

Copyright
by
Razieh Nokhbeh Zaeem
2014

The Dissertation Committee for Razieh Nokhbeh Zaeem certifies that this is the approved version of the following dissertation:

**Contract-Driven Data Structure Repair:
A Novel Approach for Error Recovery**

Committee:

Sarfraz Khurshid, Supervisor

Adnan Aziz

John Hasenbein

Kathryn S. McKinley

Dewayne E. Perry

**Contract-Driven Data Structure Repair:
A Novel Approach for Error Recovery**

by

Razieh Nokhbeh Zaeem, B.E., 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

May 2014

To my mother Fatemeh.

Acknowledgments

First and foremost, I thank God, the most merciful, for everything.

I wish to thank the multitudes of people who helped me throughout my journey as a PhD student. I would like to sincerely thank my supervisor, Dr. Sarfraz Khurshid, for his priceless help and support. Sarfraz always understood my situation (an international student with many travel limitations and a baby at home, to mention only a few), traveled on my behalf, campaigned for my job, and gave me the freedom much required to develop my own research agenda. I would like to extend my appreciation to my committee members, Dr. Adnan Aziz, Dr. John Hasenbein, and Dr. Dewayne E. Perry, for their invaluable guidance. In particular, I am grateful to Dr. Kathryn S. McKinley for helpful ideas, detailed discussion and feedback, as well as her overall support.

I am thankful to my peers in Software Verification, Validation and Testing group: Guowei Yang, Muhammad Zubair Malik, Lingming Zhang, Allison Sullivan, Shadi Abdul Khalek, Junaid Haroon Siddiqui, Chang Hwan Peter Kim, Vidya Narayanan, Mehmet Erol Yesin, and particularly my friend and co-author Divya Gopinath for her many helpful comments. I also thank Ms. Melanie Gulick, the graduate coordinator of the Electrical and Computer Engineering department.

I thank my dear friends for always being there for me: Maryam Mor-tazavi, Zahra Mohammadi, Mansoureh Peydayesh, Pegah Zabetirad, Zahra Dehghani, Kowsar Yousefi, Hosniyeh Nekoofar, Leila Moravvej, and Marzieh Sadat Tabatabayi. I especially thank Maryam Jelvehi Moghaddam for helping on my defense day.

Finally, I am deeply grateful to the love of my life and my husband, Mahdi Kefayati, not only for his affection and emotional support, but also for his inspiration and contribution to my research as a fellow PhD student. He is the one always thanked in my papers for helpful discussion and proofreading, and in my heart, for his unbelievably enormous love. Time would fail me to tell how I am indebted to my parents, Fatemeh and Davoud. May God endow them with best of rewards. I thank my sister Marzieh and my brother Mohammad for filling my empty space in the family, and wish to see them soon after seven long years. I also extend my love to my dear aunt Tooba. Lastly, I would like to make special mention of my baby, Fatima Grace, and thank her for accommodating an always busy mom, and for playing quietly as I finish this dissertation.

The idea of repair abstraction was originally proposed by Muhammad Zubair Malik in his doctoral dissertation proposal.

This work was funded in part by the NSF under Grant Nos. CCF-0845628, CCF-1319688, CNS-0958231, IIS-0438967, CCF-1018271, CCF-0811524, and SHF-0910818, AFOSR grant FA9550-09-1-0351, and Fujitsu Labs of America SRA No. UTA12-001194.

Contract-Driven Data Structure Repair: A Novel Approach for Error Recovery

Publication No. _____

Razieh Nokhbeh Zaeem, Ph.D.
The University of Texas at Austin, 2014

Supervisor: Sarfraz Khurshid

Software systems are now pervasive throughout our world. The reliability of these systems is an urgent necessity. A large degree of research effort on increasing software reliability is dedicated to requirements, architecture, design, implementation and testing—activities that are performed *before* system deployment. While such approaches have become substantially more advanced, software remains buggy and failures remain expensive.

We take a radically different approach to reliability from previous approaches, namely *contract-driven data structure repair* for runtime error recovery, where erroneous executions of deployed software are corrected on-the-fly using rich behavioral contracts. Our key insight is to transform the software contract—which gives a high level description of the expected behavior—to an efficient implementation which repairs the erroneous data structures in the

program state upon an error. To improve efficiency, scalability, and effectiveness of repair, in addition to rich behavioral contracts, we leverage the current erroneous state, dynamic behavior of the program, as well as repair history and abstraction.

A core technical problem our approach to repair addresses is construction of structurally complex data that satisfy desired properties. We present a novel structure generation technique based on dynamic programming—a classic optimization approach—to utilize the recursive nature of the structures. We use our technique for constraint-based *testing*. It provides better scalability than previous work. We applied it to test widely-used web browsers and found some known and unknown bugs. Our use of dynamic programming in structure generation opens a new future direction to tackle the scalability problem of data structure repair.

This research advances our ability to develop correct programs. For programs that already have contracts, error recovery using our approach can come at a low cost. The same contracts can be used for systematically testing code before deployment using existing as well as our new techniques. Thus, we enable a novel unification of software verification and error recovery.

Table of Contents

Acknowledgments	v
Abstract	vii
List of Tables	xii
List of Figures	xiv
List of Listings	xvii
Chapter 1. Introduction	1
1.1 Contract-Driven Data Structure Repair	2
1.2 Background: Basic Idea of Contract-Based Data Structure Repair Using Alloy	5
1.3 Optimizations for Contract-Driven Data Structure Repair	7
1.3.1 Program Execution History through Write and Read Barriers for Data Structure Repair	7
1.3.2 Repair History through Unsatisfiable Cores for Data Structure Repair	7
1.3.3 Abstract Repair History	10
1.4 Structure Generation Problem in Testing and Repair	11
1.4.1 Test Input Generation Using Dynamic Programming	11
1.4.2 Ideas for Repair Using Dynamic Programming	12
1.5 Usability	13
1.6 Contributions	14
Chapter 2. Related Work	18
2.1 Data Structure Repair	18
2.2 Test Input Generation	22

Chapter 3. Background: Contract-Based Data Structure Repair Using Alloy	25
3.1 Example	25
3.2 Background on Alloy	28
3.3 Our Previous Work: Contract-Based Data Structure Repair Using Alloy	32
Chapter 4. History-Aware Data Structure Repair Using SAT	37
4.1 Using Barriers for Data Structure Repair	39
4.2 Using UNSAT Cores for Data Structure Repair	40
4.3 Illustration of History-Aware Data Structure Repair	41
4.4 Cobbler: Implementation of History-Aware Repair	43
4.5 Cobbler Evaluation	44
4.5.1 Evaluation Metrics	45
4.5.2 Subject Programs	46
4.5.3 Errors	48
4.5.4 Subject Tools	50
4.5.5 Results	51
4.5.6 ANTLR <code>BaseTree addChild</code>	56
4.6 Summary	57
Chapter 5. Repair Abstractions	59
5.1 Repair Abstractions with Alloy Back-End	60
5.2 DREAM Framework	63
5.2.1 Abstraction and Concretization	67
5.3 DREAM with Alloy Back-End	70
5.3.1 Arreh	70
5.4 Evaluation of DREAM with Alloy Back-End	71
5.5 Summary	80

Chapter 6. Data Structure Generation Using Dynamic Programming	81
6.1 Example	85
6.2 Test Input Generation Framework	87
6.2.1 Recursive repOK Methods	88
6.2.2 Algorithms	89
6.2.2.1 DP	89
6.2.2.2 Random Generation	96
6.2.2.3 LazyDP	97
6.2.2.4 SymboLazyDP	98
6.2.3 Theorem on Test Generation Algorithm	100
6.3 Evaluation: Test Input Generation Using Dynamic Programming	106
6.3.1 Experimental Settings	108
6.3.2 Microbenchmarks	108
6.3.3 Random Test Generation	116
6.3.4 Google Chrome and Apple Safari	118
6.3.4.1 Modeling HTML and CSS Test Inputs	119
6.3.4.2 Experimental Results	120
6.3.4.3 Differential Testing	121
6.3.4.4 Bugs Found	122
6.3.4.5 Applying Symbolic Execution and Korat	126
6.3.5 Threats to Validity	127
6.4 Applicability	128
6.5 Ideas on Leveraging Dynamic Programming for Repair	129
6.5.1 Localizing Errors with Recursive Contracts	130
6.5.1.1 Detecting Cycles	132
6.5.1.2 Supporting Post-Conditions	133
6.5.2 Repairing Data Structures with Pre-Generated Patches	133
6.6 Summary	134
Chapter 7. Conclusions	136
7.1 Final Thoughts	138

Bibliography	140
Vita	158

List of Tables

4.1	The injected faults and ANTLR addChild() fault. The last column shows if the field(s) that should be corrected appear in the write barrier log (WB), read barrier log (RB), or all fields excluding the write and read barrier logs (ALL fields).	49
5.1	Description of the Singly Linked List errors used for experimental evaluation of DREAM.	73
5.2	Abstract repair actions suggested by DREAM for Singly Linked List.	74
5.3	Time taken to repair erroneous Singly Linked Lists (ms).	75
5.4	Description of the Red Black Tree errors used for experimental evaluation.	77
5.5	Abstract repair actions suggested by DREAM for Red Black Tree.	78
5.6	Time taken to repair erroneous Red Black Trees (ms). Timeout represents a timeout of 500,000 ms.	79

6.1	Exhaustive test generation for the biggest sizes considered. Benchmarks include sorted singly-linked lists (LL), binary trees (BT), red-black trees (RBT), Fibonacci heaps (FH), binary heaps (BH), and hash tables (HT). TO represents a timeout of 1000s. Best performance highlighted.	109
6.2	Random generation of ten tests with $90 \leq \text{size} \leq 100$. Benchmarks include sorted singly-linked lists (LL), binary trees (BT), red-black trees (RBT), Fibonacci heaps (FH), binary heaps (BH), and hash tables (HT). TO represents a timeout of 1000s. Best performance highlighted.	117
6.3	Chrome and Safari test input generation results.	121

List of Figures

1.1	Traditional approach to errors (left) versus repair approach (right).	2
1.2	An example of combining binary trees to build a bigger binary tree.	13
3.1	Bug <code>cycle</code> manifested as a faulty output and the repair result.	30
3.2	Relational representation of data structures in Alloy models. .	31
4.1	<code>cycle</code> manifested as a faulty output and its history-aware repair result.	42
4.2	The relationship between Cobbler, the Java Virtual Machine, and the program.	44
4.3	Performance and accuracy: repairing singly linked lists with Cobbler (C), Tarmeem (T), an enhanced version of Juzi (J), and PBnJ (P).	53
4.4	Cobbler performance and accuracy: repairing Kodkod red-black trees.	55
4.5	Cobbler performance and accuracy: repairing ANTLR trees. .	56

5.1	Concrete and abstract repair actions to repair the result of bug <code>cycle</code> on a tree of three nodes.	61
5.2	Abstract and concrete repair actions to repair the result of bug <code>cycle</code> on a tree of five nodes.	62
5.3	The relationship between DREAM, the underlying repair framework, the Java Virtual Machine, and the program.	64
5.4	A snapshot of Arreh.	72
6.1	All binary trees up to size 2.	86
6.2	A tree representation of an HTML input.	87
6.3	Finding binary trees up to size three (first iteration).	93
6.4	Finding binary trees up to size three (second iteration).	94
6.5	Finding binary trees up to size three (third iteration).	95
6.6	Performance comparison on linked lists.	111
6.7	Performance comparison on binary trees.	111
6.8	Performance comparison on red-black trees.	113
6.9	Memory usage on red-black trees.	113
6.10	Performance comparison on Fibonacci heaps.	114
6.11	Performance comparison on binary heaps.	115
6.12	Performance comparison on hash tables.	116

6.13	A back-face visibility bug found in Chrome (left). Safari (right) shows the expected output.	123
6.14	A webkit-perspective bug found in Chrome (up). Safari (down) shows the expected output.	124
6.15	A rotation direction bug found in Chrome (left). Safari (right) shows the expected output.	126
6.16	Patching structures to repair the faulty output of bug <code>cycle</code> . .	131

List of Listings

3.1	A binary search tree implementation in Java [2].	26
3.2	A binary search tree node implementation in Java [2].	27
3.3	Binary search tree contract specification in Alloy.	29
4.1	History-aware contract-based repair using read and write logs and unsatisfiable cores.	39
5.1	DREAM main algorithm.	65
6.1	A recursive binary tree in Java.	86
6.2	HTML repOK method.	87
6.3	Test generation algorithm in Java.	91
6.4	Instrumenting BinaryTree for symbolic execution.	100
6.5	An automatically generated HTML test input.	119
6.6	Abstraction of a CSS rule.	119
6.7	An automatically generated CSS test input (file.css).	120
6.8	Simplified HTML/CSS test input that reveals the webkit-perspective bug in Chrome.	124
6.9	Simplified HTML/CSS test input that reveals the rotation di- rection bug in Chrome.	125

Chapter 1

Introduction

Software systems are pervasive and integrated into almost every aspect of life. Software reliability is essential for life-critical, science, and business applications. For entertainment, software reliability drives system usability. There is a considerable body of research around producing reliable software in various phases of the software development lifecycle before deployment, from extracting requirements to design, implementation, and testing. However, improving the reliability of an already deployed (possibly faulty) system using error recovery is a less explored area.

In practice, systems are deployed with unknown and known unfixed bugs. When bugs cause failures, the usual approach is to restart the program because fixing bugs and redeploying software may take months. Although the latter approach may resolve the fundamental source of the problem, system downtime is undesirable and not always feasible. Many applications, such as operating systems, may prefer to trade slight deviations in intended functionality for system uptime. Better still, if developers annotate programs with specifications, then the runtime may restore the system state to provide its intended functionality. Continuing program execution by fixing the effect of

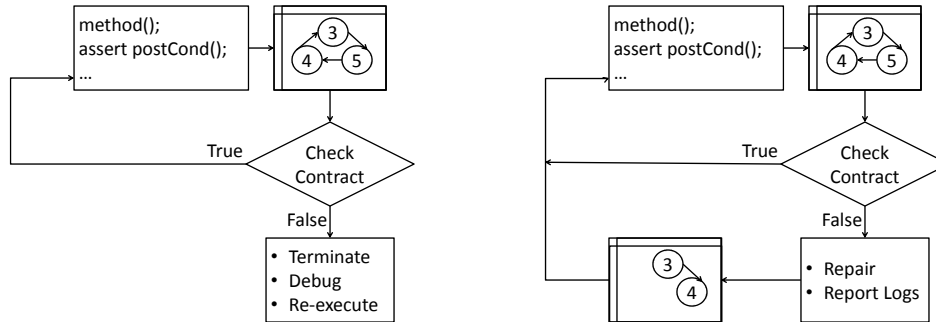


Figure 1.1: Traditional approach to errors (left) versus repair approach (right).

bugs on the program state on-the-fly is called *data structure repair*. Figure 1.1 compares the traditional approach to errors with the repair approach.

1.1 Contract-Driven Data Structure Repair

Existing techniques for repair have not so far lived up to their full potential, because they are either not general purpose or too inefficient. While repair mechanisms are not in standard use today, repair has featured in various systems over the last couple of decades [4, 6, 75, 45]. However, a limitation of the traditional approaches to repair is to use dedicated repair routines, which must be implemented for each system they are intended to work for. As a result, these routines are mostly ad-hoc and ill-understood.

Recent work introduced *constraint-based repair* where data structure constraints written using first-order logic [30, 29] or as Java assertions [55, 31] are used as a basis for repairing erroneous states. While these approaches do

not necessitate writing a dedicated repair routine, they also have a basic limitation: data structure constraint specifications are too weak for error recovery in general. To illustrate, in object-oriented programs, the *class invariant* [64] (which defines the data structure constraints for the valid objects of the class) applies to the entry and exit points of all public methods—even though the precise behaviors of the methods may be very different. For example, consider an erroneous implementation of a method to insert an element into a *binary tree*—an acyclic data structure. Previous approaches [30, 55] to constraint-based repair would accept an empty tree as a valid structure since it satisfies the acyclicity constraint. However, an empty tree is unlikely to be a valid output of insert.

Method contracts [71] naturally suit the repair process since they support class invariants as well as pre- and post-conditions. However, while they have long been used to improve software reliability in different phases of software development lifecycle, their common usage follows the same standard *halt-on-error* approach upon detecting an erroneous program state in a deployed software.

Our previous work [105, 104, 102] introduced *contract-based* error recovery, which addresses the fundamental limitation of constraint-based repair. Our insight is to transmute the inherently non-deterministic specification provided as program contract into an efficient implementation. While contract-based repair introduces a new exciting direction, developing practical solutions poses a suite of challenges: (1) **Efficiency**: Contracts used to describe the in-

tended functionality of a system are inherently inefficient if directly used as an implementation. Even though the repair framework comes into play occasionally (only when an error occurs), its use of contracts should still be efficient enough to satisfy the system requirements. Moreover, the repair framework should be lightweight when the system is functioning as intended and no problem occurs. (2) **Scalability:** Real world programs maintain data structures with thousands of objects as a part of their state, for which repair must remain capable of enforcing contracts, locating errors, and fixing them efficiently. (3) **Effectiveness:** In addition to providing a data structure that adheres to the contracts, we would like to minimize the amount of perturbation introduced by the repair process to keep the final result closer to what would be generated by the program in the absence of any bugs. (4) **Usability:** For a repair system to be usable, not only should it be fast, scalable, and effective, but it should have a user-friendly interface, give appropriate feedback, and report logs about the repaired error so that the user can fix it permanently.

In this dissertation, we propose optimizations to address the above challenges. In addition to rich behavioral contracts, we use program execution history through barriers, SAT solving history through unsatisfiable cores that SAT solvers provide, and abstracted history of previous successful repairs. Experimental results show that our new repair technique scales better than previous work.

Furthermore, we observe that structure construction, a central problem in repair, also arises in a slightly different form in systematic constraint-

based testing. We leverage dynamic programming—a well-known optimization method—to define a novel structure generation technique, which provides better scalability than previous work. We applied our technique to test two widely-used web browsers, Apple Safari and Google Chrome, and found some known and unknown bugs in Chrome. Our use of dynamic programming in structure generation opens a new future direction to tackle the scalability problem of data structure repair.

1.2 Background: Basic Idea of Contract-Based Data Structure Repair Using Alloy

The basic idea of contract-driven repair was introduced in my Master’s thesis [105, 104, 102]. This work presented a contract-based approach for data structure repair, which repairs erroneous executions in deployed software by repairing erroneous states. The key novelty is the support for rich behavioral contract specifications, such as those that relate pre-states with post-states of methods. We leveraged the Alloy tool-set, specifically the Alloy language [51] and SAT solver-based Alloy Analyzer for systematically repairing erroneous states.

This prior work mathematically defines the repair problem.

Definition: Let ϕ be a method post-condition that relates pre- and post-states such that $\phi(r, t)$ if and only if pre-state r and post-state t satisfy the post-condition. Given a valid pre-state u , and an invalid post-state s (i.e., $!\phi(u, s)$), mutate s into state s' such that $\phi(u, s')$.

Four different algorithms are presented and implemented in our previous data structure repair framework—*Tarmeem*: (1) The basic method which is oblivious to the current faulty post-state and directly applies the contract on the pre-state to obtain a solution; (2) Iterative relaxation which enhances the performance and scalability by focusing on specific parts of the faulty post-state and leverages Alloy Analyzer to find proper replacements for corrupted values one at a time; (3) Error localization, which in addition to the faulty post-state, makes use of the post-condition to accurately locate the error and repair it; and (4) Guided error localization which builds on top of error localization and takes user guides to more precisely specify the erroneous parts of the data structure.

To improve the effectiveness of repair, this prior work proposes the graph *edit distance* metric to measure the perturbation introduced by the repair process. To address the usability challenge, this work uses the Alloy language to describe invariants, pre- and post-conditions. In addition, it provides annotations for the user to give hints to the repair system via guides.

Experiments using complex specifications show the approach holds much promise in increasing software reliability [105, 104, 102]. However, our use of SAT back-end of Alloy Analyzer limited the efficiency and scalability of *Tarmeem*. For instance, it would take *Tarmeem* up to 15 seconds to repair a linked list data structure of 20 nodes.

1.3 Optimizations for Contract-Driven Data Structure Repair

We present a suite of three optimizations to for more efficient, scalable, and effective data structure repair using contracts.

1.3.1 Program Execution History through Write and Read Barriers for Data Structure Repair

History-aware repair utilizes the history of a faulty program execution by focusing repair on fields recently modified or read by the program, thereby reducing the search space for SAT [103]. We record program writes and reads to the key data structure with *barriers*. A barrier is a code sequence that performs an action just prior to a write or read. Barriers are widely available in commercial and research implementations of managed languages, e.g., the HotSpot and Jikes RVM Java Virtual Machines, and the .NET C# system. Our approach inserts barrier instrumentation on writes and reads or piggy-backs on existing barriers.

1.3.2 Repair History through Unsatisfiable Cores for Data Structure Repair

History-aware repair further utilizes the unsatisfiable core generated by a SAT run, which captures the history and core elements of the solver's reasoning and not only facilitates locating faults but can even be leveraged directly to optimize a successive SAT run [103]. While using the history of program execution through write and read barriers aids in improving repair

performance and scalability, its heuristic nature implies that there exist cases in which we have to perform a broader search and consider fields not included in the execution trace. In such cases, we take advantage of *UNSAT* cores, which are minimal unsatisfiable sub-formulas provided by failed SAT invocations. When SAT invocations fail, we utilize their UNSAT cores to identify faulty fields. A final SAT invocation with the list of faulty fields extracted from the UNSAT core results in a repaired data structure.

We implement history-aware contract-driven repair for Java programs in a tool called *Cobbler* [103]. As an enabling technology, Cobbler uses the Alloy tool-set, its Kodkod back-end, and a SAT solver. Cobbler inserts write and read instrumentation for the specified data structures to log dynamic program behavior. When Cobbler detects a contract violation, it starts by restricting the SAT solver to correcting written fields and values, followed by read fields during the execution, and if the SAT solver has still not found a correction, it utilizes the UNSAT core provided by the previous SAT invocations to identify and mutate faulty fields of the data structure.

History-aware repair addresses repair challenges in several ways. First, it improves efficiency in two orthogonal directions: (1) It promises a minimal overhead burdened on an error-free execution by presenting a non-conventional application of write barriers, which are a routinely used mechanism in garbage collection and (2) it provides speedups over basic contract-based repair by reducing the size of the search space when performing repair [103]. We explore the efficiency and accuracy of Cobbler on microbenchmarks and two open

source programs: Kodkod solver[93] and ANTLR[1, 12]. We compare our history-aware contract-based repair tool, Cobbler, to contract-based repair alone using PBnJ [82] and our previous tool Tarmeem [104], two repair tools which leverage user guides and heuristics along with a SAT solver. Cobbler is substantially more efficient and scalable than PBnJ and Tarmeem. We also compare Cobbler with Juzi, which uses data structure specifications for repair, but does not use method post-conditions [33, 32]. Juzi’s dedicated constraint solver is more efficient than Cobbler, but Juzi’s repair is applicable to far fewer cases and Cobbler is much more accurate.

Second, we make repair more scalable by making it more efficient. E.g., Cobbler repairs a linked list of 200 nodes in 15 seconds which is ten times more than what Tarmeem could handle in the same period of time.

Third, we keep the amount of perturbation introduced by repair low by focusing on fields that conceivably would have been modified by a correct implementation. Empirical evaluation of Cobbler shows it often achieves the exact output of the deployed implementation in the absence of any bugs. Our experiments show that for small to moderate instantiations of data structures, Cobbler provides repaired data structures which are 100% to 90% similar to the correct structure in more than 90% of the cases. Cobbler also finds and repairs a previously unknown error in ANTLR.

Fourth, integration of the repair framework with commonly used frameworks such as Java Virtual Machines (JVM) and SAT solvers adds to the usability of repair systems. Cobbler lays between the JVM and the Java program

and makes use of barrier logs provided by the virtual machine. The layers use shared memory to communicate. This design enhances the portability of our framework and makes it independent of JVM and the program. Furthermore, Cobble utilizes UNSAT cores provided by SAT solvers which also improve usability by pointing out the user to corrupted parts of the state, as well as parts of the contract which cannot be satisfied on the current state.

1.3.3 Abstract Repair History

Repair abstractions [107, 65] are a technique for abstracting and memoizing concrete repair actions for future reuse when a similar data structure error occurs, thereby prioritizing repair actions and likely pruning the space of structures to explore before a fix is found. Whereas history-aware repair concentrates on the program execution trace and the history of current repair attempts, repair abstractions provide an abstract summary of successful repairs on *previous errors*.

We implemented the idea of abstracting and reusing repair actions in a tool that we call DREAM (Data structure Repair using Efficient Abstraction Methods). DREAM provides a generic repair abstraction interface to be used in conjunction with different underlying repair frameworks. Experimental evaluation of DREAM, when used on top of Cobble, reveals how abstracting and reusing repair actions improves repair performance (e.g., an average of 3,000 times speedup on considered errors of a linked list), and its scalability (e.g., repairing linked lists of 500 nodes in a fraction of a second), without

compromising effectiveness (e.g., producing the exact same repair as Cobbler when repairing linked lists). Finally, DREAM provides a generic interface, which interacts with any repair framework, to obtain higher usability.

1.4 Structure Generation Problem in Testing and Repair

The central technical problem in data structure repair is the construction of a valid data structure (given an erroneous data structure). This technical problem reduces to the problem of constraint-based structure generation [15, 67]—test input generation leverages constraint solving to *enumerate* solutions that are refined as tests, whereas data structure repair leverages constraint solving to generate *a* solution that repairs the erroneous state. We present a novel technique for efficient test input generation [106] based on *dynamic programming*—a problem solving methodology designed to exploit common subproblems. We also discuss ideas on how it enables efficient data structure repair.

1.4.1 Test Input Generation Using Dynamic Programming

Constraint-based test input generation is an effective technique for testing programs, such as compilers and web browsers, which have complex inputs [15, 67, 37, 27]. Constraints are used to define desired inputs and are solved using off-the-shelf systematic constraint solvers and then refined as test inputs. Efficient and scalable constraint-based test input generation, however,

remains a challenging problem.

We present a novel input generation technique that takes constraints written as recursive predicates in the underlying programming language and uses dynamic programming to solve the constraints efficiently. Our key insight is to leverage the recursive structure of desired inputs and partition the problem of generating an input into several sub-problems of generating smaller inputs that exhibit the same structure, and then to use dynamic programming to combine them. (e.g., Figure 1.2 shows how two binary trees can be combined to build a bigger one.) A lazy initialization strategy and symbolic execution optimize our basic technique. Our technique provides not only bounded exhaustive input generation but also enables random input generation.

