[96] Armando Solar-Lezama. The sketching approach to program synthesis. In *APLAS*, 2009.

[97] Armando Solar-Lezama, Rodric M. Rabbah, Rastislav Bodík, and Kemal Ebcioglu. Programming by sketching for bit-streaming programs. In *PLDI*, 2005.

[98] SPEC. Standard Performance Evaluation Corporation. `http://www.spec.org`.

[99] Emina Torlak and Daniel Jackson. Kodkod: A relational model finder. In *TACAS*, 2007.

[100] Pieter Vanraemdonck, NoClassAttribute, Raik Schroeder, Steve Hawkins, et al. Pmd. `http://pmd.sourceforge.net/pmd-5.0.0/`, 2012.

[101] Iris Vessey and Ron Weber. Some factors affecting program repair maintenance: an empirical study. *Commun. ACM*, 26:128–134, February 1983.

[102] Srinivas Visvanathan and Neelam Gupta. Generating test data for functions with pointer inputs. In *ASE*, 2002.

[103] Yi Wei, Yu Pei, Carlo A. Furia, Lucas S. Silva, Stefan Buchholz, Bertrand Meyer, and Andreas Zeller. Automated fixing of programs with contracts. In *ISSTA*, 2010.

[104] Westley Weimer. Patches as better bug reports. In *GPCE*, 2006.

[105] Westley Weimer, Stephanie Forrest, Claire Le Goues, and ThanhVu Nguyen. Automatic program repair with evolutionary computation. *Communications of the ACM*, 53(5):109–116, May 2010.

[106] Westley Weimer, ThanhVu Nguyen, Claire Le Goues, and Stephanie Forrest. Automatically finding patches using genetic programming. In *ICSE*, 2009.

[107] Mark Weiser. Programmers use slices when debugging. *Commun. ACM*, 1982.

[108] R. Clinton Whaley and Jack Dongarra. Automatically tuned linear algebra software. In *PPSC*, 1999.

[109] Thomas Wies, Viktor Kuncak, Patrick Lam, Andreas Podelski, and Martin C. Rinard. Field constraint analysis. In *VMCAI*, 2006.

[110] Wuu Yang. Identifying syntactic differences between two programs. *Softw. Pract. Exper.*, 21(7):739–755, 1991.

[111] Razieh Nokhbeh Zaeem, Divya Gopinath, Sarfraz Khurshid, and Kathryn S. McKinley. History-aware data structure repair using sat. In *TACAS*, pages 2–17, 2012.

[112] Razieh Nokhbeh Zaeem and Sarfraz Khurshid. Contract-based data structure repair using Alloy. In *ECOOP*, 2010.

[113] Razieh Nokhbeh Zaeem, Muhammad Zubair Malik, and Sarfraz Khurshid. Repair abstractions for more efficient data structure repair. In *RV*, pages 235–250, 2013.

[114] Andreas Zeller. Isolating cause-effect chains from computer programs. In *FSE*, 2002.

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

[116] Lihi Zelnik-manor and Pietro Perona. Self-tuning spectral clustering. In *Advances in Neural Information Processing Systems 17*. MIT Press, 2004.

[117] Xiangyu Zhang, Neelam Gupta, and Rajiv Gupta. Locating faults through automated predicate switching. In *ICSE*, pages 272–281, 2006.

[118] P. Zhu and R. C. Wilson. A study of graph spectra for comparing graphs. In *BMVC*, 2005.

Vita

Zubair was born in Lahore, Pakistan in 1978. He received his Bachelor of Engineering in December 1999 from Ghulam Ishaq Khan Institute of Engineering Sciences and Technology. From 1999 to 2001 Zubair worked as a Software Engineer at IBM and Xavor Pakistan. In 2003 he completed his Masters of Science in Computer Science from National University of Computer and Emerging Sciences, during which he was funded by Punjab Information Technology Board's outstanding talent scholarship. He went back to Xavor and worked for two years as Senior Software Engineer. He completed his Masters of Engineering in Computer Engineering from University of Texas at Austin in 2007 during which he was funded by Fulbright Scholarship and Prestigious David Bruton Jr. Fellowship. During the summer of 2007 he interned at ObjectVideo Inc. His research interests focus on discovering new techniques to improve program reliability, program analysis and repair. Zubair enjoys playing tennis, squash, chess and has been a champion starcraft player.

Email address: zubair.malik@utexas.edu

This dissertation was typeset with LaTeX[†] by the author.

[†]LaTeX is a document preparation system developed by Leslie Lamport as a special version of Donald Knuth's TeX Program.