We argue that the dynamic programming algorithm is sound and complete, and show the experimental results of generating test inputs for a variety of subject programs. We demonstrate how our technique outperforms *Korat* (an efficient solver for structural constraints) and *Pex* (a state-of-the-art tool for symbolic execution). Finally, we use our technique to test Apple Safari and Google Chrome web browsers and efficiently find three bugs in the production version of Chrome.

1.4.2 Ideas for Repair Using Dynamic Programming

We discuss ideas on the unification of test input generation and repair problems. An efficient constraint-based test input generation scheme can be utilized to perform data structure repair by providing small pre-generated



Figure 1.2: An example of combining binary trees to build a bigger binary tree.

substructures to be patched onto the faulty data structure. Recursive checks and memoization [86] localize the error by recursively applying checks on the smaller sections of the data structure. Dynamic programming suggests small data structures to replace the infected part. Therefore, repair does not affect the correct parts of the state and becomes more efficient, scalable, and effective. With respect to usability, this technique gives the user the alternative way of describing the contracts recursively. Also, this idea makes the repair framework more usable since we amortize the overhead of writing and maintaining contracts between test input generation and repair.

1.5 Usability

Enhancing the usability of repair is a critical part of applying it on real world applications. In our comprehensive repair framework, the user writes the contracts in Alloy. The framework uses the contracts to monitor deployed software and repair faulty executions. Furthermore, it provides repair abstraction logs to help the user debug and permanently fix the bug. Write and

read barrier logs and UNSAT cores are other examples of reports from the repair framework which the user might find useful for debugging. Using such reports, data structure repair will be, in turn, useful in program repair. To achieve more useful and usable data structure repair, we presented three tools that implement these ideas and facilitate fulfilling the promise of repair for real programs: Tarmeem¹, Cobbler, and DREAM.

One final challenge in the face of repair is to make it non-intrusive and lightweight in the absence of errors. Frequent checks of contracts through a SAT solver (e.g., as done in Cobbler) diminishes the usability of repair because the time it takes is of the same order of magnitude as of repair. We build on the idea of translating contracts from the Alloy specification language to Java [8] and present an extension, called *Arreh*, to the Alloy 4 tool-set. Arreh receives a model in Alloy, translates its commands to Java methods, and checks them using the Java Virtual Machine instead of a SAT solver to improve the efficiency of checking contracts.

1.6 Contributions

The results in this dissertation are based on published papers at TACAS 2012 [103], FSE 2012 [106], and RV 2013 [107]. We make the following contributions:

- The basic idea of **contract-based data structure repair** was introduced in my Master’s thesis [105, 104, 102]. This PhD dissertation

¹As a part of my Master’s thesis [105, 104, 102].

presents a comprehensive framework that embodies a practical and scalable approach for error recovery using data structure repair.

- **History-aware contract-based repair** combines the program’s dynamic behavior with contracts and the current erroneous state of a program to perform repair.
- **Read and write barriers for repair** are an unconventional use of barriers to obtain program execution history for repair.
- **Minimal unsatisfiable cores** provided by SAT solvers help to reduce the search space when a field outside the execution trace should be modified to repair data structures.
- The basic idea of **repair abstraction** was introduced in Malik’s PhD proposal [65]. We develop the idea in the context of contract-based data structure repair using Alloy.
- **Dynamic programming for input generation** is a novel technique that generates structurally complex inputs. We further optimize this technique with lazy initialization and symbolic execution. We discuss ideas for using dynamic programming for data structure repair.
- **Recursive predicates in constraint-based generation and contract-based repair of complex structures** facilitate predicate formulation and enable faster input generation and more accurate repair.

- We **implement** a tool-suite for data structure repair and test input generation. **Cobbler** is an automated portable framework for repairing Java programs that enhances real applications with repair functionality, and is based on history-aware data structure repair. Experimental evaluation shows that Cobbler efficiently and accurately repairs text-book examples and real world programs. **DREAM** provides a generic framework that can be embodied by different data structure repair techniques and implements the idea of repair abstractions using Alloy. **Arreh** is a tool that extends Alloy Analyzer and translates Alloy checks to Java methods, to significantly reduce the burden of constantly checking contracts.
- We present rigorous experimental **evaluation** of our repair and test input generation frameworks using case studies, open-source projects, and production software. Experimental results, using microbenchmarks and two open source programs (Kodkod solver and ANTLR), show that our repair techniques improve repair efficiency, scalability, and effectiveness. Moreover, Cobbler finds and repairs a previously unknown bug in ANTLR. Experimental results also show that our test generation technique improves input generation performance and scalability for microbenchmarks over state-of-the-art testing tools Pex and Korat. The test generation technique when applied to test two web browsers, Apple Safari and Google Chrome, finds three known and unknown errors in the production version of Chrome, which are now fixed.

The ideas presented in this dissertation improve the efficiency, scalability, effectiveness, and usability of data structure repair. More efficient and effective repair facilitates the use of repair in real world applications and enhances software reliability. Unification of test input generation and repair makes repair an even more attractive option to improve our ability to produce and maintain reliable software.

Chapter 2

Related Work

Our work builds on two research threads and unifies them: Section 2.1 covers previous work on data structure repair, and Section 2.2 provides background on constraint-based test input generation.

2.1 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` [4] and `chkdsk`[6], database checkpointing, and rollback techniques are standard repair routines used to monitor and correct system state at runtime.

Some commercially developed systems, such as the IBM MVS operating system [75] and the Lucent 5ESS telephone switch [45], have dedicated routines for monitoring and maintaining properties of their data structures. These repair routines suffer from the limitation of being too specific and tailor-made for their system structures and hence cannot be generalized as data structure repair tools.

Demsky and Rinard [29, 30] pioneered the idea of general purpose data

structure repair with **constraint-based repair**. Users write declarative constraints. The system translates the constraints into disjunctive normal form and solves them using an ad hoc search.

The **assertion-based repair** technique [91, 31] implemented in the **Juzi** tool [31] detects errors by asserting user defined repOK methods which hold the class invariants (aka data structure integrity constraints). 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. The limitation of this technique is that class invariants hold at the entry and exit points of all public methods. The tool alters the faulty data structure to produce an arbitrary state which may satisfy the integrity constraints but may be very different from the intended output of the method. This may adversely affect the functionality of the system as a whole. For instance, the output of a faulty binary tree insert method, could get converted into an empty tree which may be a valid structure satisfying the invariants but is an unlikely output of the insert algorithm. A post-condition Java predicate could be asserted along with the repOK method to solve this problem. But as the size and complexity of properties and the size of the data structure increases, such techniques would not scale well. The repair precision and efficiency is also heavily impacted by the order in which the repOK method checks different properties or accesses different fields.

Several techniques improve assertion-based repair: e.g., STARC and DSDSR. STARC [34] uses static analysis of the repOK method to identify re-

current fields, i.e., fields that are accessed to merely traverse the data structure, and local field constraints, i.e., constraints that relate the value of neighboring objects. STARC uses the result of this static analysis to prioritize repair actions and prune the search space.

Dynamic Symbolic Data Structure Repair [50] (DSDSR) extends assertion-based repair by producing a symbolic representation of fields and objects along the path executed in repOK. DSDSR builds the path constraint required to take the current path in repOK. When repOK returns false, DSDSR 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.

Our previous work [107, 65] applies the idea of abstracting and reusing repair actions in the context of assertion-based repair. Previous successful repairs are abstracted to prioritize repair actions Juzi takes, in order to improve repair efficiency.

While assertion-based repair is geared toward data structure invariants, the Plan B approach and its tool **PBnJ** [82], similarly to our contract-driven repair, support data structure invariants as well as method pre- and post-condition specifications. The user writes these specifications in a declarative first order relational logic extension to Java that is similar to Alloy. The system then translates these specifications into Java predicate methods invocable from other methods. These Java methods are used for checking properties of data structures, similar to the basic idea of our tool Arreh. Once a check fails,

PBnJ falls back on *executing* the specifications: i.e., it ignores the Java implementation and uses a SAT solver to generate a data structure that satisfies both invariants and method post-conditions. However, PBnJ suffers from low repair performance, as it completely ignores the Java code, the execution history of the program, the previous repair actions, and the current faulty data structure.

Automatic workaround [20, 19] is another recent technique which exploits the inherent redundancy of software components to avoid failures. When a failure is detected, the state and data structures are rolled back to a checkpoint. The system then dynamically changes the code to call a different library method (instead of the one that failed) which includes a potential workaround. This technique makes the assumption that several methods implement the same logic and are indicated equivalent in library equivalence specifications.

Ditto [86] is a framework that uses write barriers to recursively check the invariants. Ditto leverages recursive checks to incrementally assert structural integrity constraints of data structures. It only re-evaluates those parts of the data structure that have been modified since the last evaluation and are logged by the write barrier. Ditto justifies this method by observing that the invariants still hold on the unchanged parts of the data structure. In contrast, our framework does not use write barrier and recursion to *only check* the invariants but also to *repair* the data structure.

We would like to highlight that our technique differs from the class of **automated debugging** and **program repair** techniques [97, 10, 69, 90, 98,

53, 79, 22, 83, 76, 96, 57, 62, 43] which are intended to be used in the testing phase prior to the deployment of software. However, as Malik et al. propose [66], dynamically performed data structure repair actions could translate into program statements thus aiding in program repair. They could act as an input to program repair frameworks such as the AUTO E-FIX tool[97], providing useful information regarding the differences between faulty and correct concrete program states.

2.2 Test Input Generation

The importance of using specifications in testing has long been recognized [42]. Several projects automate test generation from specifications in various languages [48, 77]. The specific use of logical constraints to represent inputs dates back at least three decades [24, 44, 49, 60, 80]. But a focus of prior work has been to solve constraints on primitives, and not on complex structures—which require very different constraint solving techniques. **Korat** [15] and **TestEra** [67] are among the first frameworks to provide systematic generation of structurally complex tests from constraints. Following this spirit of systematic black-box testing, **ASTGen** [27] and **UDITA** [37] are two more recent frameworks, which have been used successfully to find bugs in real applications, including refactoring engines. **ASTGen** requires the user to write imperative test input generators, whose executions produce input programs for refactoring engines. **ASTGen** bears some similarities to our test generation framework in composing test generators to build bigger inputs.

However, ASTGen is limited to testing refactoring engines and requires the user to explicitly specify *how* to generate test inputs. UDITA provides a programming language to describe test inputs using a combination of declarative and imperative styles, where constraint solving is used in conjunction with partial generators.

Lava [88] and QuickCheck [23] can also provide generation of complex structures. Lava requires the user to describe inputs using a production grammar and generates strings in the grammar, but cannot handle complex constraints, such as those of a red-black tree. QuickCheck requires the user to write a generator for complex inputs and generates random inputs for testing functional programs in Haskell using a technique similar to our recursion with lazy initialization. Similarly, Gast [59] generates tests for programs written in functional languages. However, QuickCheck and Gast use pure top-down recursion and not dynamic programming.

To our knowledge, our work on test input generation is the first use of dynamic programming for test input generation. The general principle of memoization, which is a central idea in dynamic programming, has previously been used in the context of bug finding [46, 86, 100, 17, 41], albeit without the specific framework of dynamic programming. Dynamic programming has previously been applied in the context of runtime verification to generate *monitors* from formal specifications [47].

Several tools use method sequences for testing object-oriented programs, and can generate complex structures using systematic [99] or random-

ized exploration [78]. While these tools allow unit testing, they cannot feasibly generate inputs that are parsed from strings with semantic and syntactic constraints, e.g., XML files, which our constraint-based test generation handles readily.

As a part of our test generation technique, we leverage **symbolic execution**. The recent advances in constraint solving technology [11, 28] have led to a rebirth of symbolic execution [58, 24]—a powerful program analysis technique that was traditionally used for checking small programs with primitive types. Generalized symbolic execution [56] implements Korat using the Java PathFinder model checker [95] and supports structural constraints using symbolic execution. Guiding symbolic execution using concrete executions is rapidly gaining popularity as a means of scaling it up in several recent frameworks, most notably DART [40], CUTE [85], EXE [18], and **Pex** [92]. While DART and EXE focus on properties of primitives and arrays to check for security bugs, such as buffer overflows, CUTE and Pex support the use of preconditions in white-box testing. Compositional techniques for symbolic execution, introduced by PREFIX and PREFIXfast [16], can handle larger code bases but they do not currently handle complex structural properties [39]. Our work provides a novel way to scale symbolic execution by applying it with dynamic programming in synergy.

Chapter 3

Background: Contract-Based Data Structure Repair Using Alloy

This chapter provides necessary background on our previous work on constraint-based repair [105, 104, 102]. We first give a motivating example (Section 3.1), which is followed by basics of the Alloy tool-set (Section 3.2), and then we describe our previous technique (Section 3.3).

3.1 Example

In this section, we illustrate a motivating example to describe data structure repair algorithms. This data structure is a binary search tree of unique integers. Listings 3.1 and 3.2 show the data structure and its `remove` method in Java.

Listing 3.3 demonstrates a model of the binary search tree data structure in Alloy [51]—a relational first order logic language suitable for expressing software designs—which we use for writing specifications. The `repOK` method¹ describes all method-independent constraints and include acyclicity,

¹Predicates that check constraints, especially class invariants, are traditionally called *repOK* methods [64].

```

1 class BinarySearchTree {
2     Node root;
3     int btSize;
4
5     boolean remove(int x) {
6         if (root == null)
7             return false;
8         else {
9             boolean result;
10            if (root.element == x) {
11                Node auxRoot = new Node();
12                auxRoot.left = root;
13                result = root.remove(x, auxRoot);
14                root = auxRoot.left;
15            } else {
16                result = root.remove(x, null);
17            }
18            if (result) //using uniqueness of elements
19                btSize--;
20            return result;
21        }
22    }
23 }

```

Listing 3.1: A binary search tree implementation in Java [2].

search property of binary search trees, correctness of size, and that elements are unique. The user may also express method post-conditions, as shown in `remove_postcondition`. This post-condition specifies a correct `remove` with respect to the data structure and the return value from the `remove` method.

Alloy represents Java classes with signatures (e.g., `sig BinarySearchTree` in Listing 3.3) and field relations with a relational view. The keywords `lone` and `one` for a unary relation denote that the relation may or must not be empty, respectively. Binary relations can be defined as total or partial functions among other options (e.g., `right` is a partial function). We use the syntactic sugar of adding back-tick (‘’) to distinguish post-state Alloy relations from pre-state relations. The Alloy `repOK` predicate (`pred`) expresses

```

1 class Node {
2     Node left, right;
3     int element;
4
5     boolean remove(int x, Node parent) {
6         if (x < element) {
7             if (left != null)
8                 return left.remove(x, this);
9             else
10                return false;
11        } else if (x > element) {
12            if (right != null)
13                return right.remove(x, this);
14            else
15                return false;
16        } else {//x == element
17            if (left != null && right != null) {
18                element = right.minNode().element;
19                right.remove(element, this);
20            } else if (parent.left == this) {
21                if (left != null)
22                    parent.left = left;
23                else
24                    parent.left = right;
25            } else if (parent.right == this) {
26                if (left != null)
27                    parent.right = left;
28                //to introduce bug cycle replace with
29                //left.right = parent
30            } else
31                parent.right = right;
32        }
33        return true;
34    }
35 }
36
37 Node minNode() {
38     if (left == null) return this;
39     else return left.minNode();
40 }
41 }

```

Listing 3.2: A binary search tree node implementation in Java [2].

data structural integrity rules. For instance, the directed acyclicity constraint specifies that for any node reachable from `root` by applying zero or more `left` or `right` pointers, the node cannot reach itself by following one or more `left` or `right` pointers, so it cannot traverse a cycle. `*` and `^` represent “zero or more” and “one or more” applications of a relation. Alloy supports membership, cardinality, and complement, `in`, `#`, and `-` respectively as in the acyclicity, size, and correct remove constraints. Quantifiers \forall and \exists have their usual meaning and are expressed with the keywords `all` and `some`. The expressions `not (!)` and `implies (=>)` have their expected meaning in first order logic.

To illustrate our repair process, consider the following bug introduced to the Java program to make a faulty implementation:

- **Bug cycle:** In Listing 3.2 line number 27, the programmer wrongly puts `left.right = parent` instead of `parent.right = left`.

This bug can be manifested by calling `remove` using the faulty implementation on some inputs. Figure 3.1 shows the result of executing the faulty implementation on a bug revealing input and some possible repair results.

3.2 Background on Alloy

This section uses the example to describes necessary background on the Alloy tool-set, which our repair frameworks use.

```

1 one sig True, False {}
2 abstract sig BinarySearchTree {
3   root: lone Node,
4   root': lone Node,
5   btSize: one Int,
6   btSize': one Int
7 }
8 abstract sig Node{
9   left: lone Node,
10  left': lone Node,
11  right: lone Node,
12  right': lone Node,
13  element: lone Int,
14  element': lone Int
15 }
16 pred repOK(t: BinarySearchTree){ //class invariant
17   //directed acyclicity
18   all n: t.root'.*(left'+right') | n !in n.^(left'+right')
19   //search property
20   all n,m: t.root'.*(right'+left') |
21     m in n.left'.*(right'+left') =>
22     int m.element' < int n.element'
23   all n,m: t.root'.*(right'+left') |
24     m in n.right'.*(right'+left') =>
25     int m.element' > int n.element'
26   //size OK
27   # t.root'.*(left'+right') = int t.btSize'
28   //unique elements
29   all n, m: t.root'.*(left'+right') |
30     int n.element' = int m.element' => n = m
31 }
32 pred remove_postcondition(This: BinarySearchTree, x: Int, removeResult: (True
33   +False)){
34   repOK[This]
35   //correct remove
36   This.root.*(right+left).element - x =
37     This.root'.*(right'+left').element'
38   //correct remove result
39   x in This.root.*(right+left).element <=> removeResult in True

```

Listing 3.3: Binary search tree contract specification in Alloy.

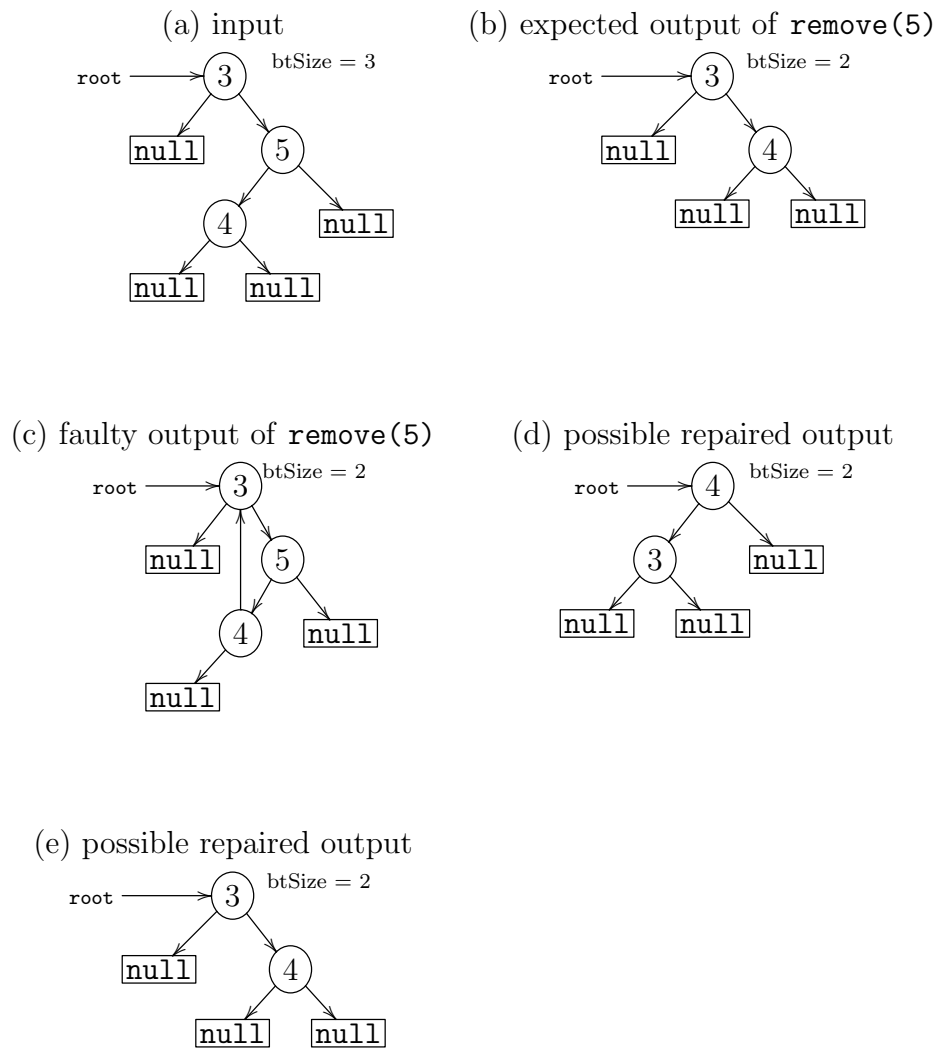
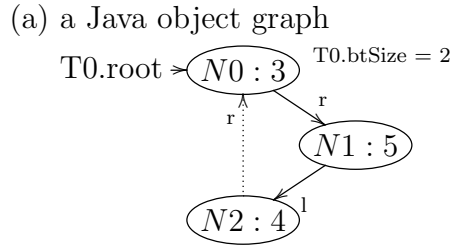


Figure 3.1: Bug cycle manifested as a faulty output and the repair result.

Alloy is a relational first order logic language [51]. An Alloy model (e.g., Listing 3.3) consists of relations and constraints on them. The Alloy Analyzer performs bounded exhaustive analysis of Alloy models. A *bound* is a function which maps each relation to a set of tuples ($\text{bound}: R \rightarrow 2^T$), where each tuple



(b) relational representation

$$\text{inst}(\text{root}) = \{(T0, N0)\}$$

$$\text{inst}(\text{btSize}) = \{(T0, 2)\}$$

$$\text{inst}(\text{right}) = \{(N0, N1), (N2, N0)\}$$

$$\text{inst}(\text{left}) = \{(N1, N2)\}$$

$$\text{inst}(\text{element}) = \{(N0, 3), (N1, 5), (N2, 4)\}$$

(c) relaxing the dotted edge

$$\text{LB}(\text{right}) = \{(N0, N1)\}$$

$$\text{UB}(\text{right}) = \{(N0, N1), (N2, N0), (N2, N1), (N2, N2)\}$$

Figure 3.2: Relational representation of data structures in Alloy models.

consists of atoms. For each relation R , two sets are defined: a lower bound $LB(R)$, which includes all tuples that R *must* have in its instance ($\text{inst}(R)$), and an upper bound $UB(R)$, which includes all tuples that R *may* have in its instance. Therefore, $LB(R) \subseteq \text{inst}(R) \subseteq UB(R)$. Figure 3.2 (b) shows the relational representation of the Java object graph shown in Figure 3.2 (a).

We use Kodkod [93], the back-end of Alloy Analyzer, which is a SAT-based constraint solver for first order logic that supports relations, transitive closure, and partial models. Kodkod provides a finite model for satisfiable specifications and an UNSAT core for unsatisfiable ones. To perform repair,

Kodkod suggests mutations to the data structure such that it meets the Alloy specification. Specifically, given a satisfiable relational formula and the bounds, Kodkod uses a backtracking search to find a satisfying instance. The search space is typically exponential in the number of atoms.

Kodkod allows explicit specification of upper and lower bounds for analysis, which provides partial solutions and restricts the search space. We use this functionality to specify which fields of the state can be mutated by the SAT solver to perform repair. Thus, to *relax* a field in Kodkod means to let the SAT solver suggest different values other than the one present in the faulty post-state, in order to find a satisfiable answer. Relaxing a field, which is a mutation of a field of a specific object, is done through binding a relation to suitable lower and upper bounds. For example, in Figure 3.2 (a) the dotted edge can be relaxed by setting the lower and upper bounds as shown in Figure 3.2 (c). Setting both lower and upper bounds to the same set makes it the only answer for that relation, i.e., the set becomes a partial solution for the Kodkod model.

3.3 Our Previous Work: Contract-Based Data Structure Repair Using Alloy

Our first work [105, 104, 102] introduced a contract-based approach to data structure repair. This work was presented as a Master’s thesis and we summarize it here. The key novelty was the support for rich behavioral specifications, such as those that relate pre-states with post-states of the method

to accurately specify expected behavior and enable precise repair.

Method contracts naturally suit the repair process since they introduce class invariants, as well as pre- and post-conditions, and they are widely used for other phases of software development. While our repair framework introduced in this work—Tarmeem²—allows specifications to be provided in different modeling languages, we translate them to Alloy and leverage Alloy Analyzer to systematically repair erroneous states. Four different heuristics were presented and implemented in this work. Additionally, this work formally defined repair and used edit distance for graph similarity to compute the effect of repair on an erroneous program state. We take a relational view of the program heap, and view data structures as edge-labeled graphs. This view enables using edit distance—defined as the minimum number of edge additions/deletions to change a graph to another—as a metric for computing the perturbation of the erroneous program state, which undergoes repair. Our algorithms attempt to keep the perturbation to a minimum.

Tarmeem instruments the Java program to record pre- and post- states. When a failure occurs during execution (i.e., a contract check fails), Tarmeem invokes a repair algorithm to let some fields of the data structure be modifiable by the SAT solver (relaxed), and uses SAT to compute values for those fields. We proposed four different repair heuristics in this work.

1. **Basic method** uses the pre-state but is oblivious to the erroneous post-

²Tarmeem means repair in Persian (Farsi).

state. Although this approach provides a correct output, it has high performance penalty and can possibly introduce unnecessary perturbation to the data structure. Considering our example of faulty remove, the basic method only uses Figure 3.1 (a) and not Figure 3.1 (c) to perform repair and it might produce Figure 3.1 (d) which is, although an acceptable answer, rather different than what the correct implementation produces (Figure 3.1 (b)).

2. **Iterative relaxation** aims to optimize performance when the number of errors is relatively small. A deployed system that has been well-tested can be assumed to have few errors. This heuristic iteratively calls SAT allowing it to modify one relation, two relations and so forth. Note that this heuristic uses a relation-based relaxation by which we mean it allows *all* fields of the same name (e.g., `right` fields of all nodes in a binary search tree) to be determined by SAT. In contrast, the edge-based relaxation has finer granularity and allows SAT to determine a specific field of a specific object. Edge-based relaxation is used in our second work (Chapter 4) to improve accuracy and performance of repair. Iterative relaxation, when applied on Figure 3.1 (c), explores different relations to finally pick `right` to be relaxed to obtain Figure 3.1 (e). Here, relaxation of one relation (`right`) suffices, but two `right` edges should be mutated by the SAT solver.
3. **Error localization** uses the post-condition to isolate erroneous parts of the output. It focuses on those parts of the specification that are

violated by the erroneous post-state and selects relations used in those parts to perform proper relaxation. In the case of the faulty remove example, it focuses on the violated constraints, namely acyclicity and correct remove post-conditions (Listing 3.3), and also search property and size constraints as byproducts, to pick relations and relax them upon SAT invocation. These constraints include `root`, `left`, `right`, `btSize`, and `element` in the post-state. Error localization does not assume few errors in the data structure, but is not very effective and might give as a perturbed result as Figure 3.1 (d).

4. **Guided error localization** builds on top of error localization and leverages user provided guides to more accurately determine the faulty parts of the data structure. A guide specifies which parts of the data structure are subject to test by a specific specification as opposed to which relations are just used and not validated in that specification (e.g., the size constraint checks `btSize` but uses `root`, `right` and `left` to traverse the tree). The guide set provides a hint to prioritize relaxations. For instance, in Figure 3.1 (c) the following constraints are violated: acyclicity (checks `left` and `right`), search property (checks `left`, `right` and `element`), size (checks `btSize`) and correct remove (checks `left`, `right` and `element`). Relaxation of these relations guides the repair process toward producing Figure 3.1 (e).

We evaluated Tarmeem using a text-book data structure (singly-linked

list) and an open-source application (ANTLR which is a part of the DaCapo benchmark [12]). We injected errors into the source codes to mimic several types of errors. We measured the effectiveness of the repair process by looking at the edit distance between the repaired data structure and the faulty one. We also measured the efficiency of repair by measuring time to repair as well as other SAT-related metrics. Tarmeem is capable of repairing errors of singly-linked lists of up to 20 nodes in 15 seconds (worst case). The framework repairs errors injected into an ANTLR tree of 30 nodes in at most 15 seconds. The best heuristics are iterative relaxation when the assumption of presence of relatively few errors holds and guided error localization in general.

While the results of Tarmeem were encouraging and showed the feasibility of repairing complex structures of small sizes with a small number of faults, our use of SAT represented a bottleneck for scaling the algorithms to larger structures. Relying solely on the contracts and not obtaining any information from the faulty code left this approach to repair with a huge search space of all possible candidates for repair. Therefore, scalability remained a key technical challenge.

Chapter 4

History-Aware Data Structure Repair Using SAT

Our previous work, Tarmeem, showed the feasibility of the basic idea of contract-based data structure repair. Repair performance and scalability, however, remained as technical challenges. In this chapter, we present a novel technique, history-aware data structure repair, for improving the scalability and efficiency of repair. This chapter is based on our TACAS 2012 paper [103].

Recall that the foundation of contract-driven data structure repair is to use class invariants and method post-conditions to detect erroneous executions and perform repair. Although the post-condition specifies the expected behavior of the method, there is often a wide range of correct possibilities for a given input since there may be many ways to implement the same specification. Additionally, for a SAT-based repair framework (e.g., Tarmeem), relaxing all fields of the data structure and letting the SAT solver mutate them explodes the search space and is infeasible for real world applications.

Our insight into history-aware data structure repair is two-fold: (1) **execution history**: the dynamic program trace of field writes and reads provides useful guidance to identify incorrect state mutations made by a faulty

program; and (2) **SAT solving history**: the unsatisfiable core generated by a SAT run captures core elements of the solver’s reasoning, which not only facilitates locating faults but can even be leveraged directly to optimize a successive SAT run.

In history-aware data structure repair, we first use the program execution history through reads and writes to guide the repair process. In deployed software, the program is expected to contain most of the intended logic. Furthermore, given sufficient pre-deployment testing, there should not be many bugs in the code. By observing the dynamic behavior of a faulty execution, we can substantially reduce the size of the search space and make the repair process more efficient and effective. The core idea is to focus on fields modified and/or read during the execution. To obtain the execution history, we record write and read actions performed by the program. Our implementation instruments the program, but alternatively the Java Virtual Machine could efficiently provide them through barriers [13] (more details in Section 4.1). We start by restricting the SAT solver to correcting written fields and values, followed by read fields during the execution, heuristically trying to find a repaired data structure by mutating only those fields. However, there exist cases in which we have to perform a broader search and consider fields not included in the execution trace. In such cases, we take advantage of *UNSAT* cores (more details in Section 4.2). If the SAT solver cannot find a satisfying solution by mutating only written and read fields, we then utilize the *UNSAT* core provided by the failing SAT invocation to identify and mutate faulty fields

of the data structure.

Listing 4.1 shows the repair algorithm in pseudo-code. If an assertion is violated, the repair framework initially only mutates (relaxes) fields in the write log, holding all other data structure fields constant (through providing a partial solution for the SAT solver). It then calls the SAT solver to compute correct values for the relaxed fields. If this step does not yield a structure satisfying the contracts, the next step relaxes the fields in the read and write logs. If it still is unsuccessful, it relaxes fields appearing in the UNSAT core. If the SAT solver finds no solution, there is an inconsistency in the contract itself which the repair framework reports.

```
1  if(!assertContracts()){
2      relaxSAT(writeBarrierLog);
3      if(!assertContracts()){
4          relaxSAT(writeBarrierLog, readBarrierLog);
5          if(!assertContracts()){
6              relaxSAT(unsatCoreFields);
7              if(!assertContracts()){
8                  reportModelInconsistency();}}}}
```

Listing 4.1: History-aware contract-based repair using read and write logs and unsatisfiable cores.

4.1 Using Barriers for Data Structure Repair

We use instrumentation or Java Virtual Machine barriers for logging write and read activities of programs under repair. A barrier is a code sequence that performs an action just prior to a write or read. Languages with automatic memory management, such as Java, widely support such barriers. Commonly-used generational garbage collectors, all incremental collectors, and

concurrent collectors require *write barriers*[13, 21, 63, 9, 14, 94]. For example, write barriers record pointers between regions for independent generational collection, and detect concurrently updated objects for completeness in concurrent collectors. Barriers are widely available in commercial and research implementations of managed languages, e.g., the HotSpot, J9, JRockit, and Jikes RVM Java Virtual Machines, and the .NET C# system. Our approach for data structure repair inserts barrier instrumentation on writes and reads or piggybacks on existing barriers. Here, we assume that read/write barriers are available for both pointer and non-pointer load and stores, although traditionally barriers are more widely implemented for pointers.

4.2 Using UNSAT Cores for Data Structure Repair

We benefit from UNSAT cores in history-aware data structure repair. UNSAT cores are minimal unsatisfiable sub-formulas provided by failed SAT invocations. If the SAT solver cannot satisfy the constraints in a model, it produces a minimal unsatisfiable core, which is a subset of the constraints of the model. Given an unsatisfiable CNF formula X , a minimal unsatisfiable sub-formula is a subset of X 's clauses that is both unsatisfiable and minimal, which means any subset of it is satisfiable. There could be many independent reasons for a formula's unsatisfiability and hence more than one minimal core. Kodkod, the back-end of Alloy Analyzer, provides UNSAT cores after failed SAT invocations. The Recycling Core Extractor algorithm, implemented as the RCE Strategy in Kodkod, returns an unsatisfiable core of specifications

written in the Alloy language that is guaranteed to be sound (constraints not included in the core are irrelevant to the unsatisfiability proof) and irreducible (removal of any constraint from the set would make the remaining formula satisfiable).

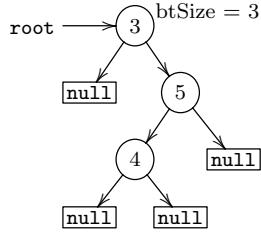
4.3 Illustration of History-Aware Data Structure Repair

To illustrate history-aware repair, consider bug `cycle` from Section 3.1. When using the incorrect implementation, after the method returns, checking the conjunction of `repOK` and the method post-condition indicates that there is an error, triggering the repair process.

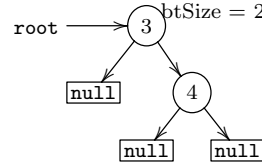
To repair the erroneous output of the `cycle` faulty implementation, constraint-based repair methods [30, 55, 50] observe the cycle and remove it from Figure 4.1(c) to produce Figure 4.1(a), but fail to remove node 5. Contract-based repair techniques *without history* [104, 82] (e.g., Tarmeem) may generate Figure 4.1(d), which although a correct output, is different from what the program would have been generated in the absence of any bugs.

History-aware contract-based repair first invokes the SAT solver and tries to find a solution by only changing the values of the fields which the program writes into during the execution (Figure 4.1 (e)). These fields are distinguished by dotted lines in the faulty output. In this invocation, the SAT solver does not find a solution because the program failed to update some fields that need to be modified. Our history-aware repair framework next considers

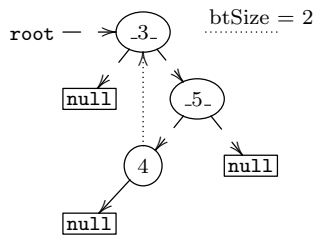
(a) input
constraint-based repair



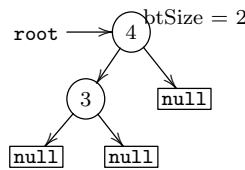
(b) expected output of `remove(5)`
history-aware contract-based repair



(c) faulty output of `remove(5)`



(d) contract-based repair



(e) write barrier log (dotted lines in part (c)):
 $\{[4].\text{right}, \text{btSize}\}$, $[x]$ represents the node with value x before execution.

(f) read barrier log (dashed lines in part (c)):
 $\{\text{root}, [3].\text{element}, [3].\text{right}, [5].\text{element}, [5].\text{left}, [5].\text{right}, [3].\text{left}\}$

Figure 4.1: `cycle` manifested as a faulty output and its history-aware repair result.

changing fields read by the program (Figure 4.1 (f)) and shown as dashed lines. It invokes SAT to find suitable replacements for the fields written or read by the program. This invocation produces a repaired structure as shown in Figure 4.1 (b), which is identical to the expected output. Utilizing the barrier logs keeps us from generating Figure 4.1 (d) since the `left` field of

node 4 is not relaxed and is held constant to be null. However, there remains a chance that a field that was not touched at all during the execution needs to be changed. Our repair framework obtains an UNSAT core from the previous SAT invocations. The UNSAT core is the conjunction of contradicting `repOK` and post-condition specifications, which were not satisfiable at the same time. In this example, if we were to proceed to the third SAT call, the UNSAT core would not include, for example, the `correct remove result` post-condition. Therefore, the final invocation of SAT would not relax the `removeResult` field.

4.4 Cobbler: Implementation of History-Aware Repair

We implemented the above history-aware repair algorithm in Cobbler¹, which repairs Java programs. Like Tarmeem, Cobbler uses the Alloy tool-set and its Kodkod back-end.

Cobbler works as follows: (1) The user provides the Java data structure class and its methods. Cobbler instruments this code with setters and getters to obtain logs of writes and reads. Cobbler also instruments the program for our experiments to measure the repair time, edit distance and other metrics. (2) Cobbler generates a stub for the `repOK` and method post-conditions for the Java class. Cobbler extracts class-specific relations, types, and properties into the stubs, and the user enhances them with the application specific logic. (3) Cobbler then instruments the program to check the post-conditions and

¹Cobbler means a person who repairs shoes.

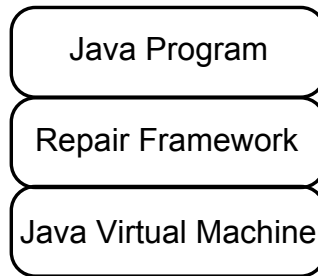


Figure 4.2: The relationship between Cobbler, the Java Virtual Machine, and the program.

call the repair method when needed. (4) The user executes the Java program inside the Cobbler framework.

Figure 4.2 shows how the repair framework sits on top of the Java Virtual Machine and executes the Java program. The layers use shared memory to communicate. This design enhances the portability of our framework and makes it independent of JVM and the program. Alternative implementations could implement the framework inside the JVM, which would lower the overhead when programs are correct. When programs need to be repaired, the SAT solving time is orders of magnitude bigger than time saved by merging the repair framework into JVM.

4.5 Cobbler Evaluation

The objectives of our evaluation are to empirically validate the hypothesis that using execution history and UNSAT cores improves the efficiency, accuracy, and scalability of contract-based repair with SAT solvers. To this end, we simulated various errors in microbenchmarks and examined two real

world applications: Kodkod [93] and ANTLR [1, 12]. Cobbler discovered a previously unreported bug in the `addChild` method of ANTLR version 3.3 that resulted in a cycle in the output Tree. Our repair algorithm fixes this error accurately for a Tree with 300 nodes within 30 seconds.

Throughout the evaluation, we ran each experiment five times and reported the averages. All the experiments used a 2.50GHz Core 2 Duo processor with 4.00GB RAM running Windows 7 and Sun’s Java SDK 1.6.0 JVM. All the repair frameworks used their default SAT solvers: Cobbler used MiniSat and MiniSatProver, Tarmeem used DefaultSAT4J, and PBnJ used MiniSat.

4.5.1 Evaluation Metrics

To evaluate the efficiency of repair, we measured: (1) **logging time**: the overhead due to logging read and write actions; (2) **check time**: the time to detect a contract violation; and (3) **repair time**: the time to search and find a repaired data structure.

To evaluate the accuracy of repair, we measure the *edit distance* between the object graphs of the repaired data structure r , and the expected data structure e that a correct implementation would produce. Note that, r satisfies the method contract but might be different from the expected output. We define edit distance as the minimum number of edge additions/deletions to change a graph to another [84, 104]. We create the correct graphs by a separate correct implementation and then measure the edit distance in set difference operations between two graphs using the relational representation

discussed in Section 3.2. Here $inst_i(R)$ is the instance of relation R in data structure i .

Definition 1. $dist(e, r) = \sum_R (|inst_e(R) - inst_r(R)| + |inst_r(R) - inst_e(R)|)$.

The lower this distance, the closer the repaired data structure is to the expected post-state data structure. We define the similarity percentage between the repaired output r and the expected output e as follows:

Definition 2. $sim(e, r) = (1 - \frac{dist(e, r)}{\sum_R |inst_e(R)|}) \times 100$.

4.5.2 Subject Programs

We applied Cobbler to (1) the `remove` method of Singly Linked List, (2) the `insert` method of the `Kodkod.util.ints.IntTree` class of the Kodkod solver implementation, and (3) the `deleteChild` and `addChild` methods of `BaseTree` of ANTLR.

Singly linked list: Linked list is widely used and is a part of libraries such as `java.util.Collection`. The post-condition of the `remove(int value)` method, checks if the method has (1) deleted all nodes with elements equal to the input value, (2) maintained acyclicity, (3) inserted no new nodes, and (4) deleted no other nodes.

Red-black tree of Kodkod: Kodkod [93] is a SAT-based constraint solver for first order logic. It consists of 33,985 lines of Java code in 169 classes. The `IntTree` class with 570 lines of code and 21 methods sits at the core of the Kodkod solver, and is a generic implementation of the red-black tree data

structure. Red-black tree comprises complex data structure invariants which include binary search tree invariants: every node has at most two children, key values of the left subtree are smaller and those of the right subtree are greater than the node value, and the tree is acyclic. In addition, constraints are imposed on the color of each node to keep the tree balanced: every node is either red or black, every leaf node is black, no red node has a red parent and all paths from the root to a descendant leaf contain the same number of black nodes. The `insert` method of this class comprises 58 lines of code with 67 branch statements. The post-condition of the `insert(int newKey)` method checks if an element with the new key value has been added without adding or deleting any other elements.

BaseTree of ANTLR: We use ANTLR (ANother Tool for Language Recognition) from the DaCapo 2009 benchmark suite, version 9.12 [1, 12]. ANTLR builds language parsers, interpreters, compilers, and translators from grammars. It comprises 29,710 lines of Java code, and has a download rate of about 5,000 per month. Rich tree data structures represent language grammars and are the backbone of this application. The abstract class `BaseTree` is a generic tree implementation. A few classes, such as `ParseTree`, extend this data structure. Each `BaseTree` instance maintains a list of successor children. The `childIndex` represents its position in the list. Each child node is a tree and points back to its parent. Every node may contain a token field that represents the payload of the node. Based on the documentation and the program logic, we derived invariants for the `BaseTree` data structure such as acyclicity

through children references, accurate parent-child relationships, and correct values for child indices. The `addChild(Tree node)` and `deleteChild(int childIndex)` methods are the main functions used to build and manipulate all tree structures in ANTLR. The respective post-conditions check that nodes are added or deleted without any unwarranted perturbations to the other nodes.

4.5.3 Errors

Table 4.1 enumerates all the inserted faults and, for ANTLR, a detected error. It explains the errors and displays the violated constraints. The accuracy and performance of the repair algorithm depends on which and how many fields are relaxed in each step, and the number of calls to the solver. The data structure size, size of the log, and size of violated constraint formula influence repair accuracy and efficiency. We explore these parameters with a range of errors that violate different constraints and appear in different program statements, such as incorrect field assignments, incorrect branch conditions, and errors of omission.

The last column in the table indicates if the field(s) that should be corrected appear in the write barrier log (WB), read barrier log (RB), or neither (ALL fields). For instance, in the first faulty linked list `remove` method (Err 1 of SLL `remove`), the header field is wrongly assigned to null. Since the wrongly updated header field appears in the write barrier log and the error lies in the value assigned to it, the tool can repair the data structure by relaxing the fields in the write barrier log alone. The tool repairs the data structure

Table 4.1: The injected faults and ANTLR `addChild()` fault. The last column shows if the field(s) that should be corrected appear in the write barrier log (WB), read barrier log (RB), or all fields excluding the write and read barrier logs (ALL fields).

Method	Fault description	Violates	Error in
Err 1	Sets the header to null	Correct remove, Size	WB
Err 2	Fails to update the size	Size	ALL fields
Err 3	Deletes a node with a non-matching element	Correct remove, Size	WB
Err 4	Introduces a cycle after performing correct remove	Acyclicity	WB
Err 5	Breaks the list to retain only the first two nodes	Correct remove, Size	WB
Err 6	Deletes the matching element but adds it again	Correct remove	WB
Err 7	Fails to remove the element and updates the size incorrectly	Correct remove, Size	WB
Err 1	Creates a cycle of length one	Acyclicity	WB
Err 2	Sets the color of a node to black instead of red	Color constraints	WB
Err 3	Adds the new element as right child instead of left	Key constraints	RB
Err 4	Violates key constraints due to a branch condition error	Key constraints	RB
Err 5	Same as Err 3 with a different input	Key constraints	WB
Err 6	Same as Err 4 with a different input	Key constraints	WB
Err 7	Skips balancing of the tree after insertion	Color constraints	ALL fields
Err 1	Skips deletion of the appropriate child	Correct Remove	RB
Err 2	Skips updating children indices after deletion	Child Index constraints	ALL fields
Err 3	Sets a wrong child index due to an incorrect branch condition in a loop	Child Index constraints	RB
Err 4	Sets a node as its own parent	Acyclicity	WB
ANTLR <code>addChild</code>	Adds a node to itself as a child	Acyclicity, Child Index	WB

by assigning the correct value to the header. Whereas, in the second faulty method, the statement which updates the size field is completely omitted hence the field does not get logged in write or read barrier logs. The tool thus needs to relax all fields in the UNSAT core to produce a correct data structure.

The program logic and thus which fields Cobbler logs depends on the input structures. Faults five and six of the red-black tree `insert` method execute the same faulty code versions as that of three and four, but with a different data structure. The program writes and reads different fields on the first and second inputs and Cobbler repairs the outputs by relaxing read and written fields respectively.

4.5.4 Subject Tools

We compare Cobbler to Juzi repair framework, which only uses structural constraints, and to Tarmeem and PBnJ, which consider post-conditions too.

Recall that Juzi’s assertion-based repair automatically corrects data structure violations in Java programs [33, 32]. Upon detecting a constraint violation, Juzi searches for a repair solution based on the data structure traversal encoded in `repOK` [15]. Juzi further boosts its performance with symbolic execution. Since Juzi does not use a SAT solver, it is generally faster than SAT-based approaches. Juzi however does not consider method post-conditions, which causes it to miss errors that result in well-formed output. Even if `repOK` is violated, without the post-condition, Juzi cannot accurately

correct the structure with respect to the contracts. To compare Juzi and Cobbler, we manually implemented a check for the post-condition in Juzi by recording the method pre-state and the desired data structure specific post-state. This implementation is a cumbersome and unusual way of checking the post-condition, but it did force Juzi to return an output that satisfies both `repOK` and post-conditions.

Our previous work, Tarmeem (Section 3.3), uses Alloy contracts and a SAT solver [104]. Tarmeem repairs faulty post-states using heuristics and user-guided techniques. Tarmeem’s heuristics limit the relaxations based on post-conditions when calling the SAT solver. User guides further focus the repair actions. We experimented with all four of Tarmeem’s heuristics and picked the best.

PBnJ executes method specifications when methods fail to produce a correct data structure [82]. Users express invariants and specifications in a declarative first order relational logic. Translating them into Java methods and then invoking the methods implements program logic declaratively. This program synthesis approach leverages constraint solving technology.

4.5.5 Results

Figure 4.3 compares the performance and accuracy of repair of Cobbler, Tarmeem, Juzi, and PBnJ on the singly linked list microbenchmark. Logging, check, and repair times are accumulated into a single bar on a logarithmic scale. Logging time is only applicable to Cobbler and is negligible. Tarmeem

and Cobbler have the same check time since they both use Kodkod evaluation (not SAT solving) to perform checks after methods execute. Juzi executes `repOK` and PBnJ translates specifications to Java assertions, which more efficiently check the data structure. Cobbler’s overhead on an error-free execution includes both logging and check times. Translating specifications to Java assertions (Section 5.3.1) could reduce the check time and the total overhead. We timeout after 60 seconds and report zero for accuracy upon a timeout.

Cobbler is substantially faster than all the other tools on five of the seven errors, despite the fact that Tarmeem and PBnJ receive specific user annotations to guide the repair process and Juzi performs symbolic execution. Error two skips a required update to size. Since the size field is not read or written, Cobbler does not correct it until the third call to the SAT solver, which causes its time to exceed the other repair schemes. Error four introduces a cycle. Juzi is tailored for such errors: it satisfies the constraint by breaking cycles quickly and performs better than Cobbler in this case.

Cobbler, except for one case, always produces the most accurate data structure among the four. When Cobbler does not time out, it achieves exactly the same output as expected. The edit distance between the result of a correct implementation and the repaired data structure is zero. This comparison is solely for evaluation, since in the wild, the system would not know the correct implementation.

Because original Juzi solely relies on the `repOK` method instead of checking method post-conditions, it does not find error six at all. Moreover, Juzi

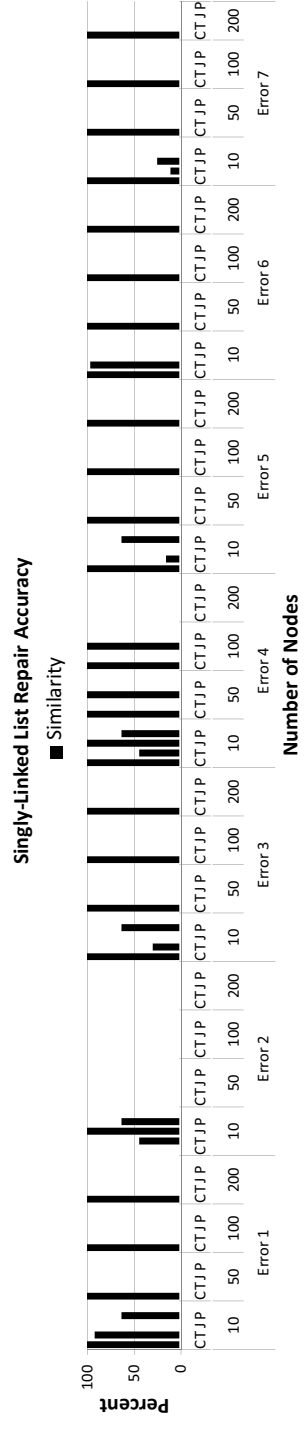
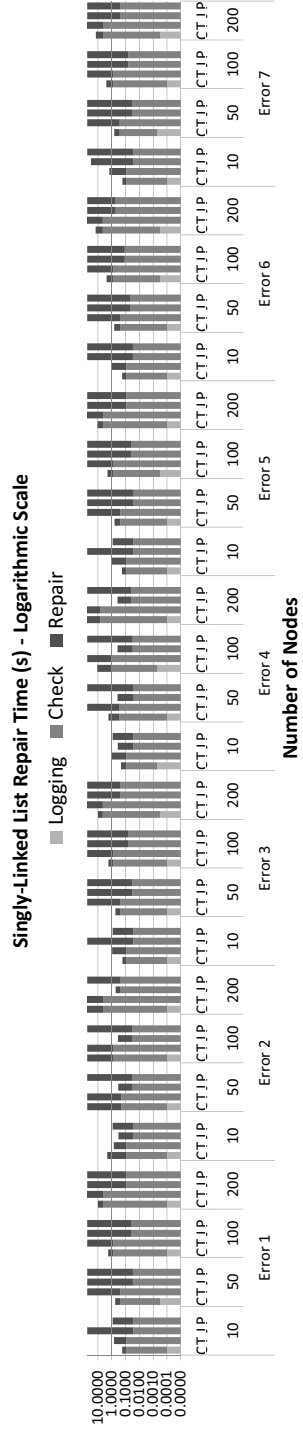


Figure 4.3: Performance and accuracy: repairing singly linked lists with Cobbler (C), Tarmeem (T), an enhanced version of Juzi (J), and PBnJ (P).

cannot access out of scope nodes that are not reachable from the header. Since Juzi does not consider the execution history, it first explores all the correct data structures nearby, but there is no guarantee that the expected output is close to the faulty one. We could enhance Juzi to work with post-conditions, as we did for evaluation of accuracy, but the original Juzi did not perform any repairs with respect to post-conditions.

Tarmeem is not very accurate because when it invokes the SAT solver, it relaxes *all* tuples of a relation together, causing unnecessary changes. Cobbler significantly improves efficiency and accuracy over Tarmeem, especially for errors which involve incorrectly updated fields. For example, Tarmeem takes around 10 seconds at best to repair a faulty linked list of only 20 nodes with error three. Cobbler relaxes based on the write log, which is accurate for this error, and thus repairs the structure within 0.8 seconds (13x faster than Tarmeem). Similarly, Cobbler reduces the repair time for error five, which breaks the list structure, by a factor of 8.6x compared to Tarmeem.

PBnJ's performance is similar to Tarmeem at best. The reason is that it always ignores the current faulty state and utilizes SAT to regenerate an acceptable output from scratch. It is however more accurate than Tarmeem in some cases.

Figure 4.4 shows the performance and accuracy of Cobbler on a faulty Kodkod red-black tree insert method. Figure 4.5 depicts the results of experimenting with ANTLR. When it does not timeout, Cobbler is very accurate on these real world applications.

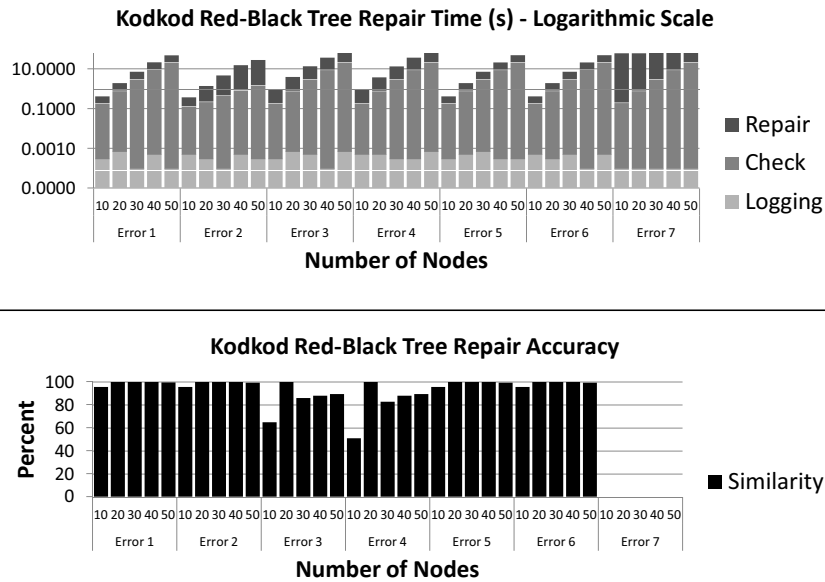


Figure 4.4: Cobbler performance and accuracy: repairing Kodkod red-black trees.

The results show that the read and write field logs improve the scalability and efficiency of repair. Cobbler repairs linked lists with up to 200 nodes within 20 seconds. It performs well even on more complex data structures. For the red-black tree `remove` method, it repairs up to 50 nodes within 40 seconds and for the `deleteChild` method of ANTLR `BaseTree`, it repairs 40 nodes within 30 seconds. The size of the logs is proportional to the number of reads and writes to the data structure and was usually a few hundred bytes with a maximum of 900 bytes for error four of ANTLR.

For errors that cannot be fixed by relaxing only written and read fields, such as error two of linked list, error seven of red-black tree insert, and error

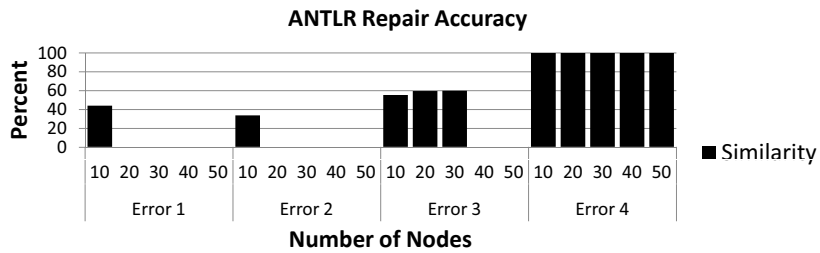
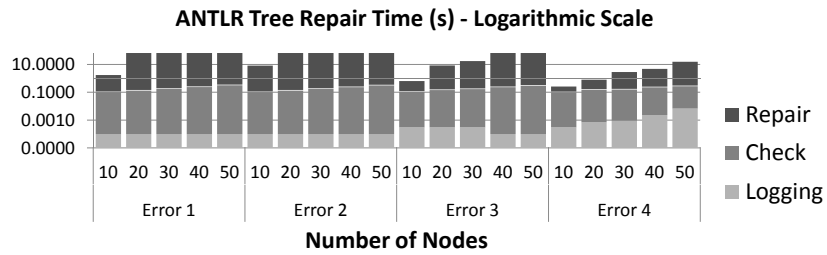


Figure 4.5: Cobbler performance and accuracy: repairing ANTLR trees.

two of ANTLR `deleteChild` (see Table 4.1), Cobbler uses the UNSAT core to identify which fields to modify, and performs better than the other SAT-based tools. These cases however are challenging for Cobbler, because despite barrier logs that indicate fields of specific objects, UNSAT cores identify all fields with the same name as potentially faulty.

4.5.6 ANTLR `BaseTree addChild`

The public method `addChild` adds child node trees to an ANTLR `BaseTree` object. When the input tree does not have a payload (`isNil`), the method adds the children of the input tree to the children list of the current tree, otherwise, it adds the input tree itself to the children list. In the

`addChild` method (version 3.3), when the input tree does not have any payload, a check ensures that the a tree is not being added to itself. However, such a check is not performed for input trees with payloads. Hence, when the current tree has a payload, it may be added as a child of itself. Similarly, any tree with a payload which is already an existing child of the current tree may be added as a child again. We generated inputs that caused invariant (such as acyclicity and ascending child indices) violations. Cobbler repairs the Tree structure and restores it back to its pre-state, which is correct. This state would be the output of `addChild` if it had been implemented correctly. Cobbler repairs a tree with 300 nodes within 30 seconds.

We contacted the ANTLR development team and they confirmed that `BaseTree` assumes acyclicity but does not check for it. Yet, since `addChild` is a public method, it should perform the check.

4.6 Summary

This chapter covered the idea of using program execution history for efficient and accurate contract-based data structure repair. We utilize program traces, specifically reads and writes of key fields, to direct repair of erroneous program states. Moreover, we use unsatisfiable cores provided by SAT solvers when we cannot repair the data structure by changing only read and written fields. We implemented this approach in Cobbler. Compared with previous repair techniques, our experimental results show Cobbler provides significant speedups and better accuracy, and finds and repairs a previously undetected

bug in the widely used open-source ANTLR program.

Chapter 5

Repair Abstractions

Tarmeem and Cobbler improve data structure repair and pave the way for it to be used in real applications. However, a more efficient, scalable, and accurate repair is needed to fulfill the promise of repair for real world applications. Recent work by Malik introduced the idea of repair abstractions [65], a new approach for scalability by memoizing and reusing repair actions to more efficiently repair errors that recur. This chapter develops the idea in the context of contract-driven data structure repair using the Alloy tool-set and is based on our joint publication at RV 2013 [107].

The key issue with current repair frameworks is that finding a sequence of repair actions, which produces a desired state, necessitates transmuting the specification into a partial implementation, say using a backtracking search over a large state space—an inherently complex operation.

Most data structure repair frameworks [32, 55, 50, 82, 104, 103] instantiate a search in the space of valid data structures to find a close and correct data structure to replace the faulty one. This search poses a major challenge to the scalability of data structure repair, as the size of the state space increases typically exponentially with the size of the data structure.

Recall the running example of binary search tree. The result of the faulty remove method with bug `cycle` when applied on input Figure 5.1 (a) to remove element 5 is shown in Figure 5.1 (b), and a possible repair result is Figure 5.1 (c). Now consider the result of the same faulty implementation when applied on Figure 5.2 (a) to remove element 7 depicted in Figure 5.2 (b). The search space grows with the size of the data structure, and so does the repair time to repair the error caused by bug `cycle`. Yet, the conceptual action required to break the cycle and remove the element is the same. Indeed, it is enough to undo and fix the effect of the wrong pointer manipulation of bug `cycle` in the code.

5.1 Repair Abstractions with Alloy Back-End

Our key insight, introduced in a co-authored work [103], is to abstract out repair actions and use them as possible repair action candidates in the future, before opting into searching the state space. The idea is that if an error in the data structure is due to a fault in software or hardware, a similar error may occur again, for example when the same buggy code segment is executed again or when the same faulty memory location is accessed again. Repair abstractions capture the essence of how certain data structure corruptions are repaired by specific actions of a data structure repair routine, such as Cobler [103], Juzi [32], PBnJ [82] or any other repair framework. Conceptually, a repair abstraction is a tuple $(condition; action)$ where action is an abstract repair action performed when condition is met on a program state that needs

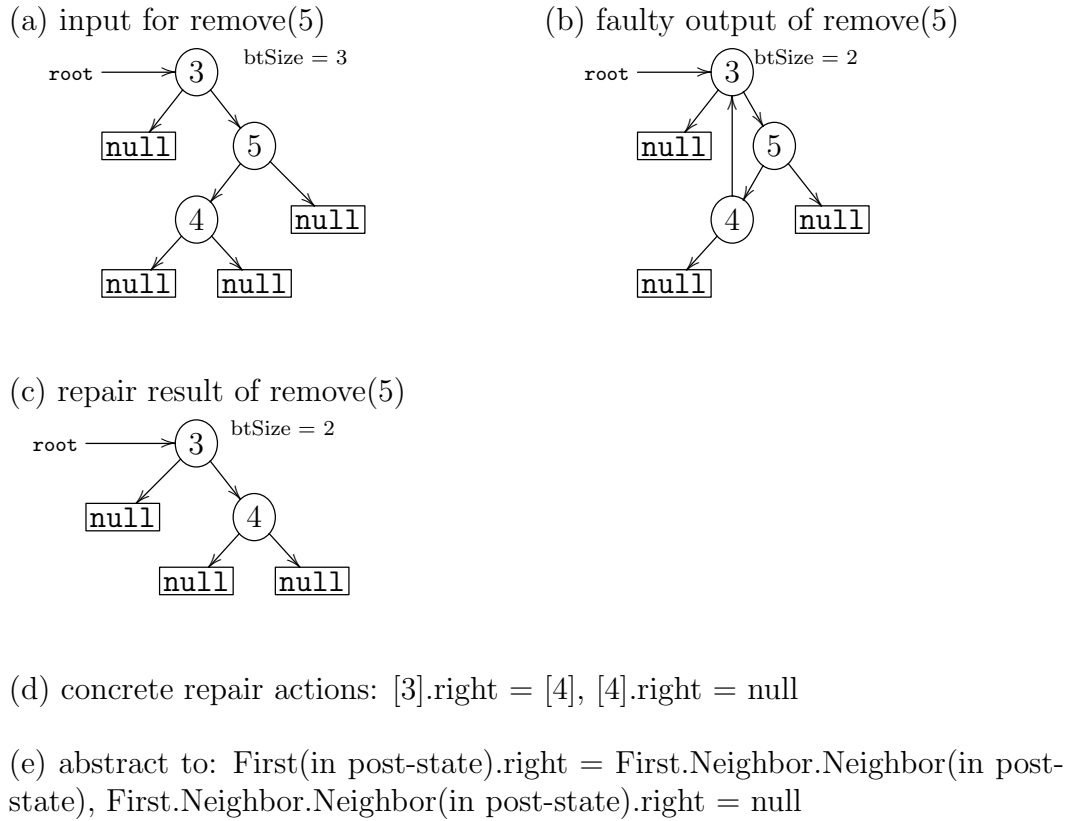
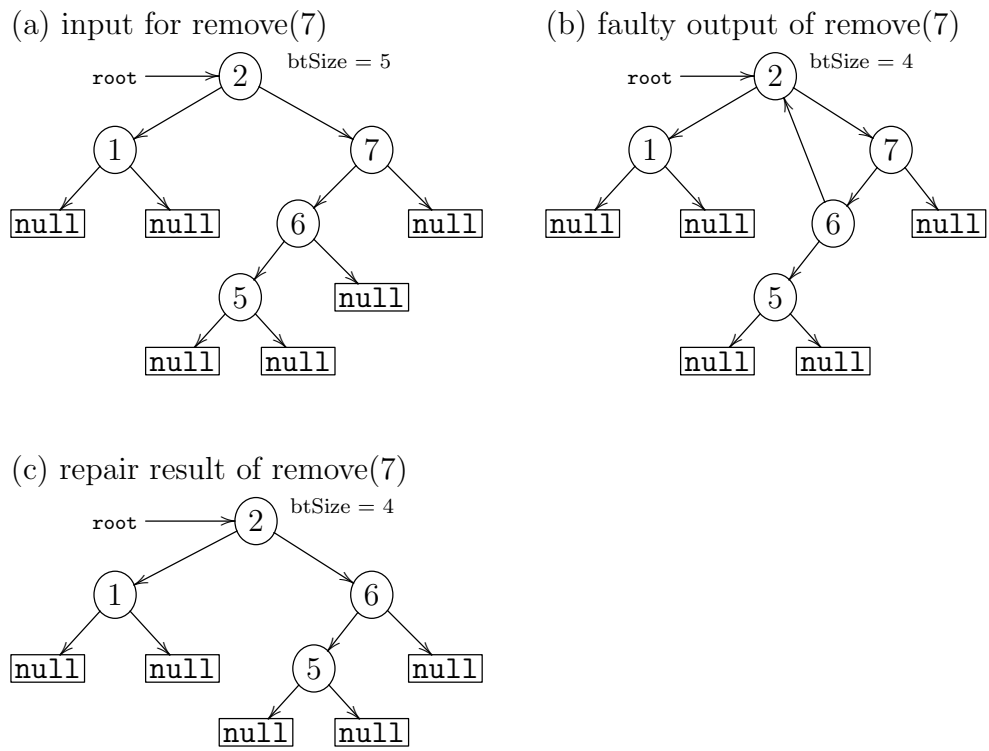


Figure 5.1: Concrete and abstract repair actions to repair the result of bug cycle on a tree of three nodes.

repair.

Consider the example of repairing the faulty output of `remove(5)` shown in Figure 5.1 (b). The concrete repair actions suggested by the underlying repair framework include the assignments $[3].\text{right} = [4]$ and $[4].\text{right} = \text{null}$ ¹ shown in Figure 5.1 (d). We abstract out these concrete repair actions to the abstract actions shown in Figure 5.1 (e). Suppose that a similar

¹[x] represents the node with value x before execution.



(d) reuse abstract repair actions: $\text{First}(\text{in post-state}).\text{right} = \text{First.Neighbor.Neighbor}(\text{in post-state})$, $\text{First.Neighbor.Neighbor}(\text{in post-state}).\text{right} = \text{null}$

(e) concretize to: $[2].\text{right} = [6]$ $[6].\text{right} = \text{null}$

Figure 5.2: Abstract and concrete repair actions to repair the result of bug cycle on a tree of five nodes.

error occurs again, now on a tree of five nodes with a different input to the remove method as shown in Figure 5.2. Before starting to search the state space of correct data structures, we first try the previous abstract repair actions in the hope of finding a quick fix. We reuse the abstract repair actions (Figure 5.2 (d)). Concretizing the abstract repair actions on the current data

structures gives Figure 5.2 (e) which is a correct repair.

Repair abstractions offer two key advantages. One, they allow summarizing concrete repair actions into intuitive descriptions of how certain errors in data structures were repaired, which helps programmers understand and debug faulty program behaviors (when the errors in the state were due to bugs in code). Two, they allow a direct reuse of repair actions without the need for a systematic exploration of a large number of data structures when the same error appears in a future program execution. The cost of repair, in cases that we do perform a search, will now be amortized over many repairs.

5.2 DREAM Framework

We implemented the idea of abstracting and reusing repair actions in a tool called DREAM (Data structure Repair using Efficient Abstraction Methods). In this section, we explain the fundamentals of DREAM. DREAM sits on top of an external data structure repair framework (Figure 5.3). Recall that when a data structure repair framework is in place, specifications are periodically checked to make sure that data structure invariants and/or method post-conditions hold. Once a check fails, the repair routine is triggered. Our repair algorithm (shown as a Java like pseudo code in Listing 5.1) has three major phases:

1. DREAM tries previously abstracted repair actions to see if it can find a repair without calling the repair routine of the underlying repair frame-

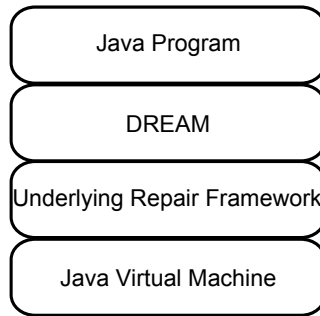


Figure 5.3: The relationship between DREAM, the underlying repair framework, the Java Virtual Machine, and the program.

work.

2. If the previous phase does not generate an acceptable repair, DREAM calls the repair routine of the underlying repair framework.
3. DREAM abstracts the concrete repair actions taken by the underlying repair framework and saves them as possible repair candidates for future.

To illustrate, consider repairing the result of bug `cycle` in the faulty output of Figure 5.1 (b). The first time this bug causes a failure, no previous repair abstraction is available (in Listing 5.1, `abstractRepairCandidateSets` is empty). Therefore, the first phase (Lines 3 to 19 of Listing 5.1) is skipped. Line 20 performs the second phase and calls the underlying repair framework, which repairs the data structure by setting `[3].right = [4]` and `[4].right = null`. These concrete actions are abstracted in the third phase by Lines 21 to 23 to be saved as `First(in post-state).right = First.Neighbor.Neighbor(in post-state)` and `First.Neighbor.Neighbor(in post-state).right = null`

```

1 Object dreamRepair(Object input , Object faultyOutput){
2   Object repairedOutput;
3   for(Set<AbstractRA> abstractCand : abstractRepairCandidateSets){
4     Set<ConcreteRA> concreteCand = new HashSet<ConcreteRA>();
5     for(AbstractRA action : abstractCand){
6       List<Field.ConcreteField> left = new LinkedList<Field.ConcreteField
7         >();
8       Object baseObject = action.derefLeftInOut ? faultyOutput : input;
9       for(Field.AbstractField f : action.derefLeft)
10        left.addAll(concretizeOnObject(f, baseObject));
11      Object leftHandSide = getObject(left, baseObject);
12      ...//Similarly for the right hand side and field
13      concreteCand.add(new ConcreteRA(leftHandSide, concreteField,
14        rightHandSide));
15    }
16    repairedOutput = apply(faultyOutput, concreteCand);
17    if(check(input, repairedOutput)){
18      increaseScore(abstractCand);
19      return repairedOutput;
20    }
21  }
22  repairedOutput = repair(input, faultyOutput);
23  Set<ConcreteRA> newConcreteCand = getConcreteRA(faultyOutput,
24    repairedOutput);
25  Set<AbstractRA> newAbstractCand = abstractOut(newConcreteCand, input,
26    faultyOutput);
27  abstractRepairCandidateSets.add(newAbstractCand);
28 }

```

Listing 5.1: DREAM main algorithm.

respectively. More details about the abstraction process follows in Section 5.2.1.

The next time an error is observed in the data structure, DREAM attempts to reuse previous repair actions to avoid the prohibitively costly repair routine of the underlying repair framework. Let us say that we have Figure 5.2 (b) as the output of faulty `remove(7)` with bug `cycle` this time. Lines 3 to 19 implement the first phase of DREAM. They examine candidate sets of abstract repair actions.

Firstly, on Lines 5 to 13, DREAM concretizes each abstract action on the current data structure. An abstract action (like `First.Neighbor.Neighbor(in post-state).right = null`) contains a left hand side dereferencing list (here `First.Neighbor.Neighbor(in post-state)`), a field on which the assignment should be applied (here `right`), and a right hand side dereferencing list (here `null`). Such dereferencing lists are abstracted forms of actual dereferencing lists that were used in concrete repair actions and may contain abstract fields (e.g., `First` and `Neighbor`) as well as concrete fields (e.g., `null` and `left`). In the concretization process (Section 5.2.1), abstract fields are translated back into sequences of concrete data structure fields (Lines 8 and 9 translate abstract fields to concrete fields to build a concrete dereferencing list). The concretized lists are applied on the input or the faulty output to identify the target object on which the assignment should take place (Line 10 finds the left hand side object), the target field, and the target value (similarly for the field and right hand side). Line 12 accumulates concrete repair actions for application. DREAM can utilize either the input or the faulty out-

put for concretizing abstract actions and identifying target objects by using `baseObject` (Line 7).

Secondly, on Line 14, the set of concrete actions is applied on the faulty output. It is noteworthy that abstraction and concretization is performed on a snapshot of the input or the faulty output, and the application of repair actions applied at the same time does not affect the meaning of one another.

Finally, Line 15 checks if the result is indeed a correct output with respect to the specification. If so, DREAM ascends the abstract set that created this repair in the ordered list of candidates `abstractRepairCandidateSets` (Line 16) and returns the repaired output without continuing to phases two and three (Line 17). If the problem was not fixed, the repair action is rolled back to the faulty post-state and the algorithm continues with other abstractions followed by phases two and three.

5.2.1 Abstraction and Concretization

This section elaborates on abstraction (Line 22 of Listing 5.1) and concretization (Line 9). DREAM uses a pre-defined yet generic and extensible repository of meaningful abstractions. We define the following abstractions as the basis of our approach:

Examples of pointer-based abstractions:

- *null*: the `null` value;
- *First*: the first object of a type reachable from the given root pointer

(e.g., the root of a tree or the first node in a list);

- *Last* or *Leaf*: the furthest object(s) of a type reachable from the given root pointer (e.g., leaves of a tree or the last node in a list);
- *Self*: the object itself;
- *Neighbor*: a neighboring object, where two object O_1 and O_2 are neighbors if a field of O_1 points to O_2 or vice versa (e.g., the parent of a node in a tree);

Examples of value-based abstractions:

- *A value with an offset*: the numeric value of a node plus/minus an offset (e.g., the size of a binary heap plus one);
- *A value with a coefficient*: the numeric value of a node multiplied/divided by a coefficient (e.g., twice the value of a key in a Red Black Tree);

Pointer-based abstractions view the program heap as a directed, edge-labeled graph. They may be defined with respect to the entire structure (e.g., null or First) or with respect to a particular node (e.g., Self or Neighbor). Value-based abstractions are used to abstract repair actions on integer fields, e.g., size.

The abstraction process has two steps:

1. A breath first search of the data structure (for both the input and faulty output) is performed along all concrete fields. This search assigns a concrete dereferencing list that can be used to reach any object. For example, [4] in Figure 5.1 (a) and (b) is reached via `root.right.left`.
2. Using the above repository of abstractions, all possible abstractions that are equal to a concrete dereferencing list are built. E.g., `root.right.left` is considered equal to `First.Neighbor.Neighbor(in post-state)`, or `Leaf(in pre-state)`. The abstractions `Self`, `null`, `Offset`, and `Coefficient` are only useful as the right hand side of a repair action.

The concretization process is the exact reverse of abstraction. First, DREAM transforms the abstract fields of a dereferencing list to their concrete forms which only use the data structure fields. E.g., `First.Neighbor.Neighbor(in post-state)` could be `root.right.right(in post-state)`, `root.right.left(in post-state)`, etc. Then, it traverses the data structure along those fields to get to the desired objects. When multiple abstractions/concretizations are applicable, all of them are used in turn as possible candidates and checked, until one is found to work or all are exhausted.

Both abstraction and concretization can be performed on the input data structure as well as the faulty output of a method. This flexibility enhances DREAM's ability to access objects that get lost because of broken pointers. `derefLeftInOut` and similar boolean flags are put in place to distinguish between cases that the faulty output and the input are used to access an

object.

5.3 DREAM with Alloy Back-End

Connecting DREAM with an Alloy-based repair framework (like PBnJ [82], Cobbler [103] or Tarmeem [105, 104]) is quite straightforward. The underlying repair framework performs regular checks and provides concrete repair actions in case abstractions do not work.

5.3.1 Arreh

Our repair techniques concentrate on the efficiency and scalability of *repair*. A repair framework, however, constantly checks contracts. Such *checking* using SAT is rather time consuming. To speed up the checking of contracts, we built the Arreh² tool for checking specifications based on the Minshar [8] technique. This technique translates Alloy checks to Java runtime checks. Using Java checks instead of Alloy improves the performance and scalability of checking. The original Minshar tool only supported data structure invariants and not pre- and post-conditions. We enhanced Minshar to include the support for pre- and post-conditions as well.

Arreh is an extension to Alloy Analyzer. It receives a model in Alloy, parses the model using the Alloy Analyzer parser, and translates its contract

²A sequence of research tools (TestEra, Minshar, etc.) that provide automated testing and checking have been named *saw*, the tool for cutting wood, in the native languages of their authors. Arreh means saw in Persian (Farsi).

checks (i.e., commands) to Java run time checks (i.e., Boolean methods). In order to translate Alloy commands, Arreh starts by parsing the Alloy specification into Alloy Abstract Syntax Tree (AST), which indicates how Alloy expressions are recursively built from subexpressions. Arreh then traverses this AST, and recursively replaces each operation with a Java method call.

Figure 5.4 shows a snapshot of the Arreh tool, which extends Alloy Analyzer. For each Alloy command, Arreh adds *Translate to Java* to the *Execute* menu, through which the user can translate the command to Java. Arreh then shows the Java check, which is a stand alone Java program, in a separate window.

Experimental evaluation (Section 5.4) shows that Arreh significantly reduces the burden of constantly checking contracts. While Arreh and the SAT-based back-end of Alloy Analyzer take the same time to translate to Java and SAT respectively, executing Java checks is orders of magnitude faster than checking through the SAT solver. Better still, translation to Java is a one time operation for Arreh and the same Java code can be reused, while reusing Alloy to SAT translation is not available at this time.

5.4 Evaluation of DREAM with Alloy Back-End

We present the experimental evaluation of DREAM combined with Cobble. 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. Cobble used MiniSat and MiniSatProver SAT solvers.

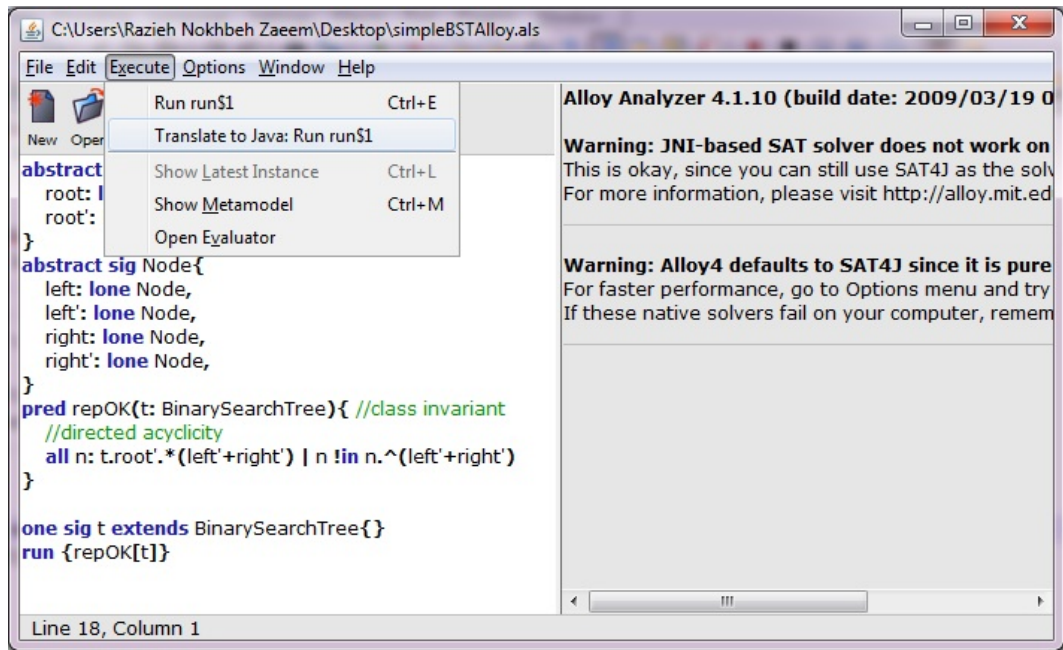


Figure 5.4: A snapshot of Arreh.

Our evaluation of DREAM with Cobbler considers the scenario when repair is initially performed on an erroneous structure of a specific size and then re-applied on another erroneous structure that has a different size but a similar fault as before. Both Cobbler and DREAM use Alloy specifications here. However, in Cobbler the specifications are checked at runtime using SAT, whereas in DREAM the specifications are checked using the JVM since the specifications are translated to Java assertions using Arreh.

The first data structure we look at is a basic Singly Linked List that also keeps its size. We use the same errors we used to evaluate Cobbler in Section 4.5.3. In that section, we included seven erroneous remove methods for Singly Linked List. We used the same seven errors plus two new ones here.

Table 5.1: Description of the Singly Linked List errors used for experimental evaluation of DREAM.

	Err. #	Description
Singly Linked List	1	Sets the header to null
	2	Fails to update the size
	3	Deletes a node with a non-matching element
	4	Introduces a cycle after performing correct remove
	5	Breaks the list to retain only the first two nodes
	6	Deletes the matching element but adds it again
	7	Fails to remove the element and updates the size incorrectly
	8	Fails to remove the element
	9	Fails to update the size because of a missing left hand side in an assignment

Table 5.1 shows the errors and a brief description of each of them. Some of the errors violate the invariants of Singly Linked List (e.g., Error 4), some violate the post-condition of the remove method (e.g., Error 1), and some violate both (e.g., Error 7).

We start by repairing a Singly Linked List of 10 nodes. Upon the very first error, no repair abstraction is available, so DREAM has to use the underlying repair routine which is Cobbler here. Then DREAM abstracts out the set of concrete repair actions taken by Cobbler and memorizes them for future use. In the next experiment, we used a Singly Linked List of 500 nodes with each error. DREAM applies the abstract repair actions which fix the problem without calling Cobbler in 8 out of 9 errors. Table 5.2 shows the abstractions that DREAM extracted for each error. Some abstractions, e.g., the first and second abstraction for Error 9, are unnecessary but harmless since

Table 5.2: Abstract repair actions suggested by DREAM for Singly Linked List.

Err. #	Abstract Repair Action(s)
1	list (in post-state).header = First (in pre-state)
2	DREAM Not Working: Call SAT
3	First (in post-state).next = header.next.next (in pre-state)
4	Last (in post-state).next = null
5	Last (in post-state).next = header.next.next.next (in pre-state)
6	First (in post-state).elt = header.elt (in pre-state)
7	First (in post-state).elt = header.elt (in pre-state)
	list (in post-state).size = size - 1 (in post-state)
8	First (in post-state).next = null
	list (in post-state).header = header.next (in post-state)
	header.next (in post-state).elt = header.elt (in post-state)
	First (in post-state).elt = null
9	header.next (in pre-state).next = null
	header.next (in pre-state).elt = null
	list (in post-state).size = size - 1 (in post-state)

Singly Linked List

they change now unreachable nodes. These unnecessary actions exist because SAT suggested them as concrete repair actions.

Table 5.3 displays the time performance of Cobbler repairing lists of size 10 and 500, as well as DREAM repairing the same lists. For the case of calling Cobbler, an initial call is made to SAT to discover the problem and trigger repair (the check column in Table 5.3). Hence, the total time for repair with Cobbler includes the initial check time plus the repair time. For DREAM, first the repair actions are abstracted (column Abs.) using concrete repair actions taken by Cobbler on the data structure of 10 nodes. Then, using the data structure of 500 nodes, a Java check is performed to find out

Table 5.3: Time taken to repair erroneous Singly Linked Lists (ms).

Error #	Cobbler/DREAM (Size = 10)			Cobbler (Size = 500)			DREAM (Size = 500)			Improvement			
	Check	Repair	Total	Check	Repair	Total	Abs.	Check	Con. Repair		App. Check	Total	
1	320	799	1119	287034	125638	412672	24	0	0	1	77	78	5291x
2	915	8846	9761		Out of Heap			Not Working:	Call SAT				Both fail
3	166	417	583	127434	240674	368108	14	38	0	39	37	114	3229x
4	128	381	509		Out of Heap		6	0	0	1	41	42	Cobbler fails
5	130	292	422	52621	61751	114372	6	0	0	6	40	46	2486x
6	145	410	555	55691	142061	197752	7	42	0	44	32	118	1676x
7	126	319	445	52356	133512	185868	6	19	0	21	32	72	2582x
8	131	259	390	51766	234913	286679	6	16	1	17	32	66	4344x
9	228	797	1025	92219	298215	390434	8	64	1	69	69	203	1923x

that the specification is violated. This Java check is a manual translation of the specification from Alloy to Java using the Arreh technique which can be automated using the Arreh tool. When this initial check fails, DREAM repair performs concretization (the Con. column) followed by the application of concretized actions (the App. column). Unlike Cobble which only suggests correct repairs, the result of applying a set of abstract repair actions by DREAM should be checked to see if the abstractions can indeed resolve the problem. Therefore, there is another Java-based check after DREAM repair. Note that abstracting repair actions is a one time procedure whose results are reused multiple times, therefore it is not included in the total time for repairing with DREAM.

DREAM abstractions do not work for Error 2, mainly because the repair suggested by Cobble is too tailored to the specific data structure of 10 nodes and cannot be generalized. However, Cobble cannot repair a data structure of 500 nodes for Error 2 either because it runs out of the heap space. Cobble also fails to repair Error 4 on 500 nodes while DREAM repairs this error in a total of 42 ms. As Table 5.3 shows DREAM (utilizing Arreh) is substantially (about 3000 times on average) faster than Cobble and it repairs 8 out of 9 errors in less than a quarter of a second. The improvement is in part due to Arreh, because it makes checking much faster. However, even if Cobble uses Arreh for checking, it would still not be as efficient as DREAM, since Cobble has to use SAT for repair without the reuse of previous repair actions. Finally, note that DREAM repair is as accurate as the underlying

Table 5.4: Description of the Red Black Tree errors used for experimental evaluation.

	Err. #	Description
Red Black Tree	1	Creates a cycle of length one
	2	Sets the color of a node to black instead of red
	3	Adds the new element as right child instead of left
	4	Violates key constraints due to a branch condition error

repair framework: In cases that Cobbler does not fail, DREAM and Cobbler generate the exact same repair.

The second data structure we consider is the Red Black Tree implementation in the open source Kodkod model finder [93] as explained in Section 4.5.2. Table 5.4 repeats a brief description of the first four errors which we use to evaluate DREAM.

Similar to the Singly Linked List experiment, we repaired Red Black Trees of 10 and 500 nodes. Table 5.5 shows the abstract repair actions suggested by DREAM.

Table 5.6 shows the performance measurements. For a Red Black Tree of 500 nodes, Cobbler always times out where the timeout value is 500,000 ms. DREAM repairs all the errors in less than one minute. Note that the time Arreh takes for final checks is high since it is a direct, unoptimized translation of Alloy to Java. Since the structures are valid after repair, final checks do not short-circuit and take much longer than initial failing checks.

Table 5.5: Abstract repair actions suggested by DREAM for Red Black Tree.

Err. #	Abstract Repair Action(s)
1	First (in post-state).parent = null
	First (in post-state).color = First.right.right.left.key (in post-state)
2	First.left.left (in post-state).color = First.color (in post-state)
	First (in post-state).color = First.right.right.left.color (in post-state)
3	First.right (in post-state).key = First.right.key (in pre-state)
4	First (in post-state).right = First.right (in pre-state)
	First (in post-state).color = First.right.right.right.color (in pre-state)

Table 5.6: Time taken to repair erroneous Red Black Trees (ms). Timeout represents a timeout of 500,000 ms.

Error #	Cobbler/DREAM (Size = 10)		Cobbler (Size = 500)		DREAM (Size = 500)				Improvement		
	Check	Repair	Check	Repair	Check	Con.	Repair	App.			
1	282	582	864	Timeout	7	11	0	14	54642	54667	Cobbler fails
2	249	521	770	Timeout	5	37	0	40	56042	56119	Cobbler fails
3	331	772	1103	Timeout	6	9	0	11	53954	53974	Cobbler fails
4	251	494	745	Timeout	6	1045	0	1048	53508	55601	Cobbler fails

5.5 Summary

In this chapter, we developed the idea of repair abstraction in the context of contract-driven data structure repair using the Alloy tool-set to allow memoization and reuse of repair actions for fixing errors that recur. Our tool embodiment DREAM piggybacks on other repair frameworks and records concrete repair actions they take to fix a particular erroneous state. The experimental evaluation of the use of DREAM in accordance with Cobble on basic and complex data structures show that DREAM offers significant performance improvement. We envision that repair abstractions can be a valuable addition to data structure repair frameworks. DREAM’s ability to integrate with different repair frameworks provides a promising step toward making repair scale to real applications.

Chapter 6

Data Structure Generation Using Dynamic Programming

In this chapter, we build on the observation that structure construction is a central problem in two research thrusts in software engineering: test input generation and data structure repair. The problem solving structures of constraint-based test generation and contract-driven data structure repair are similar. The former uses constraint solving to enumerate solutions that are refined as tests, and the latter uses constraint solving to generate a solution that repairs the erroneous state. This chapter presents a new technique for efficient structure generation and is based on our FSE 2012 paper [106]. While the focus of this chapter is on structure generation for testing, in Section 6.5 we discuss future ideas on how it enables better data structure repair.

Test input generation is one of the most challenging tasks in automated testing. Generation is especially hard for programs, such as browsers or compilers, which take complex structures, e.g., HTML or Java programs, as inputs, because such inputs are hard to generate automatically or sample at random. *Constraint-based testing* [24, 44, 49, 60, 80] provides the basis for effective techniques [26, 73, 72, 15, 67, 37, 38, 54] for generation of such structurally

complex inputs: an *input constraint* defines the structural properties of a class of desired inputs, a constraint solver enumerates solutions to the constraint, and the solutions are refined as test inputs. Advances in constraint solving technology [11, 28] have enabled solvers to handle more complex constraints. However, scaling constraint-based testing remains a challenging problem, particularly when used for systematic testing, i.e., *bounded exhaustive testing*, which tests against all inputs within a bound on the input size.

There are two fundamental questions any constraint-based test input generation technique addresses: (1) how to write constraints that define a desired class of inputs and (2) how to solve constraints. In this chapter, we introduce a novel constraint-based test input generation technique that addresses both these questions and enables efficient and scalable input generation of structurally complex inputs.

To write constraints, our technique supports *recursive* predicates that are written in the same language as the program under test. Our key insight is that not only are recursive predicates more natural to write for recursive structures (which may also have some non-recursive elements)—e.g., a binary tree is either `null`, or a node whose left and right children are each a binary tree—but also recursive predicates naturally lend themselves to recursive input generation. Intuitively, since the predicate definition is recursive, it suffices to build the non-recursive elements of a new input, and use the same method to recursively build the recursive elements. For example, to build a binary tree which is not `null`, it suffices to assign one node as the root and recursively

call the same procedure to build binary trees for the right and left children.

Our technique utilizes the recursive structure of desired inputs to divide the problem of generating an input into several sub-problems of generating smaller inputs that exhibit the same structure, and employs *dynamic programming* [25]—a well-known problem solving methodology designed to exploit common sub-problems—to combine them and generate inputs in a *bottom-up* manner. To illustrate, when exhaustively generating binary trees as inputs, we build all binary trees of sizes 0, 1, 2, and 3 and use them to construct binary trees of size 4. For random test generation, we randomly generate some binary trees of sizes 0, 1, 2, and 3, and then combine them to sample binary trees of size 4. Since many recursive test inputs exhibit the property of overlapping sub-problems, dynamic programming saves us a lot of computation.

We present three algorithms for input generation: (1) a basic algorithm that uses dynamic programming directly, (2) a lazy initialization strategy that optimizes dynamic programming, and (3) a further optimization using symbolic execution [58, 24]. Our first algorithm (*DP*) utilizes dynamic programming to generate test inputs up to a given size. To build a new test input, we take formerly generated test inputs and combine them to build the recursive parts, explore the state space to assign values to the non-recursive parts, and pass the resulting candidate object to the `repOK` predicate for evaluation. If `repOK` returns true, this object is saved as a valid input and is then used to create larger inputs.

Our second algorithm (*LazyDP*) uses *lazy initialization* to optimize performance. To cope with the limitation of other resources beyond computational time, such as memory, we store a recursive test input concisely as an array of numbers, where each number recursively represents a formerly generated test. When combining inputs to build a candidate, we do not expand all smaller test inputs in the hierarchy from their concise form to actual objects. Instead, we expand them lazily whenever repOK accesses them. For instance, consider the repOK method of a binary tree which indicates that both of the right and left children are valid binary trees, and the size of the new binary tree is one more than the sum of the sizes of its right and left children. This means that, when investigating a candidate test input, repOK will only access the size fields of the right and left children and not the size fields of all the other sub-trees. Therefore, we can limit the expansion of recursive objects to the right and left children and keep the smaller sub-trees in their concise form. Dynamic programming and lazy initialization together form our second algorithm, abbreviated as LazyDP.

Our third algorithm (*SymboLazyDP*) further enhances the performance by skipping a systematic search for non-recursive fields when possible. For recursive fields, this search is improved by using the first two algorithms where we restrict the choice of smaller sub-problems to formerly generated solutions. To avoid the search for non-recursive fields, the third algorithm adds symbolic execution [58, 56, 92], which uses symbolic values for non-recursive fields, builds a path condition while executing repOK, and uses an in-house constraint

solver to solve for the symbolic values. For example, symbolic execution solves for the size of a binary tree by using the size fields of its children, and hence avoids the invocation of repOK on various candidates with different values for the size. Furthermore, symbolic execution provides less, yet representative tests, and hence helps test execution, by representing each class of inputs with only one solution for a path condition, instead of exploring the entire state space.

Our technique not only enables efficient generation of inputs based on textbook data structures, but also a wide-range of string-based inputs that represent structured data, e.g., strings that belong to a context-free grammar, SQL queries, data in a database management system, Java programs, and HTML/CSS pages. We developed a prototype implementation of our three algorithms and evaluated it using microbenchmarks and real world applications, including Google Chrome and Apple Safari web browsers.

6.1 Example

In this section, we describe two examples: a binary tree, based on the example of Section 3.1, and an HTML input that we use alongside with a CSS input to test a web browser. We simplify the binary search tree of Section 3.1 by excluding the field `element`, for the sake of clarity in explaining the algorithms. The recursive definition of a binary tree is as follows: a binary tree is either `null`, or a node whose left and right children each point to a binary tree. The tree should not have any loops. Furthermore, in our binary

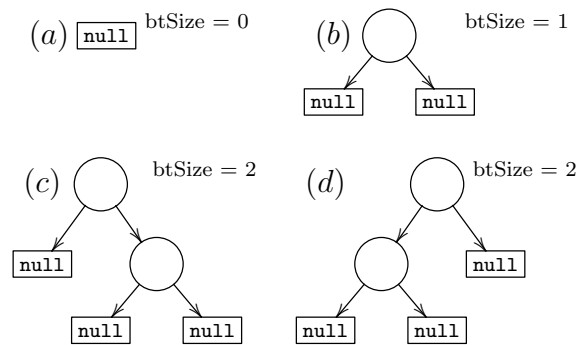


Figure 6.1: All binary trees up to size 2.

tree, `btSize` is equal to the number of nodes in the tree. Listing 6.1 shows the recursive implementation of a binary tree. One can easily see that a recursive `repOK` is a natural way of describing the properties of a recursive data structure. Figure 6.1 shows all binary trees up to size 2.

```

1 class BinaryTree {
2     BinaryTree right, left;
3     int btSize;
4     boolean repOK(){
5         if (!acyclic()) return false;
6         return recursive_repOK();
7     }
8     boolean recursive_repOK() {
9         int rightBtSize, leftBtSize;
10        if (right == null) rightBtSize = 0;
11        else {
12            if (!right.recursive_repOK())
13                return false;
14            rightBtSize = right.btSize;}
15        if (left == null) leftBtSize = 0;
16        else {
17            if (!left.recursive_repOK())
18                return false;
19            leftBtSize = left.btSize;}
20        if (btSize==rightBtSize+leftBtSize+1) return true;
21        else return false;}}

```

Listing 6.1: A recursive binary tree in Java.

Listing 6.2 shows the `repOK` method of an HTML input. The backbone of such an input is a generic tree, a recursive data structure. In addition, other

constraints are enforced on HTML tags. In Section 6.3.4, we demonstrate how to leverage this modeling of HTML files and list-modeling of CSS files to automatically generate bug revealing test inputs for the Chrome and Safari web browsers. Figure 6.2 models the nested structure of tags in Listing 6.5 (Section 6.3.4).

```

1 class HTML {
2   int tagIndex; HTML[] child;
3   Attribute[] attr;
4   String[] HTMLTags =
5   {"html", "head", "link", "body", ...};
6   boolean repOK() {
7     if (!acyclic()) return false;
8     if (tagIndex != 0) // "html"
9       return false;
10    if (!child[0].repOKHead())
11      return false;
12    if (!child[1].repOKBody())
13      return false;
14    ...
15    return true; }
16  boolean repOKBody() {
17    if (tagIndex != 3) // "body"
18      return false;
19    if (!child[0].recursive_repOK())
20      return false;
21    ...
22    return true; }}

```

Listing 6.2: HTML repOK method.

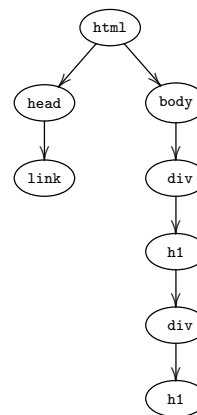


Figure 6.2: A tree representation of an HTML input.

Note that we support repOK methods with both recursive and non-recursive parts. The `btSize` of each subtree in Listing 6.1 and the `repOKHead` and `repOKBody` checks in Listing 6.2 are examples of non-recursive parts of a repOK.

6.2 Test Input Generation Framework

In this section, we describe how we use recursive definitions of data structures for exhaustive and random test generation. We explain our three algorithms and prove a theorem on their correctness.

6.2.1 Recursive repOK Methods

Many test inputs, such as data structures, have an embedded recursive structure. Sets, trees, stacks, queues, arrays, heaps, and many other data structures have recursive definitions¹. Using recursion in repOK to identify correct instances of recursive loop-free data structures makes repOK easier to write, read, and debug.

Besides identifying correct instances of a data structure, a repOK method should be able to identify incorrect structures as well, but incorrect structures are not necessarily loop-free. To comply with this standard definition of repOK, we require the template shown in the repOK method of Listing 6.1. This repOK first checks for cycles, and then enters the recursion phase to avoid an infinite loop. Throughout this chapter we use recursive repOK methods that assume acyclicity.

We process the source of a recursive repOK through a simple pattern

¹Nevertheless, recursive repOK methods are inherently unsuitable for describing data structures with loops. Therefore, a few data structures (e.g., circular linked lists) are not directly compatible with our test generation algorithm. Our framework can generate the recursive backbones of such data structures, and add loops later via a separate generation phase.

matching to find all recursive calls. A field on which repOK is recursively called is identified as a *recursive field*. We could also identify recursive calls by processing the Abstract Syntax Tree of method calls.

6.2.2 Algorithms

Here we describe DP, LazyDP, and SymboLazyDP. The use of these algorithms is orthogonal to exhaustive or random test generation. In Section 6.2.2.2 we describe a variation of the algorithms that generates random tests.

6.2.2.1 DP

Generation of test inputs can benefit from the recursive structure of repOK. Given a recursive repOK, the goal of exhaustive test generation is to generate all test inputs, in a given scope, for which repOK returns true. For example, the recursive repOK of a binary tree (Listing 6.1) checks whether its right and left children are correct binary trees and whether `btSize` is correctly set to the sum of the `btSize` fields of the children plus one. By observing the execution of repOK, we present our recursive method of generation: a new candidate test is generated by setting its recursive fields to formerly generated correct tests, and finding correct values for the non-recursive fields. Then, repOK is invoked on the candidates to filter out the incorrect ones. The repOK method directly rejects the candidates with loops and recursively calls itself to evaluate different parts of the loop-free candidates. This check in-

cludes assuring the correctness of recursive substructures. Because we provide previously generated valid test inputs as substructures, we can bypass these internal recursive calls and directly return true for them.

Besides breaking the problem into subproblems, our generation method demonstrates the other property necessary to a dynamic programming solution: overlapping subproblems. The same substructure is used multiple times to build new candidate test inputs. For instance, the binary tree of size 1 (Figure 6.1 (b)) is used twice in the generation of all binary trees of size 2 (Figure 6.1 (c,d)).

For more efficient test generation, the DP algorithm avoids generating repetitive candidates. To this end, we generate test inputs in iterations and keep three sets of previously generated inputs: `thisRoundTests` contains correct tests generated in the current iteration, `lastRoundTests` includes correct tests generated in the last iteration (which have not yet been used to build other candidates), and `pool` contains all other correct tests generated so far. At the beginning of test generation, `pool` and `thisRoundTests` are empty, and `lastRoundTests` contains only `null` (line 6 of Listing 6.3). We assume that `null` is always a valid test input because we cannot call `repOK` on `null` (e.g., it throws a `nullPointerException` in a Java program), and for all common data structures `null` is a valid instance.

The DP algorithm proceeds as follows (Listing 6.3): If `repOK` makes r recursive calls, we need r (not necessarily different, but ordered) tests to build a new test. The outer `while` loop (lines 10 to 31) executes as long as

```

1 void testGeneration () {
2     int r = getNumRecursiveChildren ();
3     int s = getNumNonRecursiveFields ();
4     pool = initialize ();
5     lastRoundTests = initialize ();
6     addTo(lastRoundTests, null, 0); //0 is the size of null
7     totalExplored++;
8     validCasesGenerated++;
9     boolean progress = true;
10    while(progress){//test generation round loop
11        progress = false;
12        thisRoundTests = initialize ();
13        int [] recursives = nextPermutation(r);
14        while(recursives != null){
15            if(size(recursives) >= maxNumRecursives){
16                recursives = nextPermutationPruning(r, size(recursives));
17                continue;}
18            int [] fields = nextValuation(s);
19            while(fields != null){
20                totalExplored++;
21                Object testcaseObj = buildCandidate(recursives, fields);
22                if(testcaseObj.repOK() && (!randomIsOn() || coinToss(size(
23                    recursives)))){
24                    progress = true;
25                    validCasesGenerated++;
26                    addTo(thisRoundTests, combine(recursives, fields), size(
27                        recursives));}
28                    fields = nextValuation(s);}
29                recursives = nextPermutation(r);}
30            for(int [] test : lastRoundTests)
31                addTo(pool, test, size(test));
32            lastRoundTests = thisRoundTests;
33        }//end test generation round loop

```

Listing 6.3: Test generation algorithm in Java.

it makes progress in generating new test inputs. At each iteration, the test inputs generated in the previous iterations are combined to form new candidates; then `repOK` is invoked to identify correct test inputs. More specifically, the `recursives` array selects a permutation of r recursive substructures by calling the `nextPermutation` method. This method iterates over `pool` and `lastRoundTests` and upon each invocation, provides the next permutation of r substructures from the set `lastRoundTests ∪ pool`, such that at least one of the r substructures is selected from `lastRoundTests`. (Section 6.2.3 shows how this constraint assures that we would avoid repetitive candidates.) When all permutations are exhausted, this method returns `null`.

A challenge for DP is the exponential growth of the number of candidates at the outer boundaries of the scope. In fact, even creating those candidates can considerably affect the performance. To address this challenge, we keep the tests in each of the three sets sorted according to their sizes. We use bucket sort [25] (since the maximum size of a valid test is known from the scope) as we add new tests. In Listing 6.3, once `recursives` is selected, we first examine its size in line 15. If a candidate built with this permutation would be outside the scope, we throw this permutation away and also prune all other permutations with the same or bigger sizes via calling the `nextPermutationPruning` method. This method gives the next permutation that is inside the scope. Once we have such a permutation, we use the r substructures to build the recursive fields of a new candidate. In order to find the proper values for the non-recursive fields, we perform a systematic search

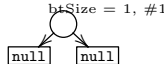
	size = 0	size = 1	size = 2	size = 3
<code>pool</code>				
<code>lastRoundTests</code>	<code>null</code> btSize = 0, #0			
<code>thisRoundTests</code>				

Figure 6.3: Finding binary trees up to size three (first iteration).

by calling the `nextValuation` method, which, upon each invocation, returns one valuation for the non-recursive fields. A permutation for `recursives` together with a valuation of `fields` gives us a candidate (`testcaseObj`) which we send to `repOK`. If `repOK` returns true (ignoring `randomIsOn` and `coinToss` for now), the new test gets added to `thisRoundTests` with its appropriate size, otherwise it is discarded. At the end of the outer `while` loop, when all permutations are exhausted, the tests in `lastRoundTests` join `pool` (lines 28 and 29), the tests in `thisRoundTests` replace `lastRoundTests` (line 30), and we move on to the next round.

In the DP algorithm, size is defined recursively as follows: If a candidate is `null`, its size is 0. Otherwise, the size of a candidate is the sum of the sizes of its recursive substructures plus one. Note that in the example of a binary tree, `btSize` has the same meaning of size. The size concept built into the DP algorithm, however, does not necessarily correspond to a field of the data structure. The DP algorithm memoizes and uses the size of a test, whereas it treats `btSize` as a non-recursive field, and performs a systematic search to find its correct value for any candidate.

Consider the example of finding binary trees up to size 3. (Figures 6.3,


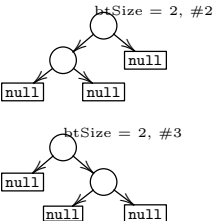
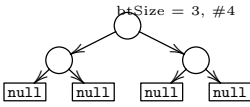
	size = 0	size = 1	size = 2	size = 3
pool	<code>null</code> btSize = 0, #0			
lastRoundTests				
thisRoundTests				

Figure 6.4: Finding binary trees up to size three (second iteration).

6.4, and 6.5 in which trees are numbered using # in the order they are generated.) In the first iteration, the invocation of `nextPermutation` returns different permutations of r (i.e., two) formerly generated binary trees such that at least one of them is from `lastRoundTests`. Here there is only one option: `null` for both right and left children. The total size of the binary trees in `recursives` is $0 + 0 = 0$, which is still less than 3, so no pruning happens at this point. Furthermore, $s = 1$ implies that there is one field (`btSize`) for which we should systematically search all values in the scope. We build candidates by assigning `null` as both children and exploring different values

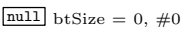

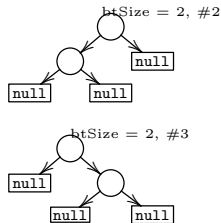
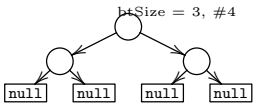
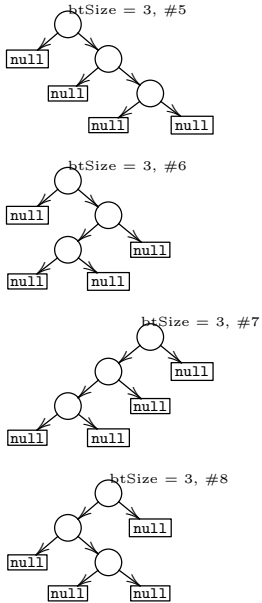
	size = 0	size = 1	size = 2	size = 3
pool				
lastRoundTests				
thisRoundTests				

Figure 6.5: Finding binary trees up to size three (third iteration).

for `btSize` ranging from 0 to 3 (these numbers come from the scope). Each of these candidates is sent to `repOK`, and the one with `btSize` correctly set to 1 returns true. Upon receiving true from `repOK`, the new test is saved in `thisRoundTests` and the algorithm continues until all permutations and valuations are exhausted. At the end of this round, `null` moves to `pool` and tree 1 moves to `lastRoundTests`. The algorithm continues in the same manner. In the third iteration, first, binary trees 2 and 3 are combined with binary tree 0 (`null`) to generate four binary trees of size 3. Then, an example of pruning occurs. Once the algorithm selects tree 2 from `lastRoundTests` and tree 1 from `pool`, the check on the size of `recursives` indicates that the resulting candidate would be outside the scope. Therefore, `nextPermutationPruning` is called to find the next permutation of `recursives` that is inside the scope. Because all other permutations are bigger in size, they are pruned. For brevity, we do not show iteration 4, wherein no new test is generated. The algorithm terminates at the beginning of iteration 5, when all valid inputs are in `pool`.

6.2.2.2 Random Generation

Our use of dynamic programming is orthogonal to random or exhaustive test generation and can be applied to both. In order to generate random tests, we introduce randomization into the process of saving valid inputs. For random test generation, `randomIsOn()` returns true on line 22 of Listing 6.3. Therefore, saving or discarding a correct test input depends on the value returned from `coinToss(size(recursives))`. This method heuristically returns true for all

small inputs (inputs with a size less than a threshold parameter) in order to save them all, and uses a random number generator (a coin toss) with a fixed probability of success to randomly save or discard other correct test inputs. As the algorithm continues to execute, discarded tests do not take part in test generation. At the end, the algorithm generates one or several random tests of a given size. Keeping all small inputs (for example the single-node binary tree) helps in reducing the chance of having repetitive patterns in a random test. In the case of generating multiple tests, if sharing structures between tests is undesirable, one could run the algorithm from scratch multiple times. As Section 6.3 shows, our algorithms are efficient enough to generate big inputs in a matter of seconds and one can run them several times.

6.2.2.3 LazyDP

One problem that arises during the test generation is the limitation of other resources beyond computational time, such as memory. If we keep tests as objects, we run out of the heap memory space for bigger scopes. As Listing 6.3 shows, to optimize memory usage, tests are saved in the compact form of an array of integers. These integers are either indexes of smaller substructures, which in turn point to other arrays of integers, or values of non-recursive fields. For example, tree 5 is saved as {(right child =) 3, (left child =) 0, (btSize =) 3} where tree 3 is in turn kept as {(right child =) 1, (left child =) 0, (btSize =) 2}. The value of a non-recursive field is saved as an index with respect to the scope. For example, if there is a field with primitive

type `boolean` whose values are false and true in the scope, 0 represents false and 1 represents true.

Whenever we create a bigger candidate test input using smaller previously generated tests, we need to call `repOK` to examine the correctness of the candidate. Consequently, we need to retrieve the smaller tests, build their corresponding objects, and then utilize them to build the candidate. We build the smaller tests with the lazy initialization technique, which means that the substructures get initialized (i.e., expanded from arrays of integers to objects) only when `repOK` accesses a field from them.

For binary trees, since `repOK` of Listing 6.1 only accesses `btSize` of the direct children of a node, we only expand the direct children to objects and keep their children as arrays. For example, the expansion of binary tree 5, is limited to expanding binary trees 3 and 0, and not binary tree 1, which is a child of binary tree 3.

6.2.2.4 **SymboLazyDP**

While both dynamic programming and lazy initialization improve finding the recursive values of a candidate, in order to find the correct values for its non-recursive fields (e.g., `btSize`), we still need to search all valuations, which diminishes the efficiency and scalability of test generation. To avoid such a search, we observe that the values of many non-recursive fields of a test can be *symbolically computed* rather than searched for. Following previous work [58, 56, 92] we use symbolic execution for non-recursive fields. Using

symbols (instead of concrete values) for fields, we build a path condition (a boolean formula over the symbols which represents the constraints that should be satisfied to follow a path) while executing `repOK`. At the end of the `repOK` execution, we use a constraint solver to solve the path condition and calculate concrete values for non-recursive fields.

In order to enable symbolic execution, we use a source to source instrumentation on the `repOK` method [56]. We replace each branch condition with a boolean variable which takes both true and false values. When it takes the true value, we add the original branch condition to the path condition. When it takes the false value, we add the negation of the original condition. All valuations of such boolean variables provide all execution paths. Listing 6.4 shows the corresponding instrumentation for `btSize` (replacing the last two lines of Listing 6.1). Upon reaching a return statement, we invoke a constraint solver to solve the path condition and consider each solution as an acceptable valuation for the non-recursive fields. For example, `btSize` of a candidate binary tree is assigned through solving the condition added on line 20 of Listing 6.4.

We save the path condition with each valid substructure, but do not save the solution; i.e., substructures are saved with symbolic non-recursive fields and constraints on them. After combining the substructures, we perform symbolic execution on the entire candidate, including the recursive calls, because solving the substructures separately does not necessarily give compatible results (e.g., consider solving the search constraint on integer elements of a binary search tree; it is possible to save valid integer elements in the right

and left children that violate the search constraint at the root, making the constraint infeasible).

```

19 if(getBoolean()){
20     addCond("btSize", EQ, "rightBtSize+leftBtSize+1");
21     return true;
22 }else{
23     addCond("btSize", NOTEQ, "rightBtSize+leftBtSize+1");
24     return false;}

```

Listing 6.4: Instrumenting BinaryTree for symbolic execution.

6.2.3 Theorem on Test Generation Algorithm

In this section, we prove that the DP algorithm generates all valid tests in the scope (i.e., it is complete), and it does not generate any valid or invalid candidate more than once. The algorithm is sound because of the final repOK check performed on each candidate.

Let us use the number of generation rounds as set subscripts. Define S_R as the value of set S (e.g., pool) at the end of round R . Furthermore, define $discarded_R$ as the set of candidates discarded during round R , including those outside the scope.

Definition 1. *Visited candidates:* $visitedCands_R = pool_R \cup lastRoundTests_R \cup \bigcup_{i=1}^R discarded_i$.

Definition 2. $T(C)$: For a candidate C , let $T(C)$ be the set of all test inputs that C uses as its recursive substructures.

Lemma 1. At the end of each round R ($R \geq 0$), and for any candidate C , the following loop invariant holds:

$$T(C) \subseteq \mathbf{pool}_R \rightarrow C \in \mathbf{visitedCands}_R$$

The invariant means that, at the end of each round, any candidate that uses only `pool` members is already generated.

Proof by induction. The base case is the beginning of round one, where $\mathbf{pool}_0 = \emptyset$. The only candidate that does not use any recursive substructures is `null`. Yet `null` \in `lastRoundTests`₀ and hence `null` \in `visitedCands`₀, so the invariant holds.

For the induction step, assume that:

$$T(C) \subseteq \mathbf{pool}_R \rightarrow C \in \mathbf{visitedCands}_R \quad (6.1)$$

At the end of round $R + 1$, the tests in `lastRoundTests` join `pool`, and then the tests in `thisRoundTests` replace `lastRoundTests`.

$$\mathbf{pool}_{R+1} = \mathbf{pool}_R \cup \mathbf{lastRoundTests}_R \quad (6.2)$$

$$\mathbf{lastRoundTests}_{R+1} = \mathbf{thisRoundTests}_{R+1} \quad (6.3)$$

Suppose that the invariant does not hold at the end of round $R + 1$.

$$\exists C' : T(C') \subseteq \mathbf{pool}_{R+1} \wedge C' \notin \mathbf{visitedCands}_{R+1} \quad (6.4)$$

Now, $T(C') \cap \mathbf{lastRoundTests}_R$ is either (a) $= \emptyset$ or (b) $\neq \emptyset$. For case (a):

$$(6.2) \wedge (6.4) \wedge (a) \rightarrow T(C') \subseteq \mathbf{pool}_R \quad (6.5)$$

$$(6.1) \wedge (6.5) \rightarrow C' \in \mathbf{visitedCands}_R \quad (6.6)$$

$$Definition1 \wedge (6.2) \wedge (6.6) \rightarrow C' \in visitedCands_{R+1} \quad (6.7)$$

which contradicts with (6.4).

For case (b):

$$\exists t \in T(C') : t \in lastRoundTests_R \quad (6.8)$$

Note that the algorithm generates all permutations of the tests belonging to `lastRoundTests` or `pool` that have at least one test from `lastRoundTests`, i.e.,

$$\begin{aligned} & (T(C) \subseteq pool_R \cup lastRoundTests_R \\ & \wedge \exists t \in T(C) : t \in lastRoundTests_R) \\ \leftrightarrow & C \in thisRoundTests_{R+1} \cup discarded_{R+1} \end{aligned} \quad (6.9)$$

$$(6.2) \wedge (6.4) \wedge (6.8) \wedge (6.9) \rightarrow C' \in thisRoundTests_{R+1} \cup discarded_{R+1} \quad (6.10)$$

$$Definition1 \wedge (6.3) \wedge (6.10) \rightarrow C' \in visitedCands_{R+1} \quad (6.11)$$

which again contradicts with (6.4). □

Theorem 1. *Part 1: for a given scope, the algorithm generates all valid tests. Part 2: the algorithm does not generate any (valid or invalid) candidate more than once.*

Proof of part 1. Let C'' be the smallest correct test that the algorithm fails to generate. If the algorithm terminates right after round fin :

$$repOK(C'') = true \wedge C'' \notin visitedCands_{fin} \quad (6.12)$$

$$\begin{aligned} \forall C : \text{repOK}(C) = \text{true} \wedge C \notin \text{visitedCands}_{fin} \\ \rightarrow \text{size}(C) \geq \text{size}(C'') \end{aligned} \quad (6.13)$$

The sizes of the valid tests that C'' uses as its recursive substructures are less than the size of C'' ; because size is defined as the sum of the sizes of the recursive substructures plus one.

$$\forall t \in T(C'') : \text{size}(t) < \text{size}(C'') \quad (6.14)$$

$$\forall t \in T(C'') : \text{repOK}(t) = \text{true} \quad (6.15)$$

Because C'' is assumed to be the smallest test that is not generated, we can show that all of its substructures are generated:

$$(6.13) \wedge (6.14) \wedge (6.15) \rightarrow \forall t \in T(C'') : t \in \text{visitedCands}_{fin} \quad (6.16)$$

Further, the termination of the algorithm at the end of round fin means that no progress was made in this round. Accordingly, no test was added to $\text{thisRoundTests}_{fin}$. By using (6.3)²:

$$\begin{aligned} \text{lastRoundTests}_{fin} = \text{thisRoundTests}_{fin} = \emptyset \\ \rightarrow \text{visitedCands}_{fin} = \text{pool}_{fin} \cup \bigcup_{i=1}^{fin} \text{discarded}_i \end{aligned} \quad (6.17)$$

Now, let us assume that C'' is inside the scope.

$$\text{size}(C'') \leq \text{scope.size} \quad (6.18)$$

² $fin > 0$ since the test generation loop executes at least once.

$$(6.14) \wedge (6.18) \rightarrow \forall t \in T(C'') : size(t) \leq scope.size \quad (6.19)$$

From the definition of *discarded*, we have:

$$\forall C \in discarded_R \rightarrow repOK(C) \neq true \vee size(C) > scope.size \quad (6.20)$$

$$(6.15) \wedge (6.19) \wedge (6.20) \rightarrow \forall t \in T(C'') : t \notin \bigcup_{i=1}^{fin} discarded_i \quad (6.21)$$

$$(6.16) \wedge (6.17) \wedge (6.21) \rightarrow \forall t \in T(C'') : t \in pool_{fin} \quad (6.22)$$

$$Lemma1 \wedge (6.22) \rightarrow C'' \in visitedCands_{fin} \quad (6.23)$$

But (6.23) contradicts with (6.12), which proves part 1. \square

Proof of part 2 by induction. Consider $R = 0$ for the induction base. Because only one instance of `null` is generated before the first round, no repetition happens at $R = 0$.

For the induction step, assume that all candidates visited up to the end of round R are distinct. We use \neq to show that two candidates are different instances, although they might be equal ($=$).

$$\nexists t, t' \in visitedCands_R : t \neq t' \wedge t = t' \quad (6.24)$$

Suppose that the first repetitious candidate, named C , is generated during round $R + 1$.

$$C \in thisRoundTests_{R+1} \cup discarded_{R+1} \quad (6.25)$$

C is repetitious, so another instance of it, named C' , is already generated at round R^3 .

$$C' \in \text{thisRoundTests}_{R'} \cup \text{discarded}_{R'} \quad (6.26)$$

$$(6.9) \wedge (6.25) \rightarrow T(C) \subseteq \text{pool}_R \cup \text{lastRoundTests}_R \quad (6.27)$$

$$(6.9) \wedge (6.26) \rightarrow T(C') \subseteq \text{pool}_{R'-1} \cup \text{lastRoundTests}_{R'-1} \quad (6.28)$$

Either (a) $R' = R + 1$ or (b) $R' < R + 1$. For case (a), note that during one round of test generation, methods `nextPermutation` and `nextPermutationPruning` provide distinct permutations. So in order to have repetitive candidates, at least one of the substructures should have more than one instance.

$$\exists t \in T(C), t' \in T(C') : t \neq t' \wedge t = t' \quad (6.29)$$

$$(6.27) \wedge (6.28) \wedge (6.29) \wedge (a) \rightarrow \exists t, t' \in \text{visitedCands}_R : t \neq t' \wedge t = t' \quad (6.30)$$

which contradicts with (6.24).

For case (b), notice that at least one substructure is selected from `lastRoundTests`.

$$(6.9) \wedge (6.25) \wedge C = C' \rightarrow \exists t \in T(C') : t \in \text{lastRoundTests}_R \quad (6.31)$$

$$(6.28) \wedge (6.31) \rightarrow \exists t \in \text{lastRoundTests}_R \cap \text{visitedCands}_{R'-1} \quad (6.32)$$

$$(6.24) \wedge (6.32) \rightarrow R \leq R' - 1 \quad (6.33)$$

which contradicts with (b). □

³The algorithm never generates `null` again, so $R' \neq 0$.

6.3 Evaluation: Test Input Generation Using Dynamic Programming

In order to evaluate our test generation methods, we implemented a prototype of the algorithms and designed some experiments wherein we address two research questions:

- **RQ1** How efficient and scalable are our algorithms, compared to state-of-the-art test generation tools (Pex and Korat)?
- **RQ2** How effective are the generated tests in finding bugs in real world applications (Chrome and Safari web browsers)?

In the first set of experiments, we used six microbenchmarks, which are complex data structures widely used in programs and as test inputs. Previous work has extensively used these benchmarks for evaluation [85, 78, 89, 81, 15, 37, 67]. In order to answer the first question, we need an alternative exploration method of the state space of possible test inputs. A naive exploration of the state space will give rather unacceptable results. Therefore, we compare our methods to Microsoft Pex [92]—a state-of-the-art test generation tool for .Net—and Korat [15]—a well-known test generation method and an open source tool [74] for Java programs.

Pex is a white-box test generation tool that performs symbolic execution. In addition, it uses path-bounded model-checking to cover different paths in the program. Pex is an appropriate subject tool; it particularly addresses

the effect of symbolic execution. Generation of test inputs considered in this work is black-box with respect to the code under test, yet we allow Pex to explore different paths in repOK. We used the same repOK methods for Pex and Korat, except for minor changes to accommodate syntactic differences between C# and Java respectively.

The Korat algorithm monitors repOK executions to prune large portions of the bounded space of candidate structures. Korat is an appropriate subject tool too; previous work [87] shows that Korat is among the most efficient solvers for *complex structural constraints*, when compared to other techniques (JPF model checker [95], Alloy alongside with a SAT solver [52], and CUTE symbolic execution engine [85]).

In addition to exhaustively generating test inputs, we compare the efficiency and scalability of our algorithms with Pex and Korat, when sampling a few large test inputs.

In the second set of experiments, we show how to naturally model HTML and CSS3 files as acyclic data structures. Such files, which are test inputs to any web browser, are examples of practical and common, yet user-defined test inputs. By systematically generating HTML and CSS3 test inputs, our generation methods tested the latest versions of two well-known web browsers, Google Chrome and Apple Safari, and found real bugs in Chrome.

6.3.1 Experimental Settings

Throughout the evaluation, we ran each experiment five times and reported the averages. All experiments used a 2.50GHz Core 2 Duo processor with 4.00GB RAM running Windows 7. We used Sun’s Java SDK 1.6.0 JVM with our methods and Korat, and Microsoft Visual C# 2010 version 4.0.30319 RTMRel with Pex version 0.94.51006.1. Pex used Z3 theorem prover [28] version 2.0. In Section 6.3.4, we used Google Chrome version 13.0.782.220 m for Windows and Apple Safari 5.1 (7534.50), the latest versions as of the date of these experiments.

For symbolic execution, we used our in-house constraint solver developed in Java. The source to source instrumentation for symbolic execution is currently manual, but is mechanical and can be automated [56]. For exhaustive test generation with Pex, we used the following setting to force Pex to generate all test inputs:

```
TestEmissionFilter = PexTestEmissionFilter.All
```

6.3.2 Microbenchmarks

To address RQ1 for exhaustive test generation, we considered six microbenchmarks.

Table 6.1 shows the results for the biggest sizes considered. For all six microbenchmarks and all sizes considered (including those not shown), DP and LazyDP generate the same number of tests as Korat. SymboLazyDP and Pex

Table 6.1: Exhaustive test generation for the biggest sizes considered. Benchmarks include sorted singly-linked lists (LL), binary trees (BT), red-black trees (RBT), Fibonacci heaps (FH), binary heaps (BH), and hash tables (HT). TO represents a timeout of 1000s. Best performance highlighted.

Bench.	Size	Valid Tests				Candidates				Generation Time (s)				State Space	
		Korat/DP/		Symb-		Korat		DP/		Korat		DP		Pex	Pex
		LazyDP	LazyDP	LazyDP	LazyDP	LazyDP	LazyDP	LazyDP	LazyDP	LazyDP	LazyDP	LazyDP	LazyDP	Symbolic	Symbolic
LL	17	131072	18	18	17825963	2228208	34	36	33.188	7.089	3.491	0.025	9.195	10 ⁴¹	
BT	12	290512	290512	TO	15770974	3776644	581023	TO	51.533	17.524	7.947	7.416	TO	10 ²⁶	
RBT	9	6753	271	TO	2207699	610217461	445227	TO	8.728	962.483	TO	4.590	TO	10 ²⁶	
FH	6	1125139	197	TO	1364398	2101957	3190	TO	5.654	10.254	16.035	0.078	TO	10 ¹⁴	
BH	8	344571	9	9	24900165	186644641	3593	70	65.129	300.862	283.099	0.075	9.776	10 ²¹	
HT	12	4083	233	TO	11098075	31953529	4176	TO	51.043	75.834	56.740	0.159	TO	10 ³⁸	

generate the same number of tests, since they both use symbolic execution and report one solution for each path condition, instead of exploring all valuations from the state space. In addition to improving test generation performance and scalability, symbolic execution improves *test execution* by reporting less, yet representative test inputs. SymboLazyDP is the most efficient generation method for all the microbenchmarks on the biggest size.

We consider six microbenchmarks: sorted singly-linked lists (LL), binary trees (BT), red-black trees (RBT), Fibonacci heaps (FH), binary heaps (BH), and hash tables (HT). The first data structure is a sorted singly-linked list of integer elements. In addition to acyclicity, a sorted list requires the integer elements to be sorted in ascending order. Repetitions are not allowed. Figure 6.6 shows the performance evaluation results for linked lists. Here, DP and LazyDP constantly outperform Korat. Lazy initialization is effective here, especially because properties like being sorted are verified locally: it suffices to compare each node's element with its neighbor's. Pex outperforms DP and LazyDP on bigger sizes for two data structures: singly-linked list and binary heap. The reason is that Pex generates far fewer tests by symbolically executing repOK and representing all sorted lists of a given size with only one test, while Korat, DP, and LazyDP exhaust different valuations. SymboLazyDP, however, shows the best performance among all the methods. It generates the same number of tests and grows with the same pace as Pex, but it is multiple times faster. Notice that the vertical axis is logarithmic.

The next benchmark is a binary tree as described in Section 6.1. As

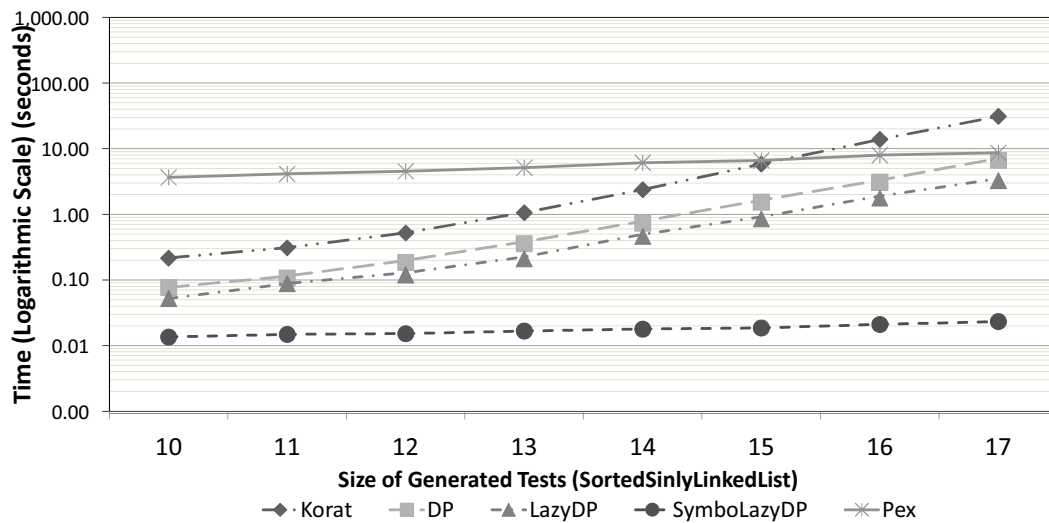


Figure 6.6: Performance comparison on linked lists.

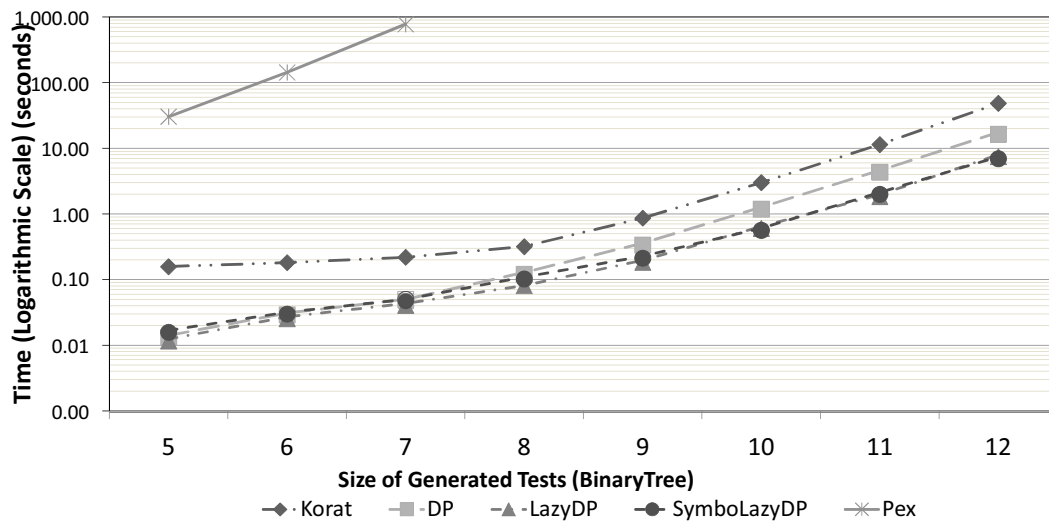


Figure 6.7: Performance comparison on binary trees.

Figure 6.7 displays, SymbolLazyDP performs the best. Pex can only enumerate all binary trees up to size 7 before timing out.

We also experimented with red-black trees with size, key and color. Our repOK enforces the following properties: all leaves are black, both children of a red node are black, every path from a node to any of its descendant leaves contains the same number of black nodes, and keys maintain the binary search tree property (the key of each node is bigger than all keys in the left sub-tree and smaller than all keys in the right sub-tree). Figure 6.8 depicts the results. While DP and LazyDP are faster than Korat at first, Korat takes over them at some point because DP and LazyDP have to explore all valuations of key and color, but Korat prunes many of them. Notice that LazyDP takes slightly more time than DP on this benchmark. This is because repOK needs to explore down the tree to find minimum and maximum keys to evaluate the search tree property, and eventually expands many substructures, which undermines the usage of lazy initialization. Pex uses symbolic execution for key and color, but cannot generate all red-black trees with four nodes or more. SymboLazyDP, using dynamic programming and symbolic execution, closely competes with Korat and eventually takes over.

Figure 6.9 compares the memory usage of our methods and Korat for red-black trees. DP and LazyDP generate the same number of tests, so they always use equal amounts of memory. SymboLazyDP generates less tests and hence uses less memory. Korat keeps only one candidate vector while we have to keep all correct tests (albeit in a compact format). Hence, the memory usage of our methods grows faster than Korat. However, since the Java heap space is usually in the order of a few GB's, 10MB of memory usage should

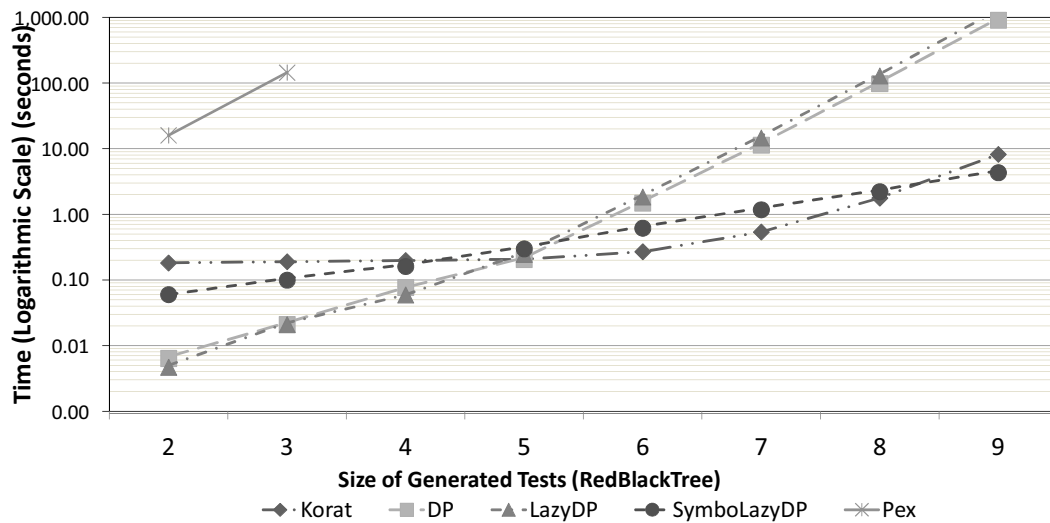


Figure 6.8: Performance comparison on red-black trees.

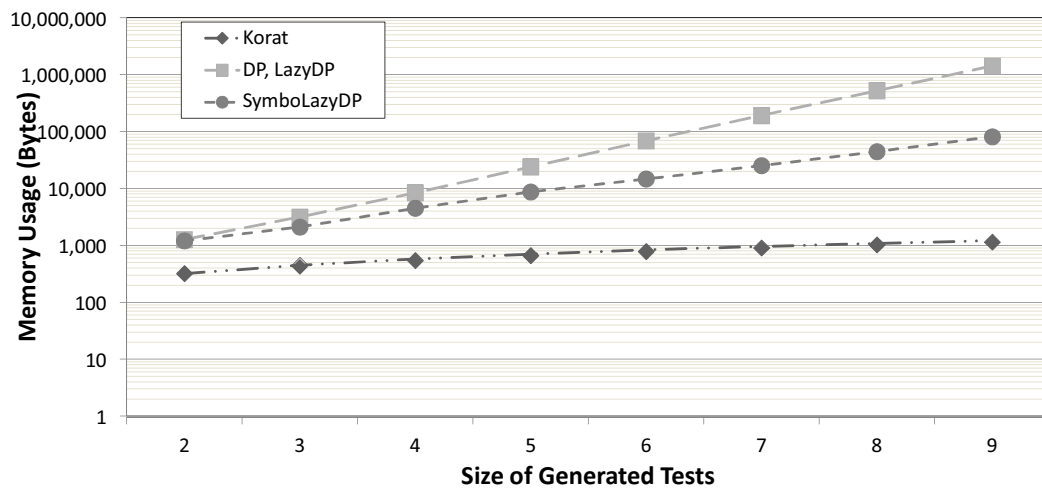


Figure 6.9: Memory usage on red-black trees.

not be a problem. The memory usage of the other benchmarks follows a very similar pattern.

The next benchmark is a Fibonacci heap (Figure 6.10), a collection of

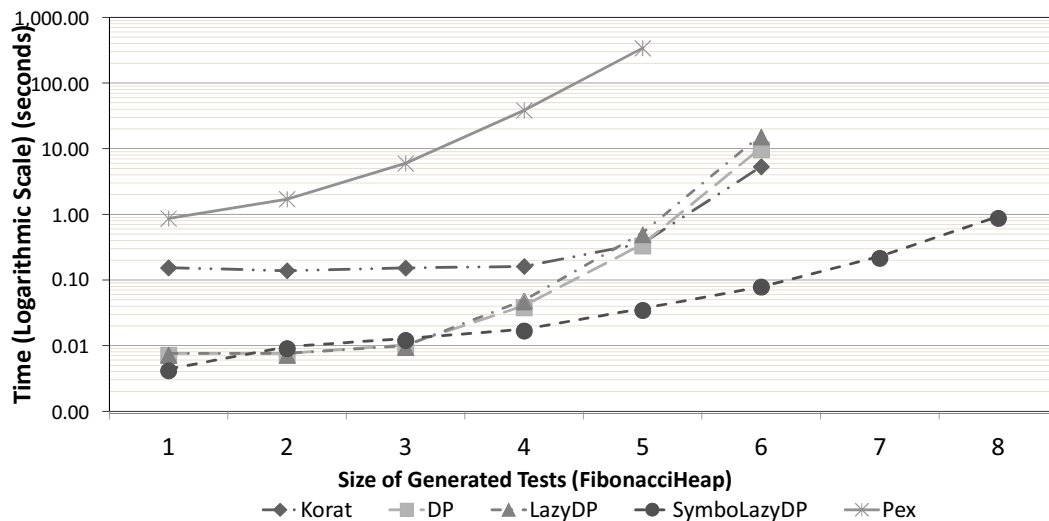


Figure 6.10: Performance comparison on Fibonacci heaps.

trees with no limit on the number of children of a node. The heap maintains minimum-heap property (the key of a node is always less than the keys of its children). The Korat implementation of repOK for this class is recursive which shows that an intuitive way of checking the properties of this class is through recursion. Nevertheless, Korat does not make use of recursion for test generation. This is the only case where the memory usage is a concern for DP and LazyDP. In fact, for Fibonacci heaps with more than six nodes, DP and LazyDP run out of the heap space and Korat and Pex run out of time. Yet, SymboLazyDP does not time out and is the most efficient.

The next benchmark, a binary heap (Figure 6.11), is a complete binary tree (all levels of the tree, except possibly the last one, are fully filled), the tree is balanced, and the leaves of the last level are filled from left to right.

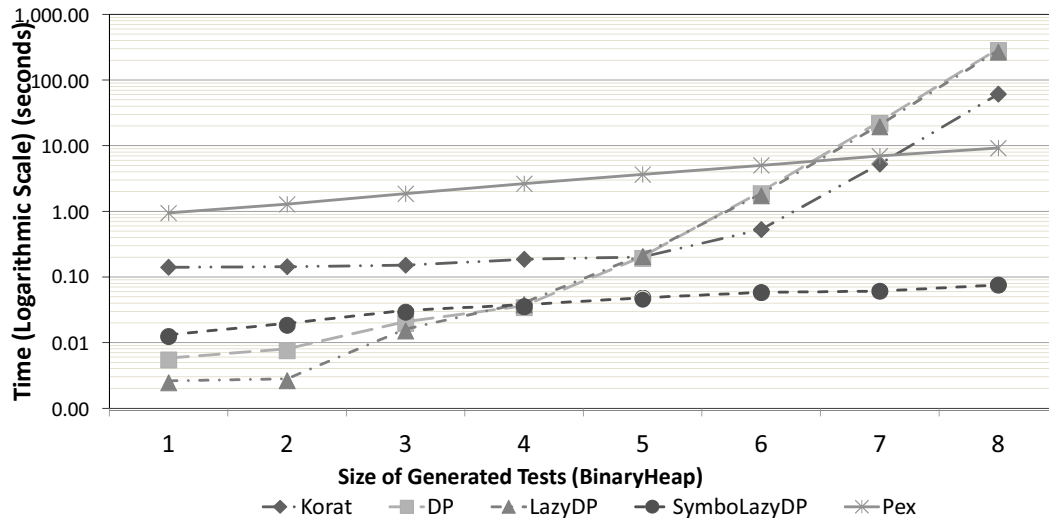


Figure 6.11: Performance comparison on binary heaps.

Also, it maintains the minimum-heap property. This benchmark gives results similar to the sorted singly-linked list. SymboLazyDP is the most efficient.

The last benchmark is a hash table implemented using nested lists (Figure 6.12). Each integer element in the hash table is hashed to a value using a given hash function. Entries with the same hash value are kept in a list. LazyDP and DP are slightly better than Korat. The difference increases when a time-consuming hash function is used. Since we use previously generated tests, we avoid many calls to the hash function. The generation time for Pex starts off at a bigger value and increases at the same pace as SymboLazyDP. SymboLazyDP outperforms all the other methods.

Finally, it is worth mentioning that Pex and SymboLazyDP both require path bounds (e.g., for loop unrolling). Throughout the experiments, we

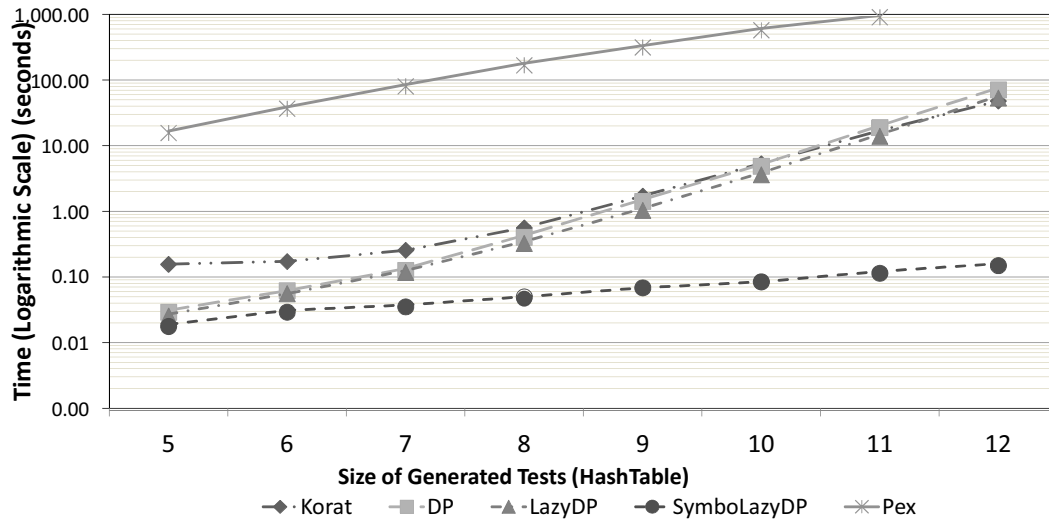


Figure 6.12: Performance comparison on hash tables.

used trial and error to find and set the smallest bounds that provide all tests. Similarly, Korat, DP, and LazyDP need a bound on primitive (e.g., integer) values. The number of primitives used usually has a relationship with the test size.

6.3.3 Random Test Generation

To address RQ1 for random test generation, we considered generating ten random tests of ninety to a hundred nodes (Table 6.2). For Korat and Pex, we took the first ten tests generated in the desired size range. For SymboLazyDP, all small tests (up to size 3) were saved and bigger tests were saved or discarded at random (Section 6.2.2.2). Except for the Fibonacci heap benchmark, SymboLazyDP is the most efficient and scalable.

Table 6.2: Random generation of ten tests with $90 \leq \text{size} \leq 100$. Benchmarks include sorted singly-linked lists (LL), binary trees (BT), red-black trees (RBT), Fibonacci heaps (FH), binary heaps (BH), and hash tables (HT). TO represents a timeout of 1000s. Best performance highlighted.

Bench.	Valid Tests			Candidates			Generation Time (s)			State	
	Korat	SymboLazyDP	Pex	Korat	SymboLazyDP	Pex	Korat	SymboLazyDP	Pex	Space	Space
LL	10	10	10	8110	197	198	0.137	0.136	322	322	10^{351}
BT	10	10	10	8746	11615	602	0.266	0.174	375	375	10^{353}
RBT	TO	10	TO	TO	2350732	TO	TO	20.256	TO	TO	10^{529}
FH	10	10	TO	4106	54624	TO	0.114	0.567	TO	TO	10^{527}
BH	10	10	10	343300	467522	153	7.617	2.823	83	83	10^{527}
HT	TO	10	TO	TO	153508	TO	TO	3.947	TO	TO	10^{527}

6.3.4 Google Chrome and Apple Safari

To address RQ2 and showcase the ability of our methods in finding bugs in real world, well-tested⁴, commercial applications, we tested the support for rendering CSS3 3D effects by Chrome and Safari web browsers. CSS (Cascading Style Sheets) is a style sheet language that separates the presentation of a markup language document from its content, and is commonly used to style web pages written in HTML and XHTML. CSS3, the latest variation of CSS, enables web developers to add 3D effects to web pages, i.e., position and move elements in the three dimensional space.

Apple Safari and Google Chrome web browsers support CSS3 3D effects. As of the date of these experiments, Chrome and Safari are the third and fourth most widely used desktop web browsers with 21.5% and 4.8% worldwide usage share respectively [7]. Safari is developed in C++ and Objective-C, and precedes Chrome in supporting 3D transforms. Chrome is developed in C++, Assembly, Python, and JavaScript. Both of these browsers use Webkit layout engine which introduced 3D transforms in CSS.

We directly tested 11 KLOC of C++ code (74 .cc/.h files) from Chrome. Safari is 37 MB compiled. Our test for Safari included 2.7 KLOC (19 .cpp/.h files) of its open source code plus its closed source implementation.

⁴For example, Chrome is extensively tested before release and claims to pass 99% of WebKit's layout tests [5]. The CSS3 3D effects are among the WebKit's layouts.

6.3.4.1 Modeling HTML and CSS Test Inputs

An HTML file is composed of a set of nested HTML elements. An HTML element includes a start tag (e.g., `<h1>`) and an end tag (e.g., `</h1>`). The start tag might also have some attributes (e.g., `class="ClassName"`). We modeled an instance of an HTML file as a tree. The whole document is contained between `<html>` start and end tags, which we consider as the root of the tree. Further, each tag is represented as a node that has some attributes and an ordered set of children, which are the tags immediately inside it. Listing 6.2 shows some parts of the HTML model. Figure 6.2 shows the tree representation of Listing 6.5.

```
1 <html>
2   <head>
3     <link rel="stylesheet" type="text/css" href="file.css">/link>/head>
4   <body>
5     <div class="ClassName4">
6       <h1>This is some text
7         <div class="ClassName12">
8           <h1>This is some text</h1>/div>/h1>/div>/body>/html>
```

Listing 6.5: An automatically generated HTML test input.

```
1 selector1 {
2   property1: value11 [ value12 ...];
3   [property2: value12 [value22 ...];
4   ...]}
```

Listing 6.6: Abstraction of a CSS rule.

A CSS file consists of a list of rules. A rule has a selector and a declaration block. Inside a block, each declaration has a property, followed by a list of values. Listing 6.6 shows an abstraction of a CSS rule. Since multiple

```
1 .ClassName4{
2   -webkit-transform: rotateY( 180deg );}
3 .ClassName12{
4   -webkit-perspective: 800;
5   -webkit-backface-visibility: hidden;}
```

Listing 6.7: An automatically generated CSS test input (file.css).

selectors can be modeled by duplicating the declaration block for each of them, we only support single selectors. We modeled each CSS rule as a linked list of alphabetically sorted⁵ properties where each property has a linked list of values. As one could see, our HTML and CSS models are intuitive and easy to implement as recursive loop-free data structures. Indeed, we have already implemented both of them as microbenchmarks.

Listings 6.7 and 6.5 show bug-revealing examples of HTML and CSS inputs, automatically generated by our methods.

6.3.4.2 Experimental Results

Using the above models, we systematically generated all test inputs with up to eight tags (two `<div>` tags) inside an HTML file and two declarations inside a CSS declaration block. Five CSS properties were used: perspective, backface-visibility, transform, transform-origin, and transform-style. Also, various values for these properties were used including perspective, rotate, scale, skew, and translate for the transform property. Each HTML tag could have a CSS selector as its class attribute (See Listings 6.7 and 6.5 as

⁵Because the order of properties is irrelevant, we keep them sorted to avoid duplicates.

Table 6.3: Chrome and Safari test input generation results.

	Candidates		Gen. Time (s)		#Tests
	CSS	HTML	CSS	HTML	
DP	10,231	3,815,626	0.140	10.851	3081
LazyDP	10,231	3,815,626	0.140	10.850	3081
SymboLazyDP	N/A	116,766	N/A	1.628	3081

an example). Consistency constraints between CSS and HTML files are maintained by first running the CSS input generator and then feeding the number of classes it generates to the HTML input generator to exhaustively cover all classes.

Table 6.3 shows a summary of the results. In total, 3081 test inputs (each including an HTML file with the corresponding CSS file) were generated. The size of the input space for the chosen bounds is 10^{10} . The size bound in this experiment is too small to get LazyDP benefits. However, SymboLazyDP gives a clear advantage. In our model, SymboLazyDP is not applicable on CSS inputs for the lack of non-recursive fields that can be executed symbolically.

6.3.4.3 Differential Testing

So far, we have automatically generated the test inputs. But in order to test Chrome and Safari with these tests, we need an oracle that defines the correct rendered output for any given test input. Since no such oracle was available⁶, we use differential testing [70], where the outputs of two im-

⁶In some domains, it is possible to exploit domain knowledge to define specific purpose oracles, as we did in a previous work to test Android apps [108].

plementations are checked against each other. Whenever the outputs are not the same, there likely is a bug in at least one of the implementations. We wrote a test harness in Java that automatically launches Chrome and Safari with each test input, and performs a basic image differencing algorithm to compare screen shots taken from them. All test inputs were checked in less than 2 hours. Such time-consuming checks are specific to this application. Furthermore, improving the performance of launching the browsers and image differencing is possible, yet beyond the scope of this work.

6.3.4.4 Bugs Found

Among the 3081 tests generated, 818 tests were rendered differently by Chrome and Safari. We semi-automatically investigated these tests. Out of these 818 failures, 148 cases were false positive due to the inaccuracy of our image differencing algorithm. We manually classified the rest of the failing tests (670 tests) based on the CSS properties used, and found at least three distinct bugs in the production code, stable release of Chrome. The actual number of faults in the code, which produce these failures, in fact, may be greater. However, localizing the faults was beyond the scope of this work.

We found three bugs in Chrome. One of these bugs was regarding the hidden backface-visibility of an element. Listings 6.7 and 6.5 reveal this bug. Figure 6.13 shows the faulty output of Chrome (left) versus the expected output of Safari (right). The second line should be hidden because it is showing its back-face (it is inside `ClassName4`) and has its backface-visibility set to



Figure 6.13: A back-face visibility bug found in Chrome (left). Safari (right) shows the expected output.

hidden (it is inside `ClassName12`). This bug in Chrome was already reported and confirmed (issue 76947 in the Chromium project) and is fixed in the next Canary release.

Another bug involved the `webkit-perspective` property. We reported this bug (issue 93682 in the Chromium project). This bug is now confirmed and fixed. Listing 6.8 shows a manually simplified version of this bug as a single HTML file. Figure 6.14 shows the outputs.

The last bug was due to a rotation direction inconsistency with the W3C editor's draft (21 March 2011) [3] as the standardization in progress of CSS 3D transforms. As Listing 6.9 simplifies, the red box has a positive rotation around the Y axis (whose positive direction is down). According to the standardization, such a rotation should be performed clockwise. Figure 6.15 shows snapshots of this bug. This bug was fixed independently.

Consider Listings 6.7 and 6.5. To reveal this bug, we need two nested classes where the outer one has a 180 degree rotation and the inner one has

```

1 <head>
2   <style media="screen">
3     .red {
4       -webkit-perspective: 800;
5     }
6     .box{
7       background-color: red;
8       -webkit-transform: rotateY( 45deg );
9     }
10  </style>
11 </head>
12 <body>
13   <div class="red">
14     <div class="box">
15       This box should be rotated, but it's not! Only the text looks kind of
16         weird.
17     </div>
18   </div>

```

Listing 6.8: Simplified HTML/CSS test input that reveals the webkit-perspective bug in Chrome.



Figure 6.14: A webkit-perspective bug found in Chrome (up). Safari (down) shows the expected output.

```
1 <head>
2   <style media="screen">
3
4     .container {
5       width: 200px;
6       height: 200px;
7       border: 1px solid #CCC;
8       margin: 0 auto 40px;
9     }
10
11    .box {
12      width: 100%;
13      height: 100%;
14    }
15
16    #red .box {
17      background-color: red;
18      -webkit-transform: perspective( 600 ) rotateY( 45deg );
19    }
20
21  </style>
22 </head>
23 <body>
24   <section id="red" class="container">
25     <div class="box">test</div>
26   </section>
27 </body>
```

Listing 6.9: Simplified HTML/CSS test input that reveals the rotation direction bug in Chrome.

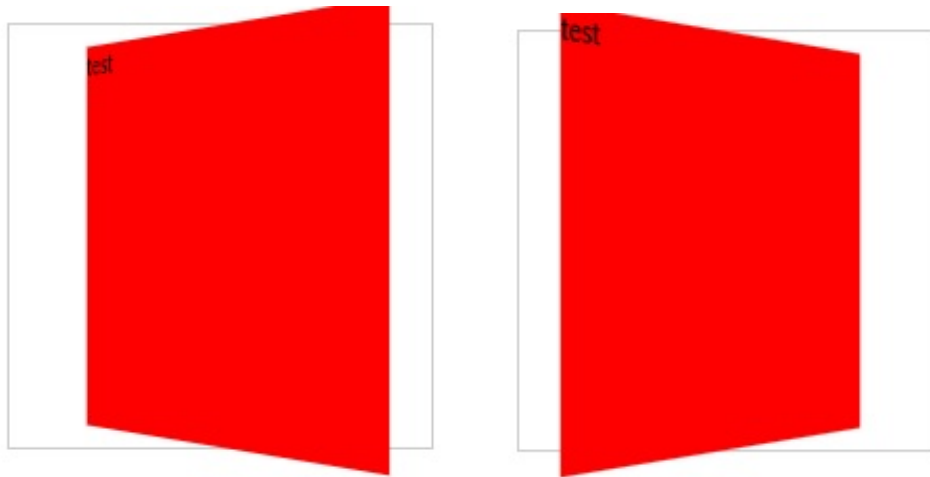


Figure 6.15: A rotation direction bug found in Chrome (left). Safari (right) shows the expected output.

a hidden visibility as well as the `webkit-perspective` property. Invoking the hidden visibility by itself or in any other setting is not enough to show the problem.

6.3.4.5 Applying Symbolic Execution and Korat

We strove to use symbolic execution on the source code available from Chrome. The corresponding code, however, includes 74 `.cc/.h` files (11 KLOC of code) that collectively render a CSS 3D effect. We were unable to apply white box symbolic execution due to the code size and complexity. Symbolic execution is not feasible for testing closed source systems (Safari). Korat can, in principle, find the bugs if given enough time. Yet, as we showed, our technique outperforms Korat in all the cases of exhaustive test generation.

6.3.5 Threats to Validity

Internal validity. (1) To implement our algorithms for test generation using dynamic programming, we strictly followed the original algorithms, used well-known libraries, and validated the number of inputs generated to match the numbers generated by other independently developed tools, namely Korat and Pex. (2) To compare with Korat, we used its open-source implementation that has been in the public domain for over five years, and used the repOK's that are distributed with it. (3) To compare with Pex, we used its public distribution (version 0.94.51006.1) while setting the search depth bound to the smallest number required to complete the generation of all inputs within the chosen size in order to minimize the exploration time for Pex. We carefully performed a faithful translation of Java repOK's used for Korat into C# to run Pex. We used our own in-house constraint solver developed in Java for symbolic execution with dynamic programming, which might give different performance results compared to Z3 [28] used by Pex. It is unlikely that our solver in Java is faster than the state-of-the-art Z3.

External validity. The main threat here involves using only two industrial programs (Chrome and Safari). To address this threat, we experimented with microbenchmarks that have previously been used by a number of other authors [85, 78, 89, 81, 15, 37, 67].

Construct validity. We used metrics commonly used in software testing research to compare test generation tools, and automated our entire test generation and execution process. Furthermore, we manually investigated the

failures reported for the browser testing.

6.4 Applicability

Our work directly enables systematic (i.e., bounded exhaustive) testing to scale better for certain applications, e.g., refactoring engines, compilers, model checkers, and browsers, which clearly must be tested against larger inputs. The inputs to these applications are programs themselves, which can be modeled and generated at the Abstract Syntax Tree level—an acyclic structure—using structural constraints. E.g., Alloy programs were modeled and generated to find bugs in Alloy-alpha [67]; more recently, systematic testing found bugs in Eclipse, NetBeans, Sun javac, and JPF [37, 27].

More generally, our work can help systematic grammar-based testing techniques [68, 61]. Such techniques enumerate all strings, up to a given bound, that belong to a context-free grammar. Context-free grammars can describe various input types, such as XML schemas and programs. To illustrate, consider the work of Khalek et al. [54], which uses constraint-based testing to reveal bugs in Oracle 11g. It enumerates solutions for a subset of SQL grammar and a schema to provide queries and populate the database. Our technique has a direct application in generating strings that belong to the SQL grammar, and can also improve the generation of tabular test data.

Our use of dynamic programming is not limited to bounded exhaustive generation, rather our technique also facilitates random test generation, which complements systematic testing and has also been used successfully to find

bugs [36, 23, 78, 40]. Most recently, Yang et al. used random test generation to find numerous bugs in mainstream C compilers [101]. As our work shows, dynamic programming can be used in synergy with random test generation.

While we describe algorithms for generating recursive structures without cycles, our approach can be used to generate cyclic structures as a part of a multi-step generation technique. For example, we can generate an acyclic backbone in the first step using dynamic programming and populate the remaining fields using constraint-based data structure repair [35] in the second step.

6.5 Ideas on Leveraging Dynamic Programming for Repair

Another research thrust is based on our insight into the similarities in the problem solving structures of constraint-based test generation and contract-driven repair. Test generation frameworks have already been used as special solvers for data structure repair [32, 33]. We generalize these specific usages and observe that both constraint-based test generation and contract-driven data structure repair find data structures that satisfy given specifications (constraints or contracts). The former provides those data structures as test inputs while the latter uses them as repair candidates. By identifying this similarity, we discuss future ideas for the unification of these two problems.

Our idea combines recursive and memoized checks of specifications [86] with the dynamic programming scheme. By utilizing the recursive nature of

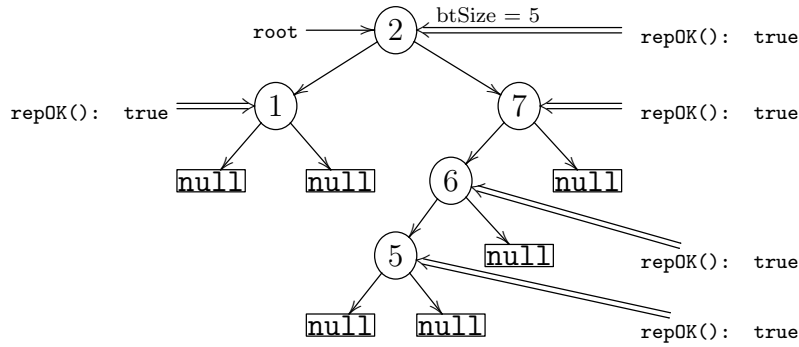
common data structures, (1) we recursively apply the contracts (including `repOK` methods and post-conditions) to locate the fault in the state, and (2) we use a set of small pre-generated substructures built using our structure generation technique as patches to repair the corrupted section of the state.

6.5.1 Localizing Errors with Recursive Contracts

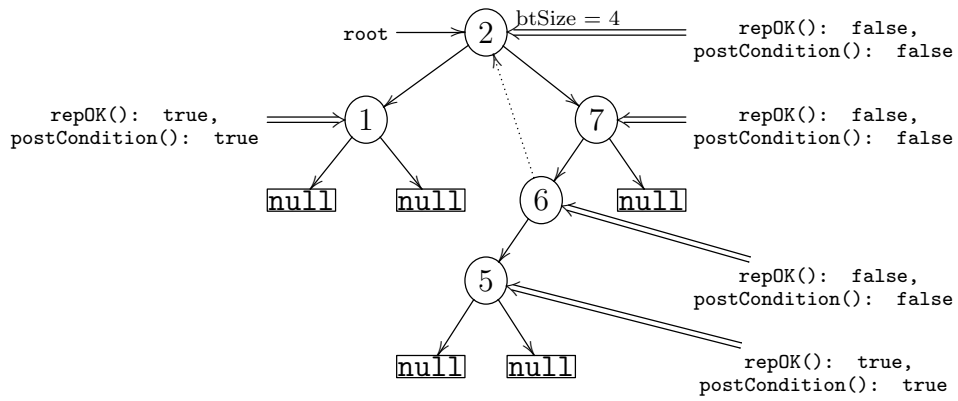
To localize the error in the data structure, we build on the technique presented in Ditto [86] and improve it with the support for (1) cycles and (2) pre- and post-conditions. Ditto is a framework that incrementally checks data structure invariants (i.e., only `repOK`) in recursive data structures. Assuming a recursive `repOK`, Ditto memoizes the result of previous checks and re-checks the invariants only on the parts that have been modified since the last check. Ditto uses write barriers to identify those modified parts for checking (similarly to what we did in Chapter 4 for repair). To illustrate, consider the result of the faulty `remove(7)` with bug `cycle` shown in Figure 6.16. Ditto assumes that the recursive `repOK` has already been executed on the input and the results are memoized (Figure 6.16 (a)). After running `remove(7)`, Ditto monitors the write barrier log (dotted lines) to identify the parts of the data structure that have changed, and checks `repOK` anew only on those parts. For example, in Figure 6.16 (b), `repOK` is *not* called on `[1]` and `[5]` again, because the write barrier log indicates that the implicit and explicit inputs to `[1].repOK()` and `[5].repOK()` have not changed.

Furthermore, Ditto optimistically assumes that the result of checking

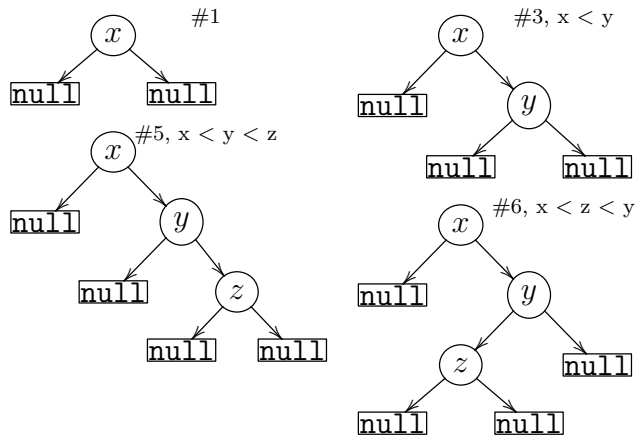
(a) input for remove(7)



(b) faulty output of remove(7)



(c) patches generated with SymboLazyDP



(d) repair result of remove(7)

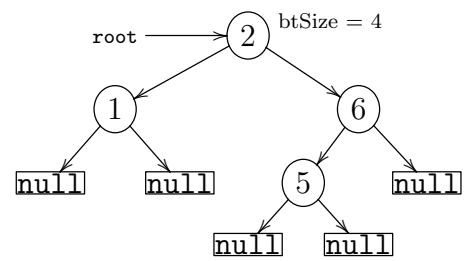


Figure 6.16: Patching structures to repair the faulty output of bug cycle.

the invariant on substructures remains the same as before, unless proved otherwise. In case the optimistic assumption turns out wrong, Ditto has to propagate the result upward and re-calculate `repOK`. For example, when calling `repOK` on [6] in Figure 6.16 (b), Ditto optimistically assumes that `repOK` still holds on [5]. Had there been any changes logged by write barriers in `[5].right` or `[5].left`, the optimistic assumption of Ditto would still hold as long as the result of `repOK` on [5] remained true. The optimistic assumption, however, might turn out wrong, and hence Ditto might have to re-check `repOK` for all the places that incorrectly assumed a return result for a `repOK`.

6.5.1.1 Detecting Cycles

Ditto, as it currently is, does not support data structures with cycles. The reason is that optimistic assumption does not necessarily work with cycles. Consider the execution of `[6].repOK()`, which checks acyclicity, in Figure 6.16 (b). Ditto optimistically assumes that both [5] and [2] still return true for the `repOK` method. However, [2] makes the same assumption about [7] and [7] in turn makes the same assumption about [6]. The optimistic assumption of Ditto makes the cycle go undetected and a `repOK` that checks it returns true.

Our structure generation technique assumes acyclicity too. Our repair technique, however, should be able to *detect* cycles introduced as errors and repair them. Therefore, we improve Ditto to detect erroneous cyclic states. Utilizing the domain knowledge that any cycle in our recursive data structures

is the result of an error, we return `false` as soon as we detect a cycle by requiring the template shown in the `repOK` method of Listing 6.1 which first checks for cycles.

6.5.1.2 Supporting Post-Conditions

We further improve Ditto to check recursive post-conditions. Such post-conditions cannot be memoized, as they check properties that relate pre- and post-states and so Ditto's caching and optimistic assumption do not make sense for checking post-conditions. We record the pre-state and check the post-condition recursively on substructures. For example, in Figure 6.16 (b), the post-condition which checks that the element 7 is removed is calculated recursively. Despite checking `repOK`, recursively checking post-conditions is a heuristic, and does not work for all types of post-conditions, e.g., an `add` method.

6.5.2 Repairing Data Structures with Pre-Generated Patches

The previous step heuristically determined the *location of error* in the data structure as [2], [7], and [6]. To repair the error in the data structure, we use small structures pre-generated by dynamic programming with symbolic execution (Figure 6.5) as patches to replace the location of error. In this case, we need a patch whose root has a null left child in order to connect to [1]. Therefore, trees #0, #2, #4, #7, and #8 are inappropriate. Next, we in turn use each of trees #1, #3, #5, and #6 (Figure 6.16 (c)) as candidate

patches. We solve for the symbolic values saved on the candidate patch using the values from the location of error, replace the location of error with the candidate patch, and finally check the `repOK` and `postCondition`, until they both return true for a patch. Tree #3 with $x = 2$ and $y = 6$ returns true for both `repOK()` and `postCondition()` and repairs the data structure.

6.6 Summary

In this chapter, we presented a novel technique for exhaustive and random generation of test inputs for programs that operate on structurally complex tests, e.g., recursive data structures. Our key insight is to leverage the recursive structure of desired inputs and partition the problem of generating an input into several sub-problems of generating smaller inputs that exhibit the same structure, and to use dynamic programming to combine them. We used a lazy initialization strategy as well as symbolic execution to optimize the technique. We formally proved the correctness of our algorithm. Experimental results show that our technique provides more efficient and scalable generation of structurally complex tests for a variety of subject programs, compared to state-of-the-art test generation tools Pex and Korat. Furthermore, our technique found real bugs in a well-tested commercial application, Google Chrome.

While in this work we focused on the generation of recursive data structures for testing, our work paves the way for the development of novel techniques for data structure repair. Given the increasing use of constraint solving

technology in software verification, we believe the time is ripe for dynamic programming to make a significant impact on our ability to find more bugs faster and to deploy more reliable software.

Chapter 7

Conclusions

We presented contract-driven data structure repair, a novel approach for error recovery, which uses rich behavioral contracts to repair erroneous executions on-the-fly. We addressed challenges that arise in transmuting high level and inherently non-deterministic contracts to efficient and scalable implementations that repair errors. Our three fold insight seeks to improve the efficiency, scalability, effectiveness, and usability of repair.

Firstly, we take advantage of the history of program execution and current repair in history-aware data structure repair. We obtain program execution history through write and read barriers, which log writes and reads the program performs to its data structures. Moreover, we access the history of current repair attempts through unsatisfiable cores provided by SAT solvers we use. Experiments using our prototype implementation, Cobbler, demonstrate the potential of history-aware data structure repair in correcting the effect of errors efficiently without unnecessarily perturbing data structures. Cobbler found and fixed a previously undetected error in an open source software, ANTLR.

Secondly, we applied the idea of abstracting and reusing repair actions

in the context of repair with SAT solvers. Our prototype tool, DREAM, piggybacks on other repair frameworks to record, abstract, and reuse repair actions they take in the event of future errors. We further implemented a prototype tool named Arreh which translates contracts to Java checks and checks them through the Java Virtual Machine to avoid repetitive calls to the SAT solver for checking. Experimental evaluation of DREAM, in accordance with Cobbler as the underlying repair framework and Arreh as the checking tool show that repair abstractions offer significant performance improvement.

Thirdly, we observed the similarities between contract-driven data structure repair and another important problem in software engineering, namely constraint-based structure generation for testing. We presented a new technique for exhaustive and random generation of test inputs for programs that operate on complex data structures. By exploiting the recursive nature of common data structures through dynamic programming, as well as using lazy initialization and symbolic execution, we outperformed state-of-the-art test generation tools Pex and Korat. We tested two commercial web browsers, Google Chrome and Apple Safari, and found two known and one new bug in the production version of Chrome. Finally, we discussed future ideas on the unification of contract-driven data structure repair and constraint-based structure generation.

7.1 Final Thoughts

Using our repair techniques, a program can be enhanced with the ability to recover from bugs quickly and with minimum amount of perturbation. This ability can act as a quick workaround bug fix, which potentially provides useful insights that could be utilized by the user to perform localization and correction of the fault in code later on. Data structure repair, in turn, can help automated debugging.

While repair has various applications, it does not suit all types of software systems. For systems that cannot tolerate even slight divergences in the state of the program from the original behavior (e.g, financial systems), it is not advisable to use automatic repair routines unless complete contracts with all the required details are available.

Amortizing the cost of writing and maintaining contracts between testing and repair seems a promising avenue to make repair useful in real world settings. Another idea along the same lines is amortizing the cost of repair between multiple errors. Moreover, sometimes the cost of writing contracts can even be amortized over different programs. Our recent work [108] exploits this possibility in the context of testing mobile apps where there is a common expectation of how an app would behave when subjected to a specific event, such as device rotation, for automated test oracles.

The use of dynamic programming, a classic optimization algorithm, in our test generation technique showed how fundamental text book methods

can solve software engineering problems. We think that similar theoretically well-founded algorithms can address arising challenges in software engineering.

Finally, we believe contract-driven data structure repair holds much promise in improving our ability to correct errors in deployed software. When unified with solutions to the widespread problem of testing, repair becomes an even more appealing idea to make software systems more reliable.

Bibliography

- [1] ANTLR parser generator home page. <http://www.antlr.org>.
- [2] Binary search tree remove implementation. http://www.algolist.net/Data_structures/Binary_search_tree/Removal.
- [3] CSS 3D transforms, editor's draft 21 March 2011. <http://dev.w3.org/csswg/css3-3d-transforms>.
- [4] Ext2 fsck manual page. <http://e2fsprogs.sourceforge.net>.
- [5] Google Chrome. <http://www.google.com/googlebooks/chrome>.
- [6] Microsoft chkdsk manual page. <http://support.microsoft.com/kb/315265>.
- [7] Statcounter. <http://statcounter.com>.
- [8] Bassel Y. Al-Naffouri. MintEra: A testing environment for Java programs. Master's thesis, Massachusetts Institute of Technology, 2004.
- [9] Andrew W. Appel. Simple generational garbage collection and fast allocation. *Software: Practice and Experience*, 19(2):171–183, 1988.
- [10] Shay Artzi, Julian Dolby, Frank Tip, and Marco Pistoia. Directed test generation for effective fault localization. In *Proceedings of 19th In-*

ternational Symposium on Software Testing and Analysis, pages 49–60, Trento, Italy, 2010.

- [11] Clark Barrett and Sergey Berezin. CVC Lite: A new implementation of the cooperating validity checker. In *Proceedings of the 16th International Conference on Computer Aided Verification*, pages 515–518, Boston, MA, July 2004.
- [12] Stephen M Blackburn, Robin Garner, Chris Hoffmann, Asjad M Khan, Kathryn S McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z Guyer, Martin Hirzel, Antony Hosking, Maria Jump, Han Lee, J Eliot B Moss, Aashish Phansalkar, Darko Stefanovic, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiederemann. The DaCapo Benchmarks: Java Benchmarking Development and Analysis. In *Proceedings of Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 190–211, Portland, Oregon, 2006.
- [13] Stephen M. Blackburn and Antony L. Hosking. Barriers: friend or foe? In *Proceedings of the 4th International Symposium on Memory Management*, pages 143–151, New York , NY, 2004.
- [14] Stephen M Blackburn and Kathryn S McKinley. In or out? putting write barriers in their place. In *Proceedings of 3rd International Symposium on Memory Management*, pages 175–184, Berlin, Germany, 2002.

- [15] Chandrasekhar Boyapati, Sarfraz Khurshid, and Darko Marinov. Korat: Automated testing based on Java predicates. In *Proceedings of International Symposium on Software Testing and Analysis (ISSTA)*, pages 123–133, Rome, Italy, July 2002.
- [16] William R. Bush, Jonathan D. Pincus, and David J. Sielaff. A static analyzer for finding dynamic programming errors. *Software—Practice and Experience*, 30(7), 2000.
- [17] Cristian Cadar, Daniel Dunbar, and Dawson R Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *USENIX Symposium on Operating Systems Design and Implementation*, pages 209–224, San Diego, CA, 2008.
- [18] Cristian Cadar and Dawson Engler. Execution generated test cases: How to make systems code crash itself. In *Proceedings of 12th SPIN Workshop on Software Model Checking*, pages 2–23, San Francisco, CA, 2005.
- [19] Antonio Carzaniga, Alessandra Gorla, Andrea Mattavelli, Nicolo Perino, and Mauro Pezze. Automatic recovery from runtime failures. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 782–791, San Francisco, CA, 2013.
- [20] Antonio Carzaniga, Alessandra Gorla, Nicolò Perino, and Mauro Pezzè. Automatic workarounds for web applications. In *Proceedings of the 8th*

ACM SIGSOFT International Symposium on Foundations of Software Engineering, pages 237–246, Santa Fe, New Mexico, 2010.

- [21] Patrick J. Caudill and Allen Wirfs-Brock. A third generation Smalltalk-80 implementation. In *Proceedings of Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 119–130, Portland, Oregon, 1986.
- [22] Satish Chandra, Emina Torlak, Shaon Barman, and Rastislav Bodik. Angelic debugging. In *33rd International Conference on Software Engineering (ICSE)*, pages 121–130, Waikiki, Honolulu, Hawaii, May 2011.
- [23] Koen Claessen and John Hughes. QuickCheck: A lightweight tool for random testing of Haskell programs. In *Proceedings of Fifth ACM SIGPLAN International Conference on Functional Programming*, pages 268–279, Montreal, Canada, 2000.
- [24] Lori A. Clarke. A system to generate test data and symbolically execute programs. *IEEE Transactions on Software Engineering*, pages 215–222, September 1976.
- [25] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. The MIT Press, 1990.
- [26] Christoph Csallner, Yannis Smaragdakis, and Tao Xie. DSD-Crasher: A hybrid analysis tool for bug finding. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 17(2):8, 2008.

- [27] Brett Daniel, Danny Dig, Kely Garcia, and Darko Marinov. Automated testing of refactoring engines. In *Proceedings of the 6th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT symposium on the Foundations of Software Engineering*, pages 185–194, Dubrovnik, Croatia, 2007.
- [28] Leonardo de Moura and Nikolaj Bjorner. Z3: An efficient SMT solver. In *Proceedings of Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 337–340, Budapest, Hungary, 2008.
- [29] Brian Demsky. *Data Structure Repair Using Goal-Directed Reasoning*. PhD thesis, Massachusetts Institute of Technology, January 2006.
- [30] Brian Demsky and Martin Rinard. Automatic detection and repair of errors in data structures. In *Proceedings of Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 78–95, Anaheim, California, 2003.
- [31] Bassem Elkarablieh. *Assertion-based Repair of Complex Data Structures*. PhD thesis, University of Texas at Austin, 2009.
- [32] Bassem Elkarablieh, Ivan Garcia, Yuk Lai Suen, and Sarfraz Khurshid. Assertion-based repair of complex data structures. In *22nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 64–73, Atlanta, GA, 2007.

- [33] Bassem Elkarablieh and Sarfraz Khurshid. Juzi: A tool for repairing complex data structures. In *Proceedings of 30th International Conference on Software Engineering (ICSE)*, pages 855–858, Leipzig, Germany, May 2008. Research Demo Paper.
- [34] Bassem Elkarablieh, Sarfraz Khurshid, Duy Vu, and Kathryn S. McKinley. STARC: Static analysis for efficient repair of complex data. In *Proceedings of Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 387 – 404, Montreal, Quebec, Canada, October 2007.
- [35] Bassem Elkarablieh, Yehia Zayour, and Sarfraz Khurshid. Efficiently generating structurally complex inputs with thousands of objects. In *Proceedings of 21st European Conference on Object-Oriented Programming (ECOOP)*, pages 248–272, August 2007.
- [36] Justin E Forrester and Barton P Miller. An empirical study of the robustness of Windows NT applications using random testing. In *Proceedings of the 4th USENIX Windows System Symposium*, pages 59–68, San Diego, CA, 2000.
- [37] Milos Gligoric, Tihomir Gvero, Vilas Jagannath, Sarfraz Khurshid, Viktor Kuncak, and Darko Marinov. Test generation through programming in UDITA. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*, pages 225–234, Cape Town, South Africa, 2010.

- [38] Milos Gligoric, Tihomir Gvero, Steven Lauterburg, Darko Marinov, and Sarfraz Khurshid. Optimizing generation of object graphs in Java PathFinder. In *International Conference on Software Testing Verification and Validation (ICST)*, pages 51–60, Denver, Colorado, 2009.
- [39] Patrice Godefroid. Compositional dynamic test generation. In *Proceedings of 34th annual ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages*, pages 47–54, Nice, France, 2007.
- [40] Patrice Godefroid, Nils Klarlund, and Koushik Sen. DART: directed automated random testing. In *Programming Language Design and Implementation (PLDI)*, pages 213–223, Chicago, Illinois, 2005.
- [41] Patrice Godefroid, Michael Y Levin, and David A Molnar. Automated whitebox fuzz testing. In *Proceedings of Network and Distributed Systems Security*, pages 151–166, San Diego, CA, 2008.
- [42] John B. Goodenough and Susan L. Gerhart. Toward a theory of test data selection. *IEEE Transactions on Software Engineering*, SE-1(2):156–173, 1975.
- [43] Divya Gopinath, Razieh Nokhbeh Zaeem, and Sarfraz Khurshid. Improving the effectiveness of spectra-based fault localization using specifications. In *Proceedings of the 27th International Conference on Automated Software Engineering*, pages 40–49, Essen, Germany, 2012.

- [44] Arnaud Gotlieb, Bernard Botella, and Michel Rueher. Automatic test data generation using constraint solving techniques. In *Proceedings of International Symposium on Software Testing and Analysis (ISSTA)*, pages 53–62, Clearwater Beach, FL, 1998.
- [45] G Haugk, FM Lax, RD Royer, and JR Williams. The 5ESS(TM) switching system: Maintenance capabilities. *AT&T Technical Journal*, 64(6 part 2):1385–1416, 1985.
- [46] Klaus Havelund and Grigore Roşu. Monitoring java programs with java pathexplorer. *Electronic Notes in Theoretical Computer Science*, 55(2):200–217, 2001.
- [47] Klaus Havelund and Grigore Roşu. Synthesizing monitors for safety properties. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 257–268, Grenoble, France, 2002.
- [48] Hans-Martin Horcher. Improving software tests using Z specifications. In *Proceedings of 9th International Conference of Z Users, The Z Formal Specification Notation*, pages 152–166, 1995.
- [49] J. C. Huang. An approach to program testing. *ACM Computing Surveys*, 7(3):113–128, 1975.
- [50] Ishtiaque Hussain and Christoph Csallner. Dynamic symbolic data structure repair. In *Proceedings of 32th International Conference on Software Engineering (ICSE)*, pages 215–218, Cape Town, South Africa, 2010.

- [51] Daniel Jackson. *Software Abstractions: Logic, Language, and Analysis*. The MIT Press, 2006.
- [52] Daniel Jackson, Ian Schechter, and Ilya Shlyakhter. ALCOA: The Alloy constraint analyzer. In *Proceedings of 22nd International Conference on Software Engineering (ICSE)*, pages 730–733, Limerick, Ireland, June 2000.
- [53] Dennis Jeffrey, Min Feng, Neelam Gupta, and Rajiv Gupta. BugFix: a learning-based tool to assist developers in fixing bugs. In *Proceedings of 17th International Conference on Program Comprehension*, pages 70–79, Vancouver, BC, Canada, 2009.
- [54] Shadi Abdul Khalek and Sarfraz Khurshid. Systematic testing of database engines using a relational constraint solver. In *4th International Conference on Software Testing, Verification and Validation (ICST)*, pages 50–59, Berlin, Germany, 2011.
- [55] Sarfraz Khurshid, Iván García, and Yuk Lai Suen. Repairing structurally complex data. In *12th SPIN Workshop on Model Checking of Software*, pages 123–138, San Francisco, CA, August 2005.
- [56] Sarfraz Khurshid, Corina Pasareanu, and Willem Visser. Generalized symbolic execution for model checking and testing. In *Proceedings of 9th Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS)*, pages 553–568, Warsaw, Poland, April 2003.

- [57] Dongsun Kim, Jaechang Nam, Jaewoo Song, and Sunghun Kim. Automatic patch generation learned from human-written patches. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 802–811, San Francisco, CA, 2013.
- [58] James C. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976.
- [59] Pieter Koopman, Artem Alimarine, Jan Tretmans, and Rinus Plasmeijer. Gast: Generic automated software testing. *Implementation of Functional Languages*, pages 991–991, 2003.
- [60] Bogdan Korel. Automated test data generation for programs with procedures. In *Proceedings of International Symposium on Software Testing and Analysis (ISSTA)*, pages 209–215, San Diego, CA, 1996.
- [61] Ralf Lämmel and Wolfram Schulte. Controllable combinatorial coverage in grammar-based testing. In *Testing of Communicating Systems*, pages 19–38, New York, NY, 2006.
- [62] Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. Genprog: A generic method for automatic software repair. *IEEE Transactions on Software Engineering*, 38(1):54–72, 2012.
- [63] Henry Lieberman and Carl Hewitt. A real-time garbage collector based on the lifetimes of objects. *Communications of the ACM*, 26(6):419–429, 1983.

- [64] Barbara Liskov and John Guttag. *Program Development in Java: Abstraction, Specification, and Object-Oriented Design*. Addison-Wesley, 2000.
- [65] Muhammad Zubair Malik. Combining data structure repair and program repair. PhD Thesis Proposal, University of Texas at Austin, Austin, TX, 2013.
- [66] Muhammad Zubair Malik, Khalid Ghori, Bassem Elkarablieh, and Sarfraz Khurshid. A case for automated debugging using data structure repair. In *Proceedings of 24th Conference on Automated Software Engineering (ASE)*, pages 620–624, Auckland, New Zealand, 2009.
- [67] Darko Marinov and Sarfraz Khurshid. TestEra: A novel framework for automated testing of Java programs. In *Proceedings of 16th Conference on Automated Software Engineering (ASE)*, pages 22–31, San Diego, CA, November 2001.
- [68] Peter M. Maurer. Generating test data with enhanced context-free grammars. *IEEE Software*, 7(4):50–55, 1990.
- [69] Wolfgang Mayer and Marcus Stumptner. Evaluating models for Model-Based debugging. In *Proceedings of 23th Conference on Automated Software Engineering (ASE)*, pages 128–137, L’Aquila, Italy, 2008.
- [70] William M McKeeman. Differential testing for software. *Digital Technical Journal*, 10(1):100–107, 1998.

- [71] Bertrand Meyer. Applying “design by contract”. *IEEE Computer*, 25(10):40–51, 1992.
- [72] Bertrand Meyer, Ilinca Ciupa, Andreas Leitner, and Lisa Ling Liu. Automatic testing of object-oriented software. In *Current Trends in Theory and Practice of Computer Science (SOFSEM)*, pages 114–129, Harrachov, Czech Republic, 2007.
- [73] Bertrand Meyer, Arno Fiva, Ilinca Ciupa, Andreas Leitner, Yi Wei, and Emmanuel Stapf. Programs that test themselves. *Computer*, 42(9):46–55, 2009.
- [74] Aleksandar Milicevic, Sasa Misailovic, Darko Marinov, and Sarfraz Khurshid. Korat: A tool for generating structurally complex test inputs. In *Proceedings of 29th International Conference on Software Engineering (ICSE)*, pages 771–774, Minneapolis, MN, May 2007.
- [75] Samiha Mourad and Dorothy Andrews. On the reliability of the IBM MVS/XA operating system. *IEEE Transactions on Software Engineering*, 13(10):1135–1139, 1987.
- [76] Hoang Duong Thien Nguyen, Dawei Qi, Abhik Roychoudhury, and Satish Chandra. Semfix: Program repair via semantic analysis. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 772–781, San Francisco, CA, 2013.

- [77] Jeff Offutt and Aynur Abdurazik. Generating tests from UML specifications. In *Proceedings of Second International Conference on the Unified Modeling Language*, pages 416–429, Fort Collins, CO, October 1999.
- [78] Carlos Pacheco, Shuvendu K Lahiri, Michael D Ernst, and Thomas Ball. Feedback-directed random test generation. In *29th International Conference on Software Engineering*, pages 75–84, Minneapolis, MN, 2007.
- [79] Jeff H. Perkins, Sunghun Kim, Sam Larsen, Saman Amarasinghe, Jonathan Bachrach, Michael Carbin, Carlos Pacheco, Frank Sherwood, Stelios Sidiroglou, Greg Sullivan, Weng-Fai Wong, Yoav Zibin, Michael D. Ernst, and Martin Rinard. Automatically patching errors in deployed software. In *Proceedings of 27th ACM Symposium on Operating Systems Principles (SOSP)*, pages 87–102, Big Sky, MT, 2009.
- [80] Chittoor V Ramamoorthy, Siu-Bun F Ho, and WT Chen. On the automated generation of program test data. *IEEE Transactions on Software Engineering*, 2(4):293–300, 1976.
- [81] Michael Roberson and Chandrasekhar Boyapati. Efficient modular glass box software model checking. In *Proceedings of the 25th Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 4–21, Reno/Tahoe, Nevada, 2010.
- [82] Hesam Samimi, Ei Darli Aung, and Todd Millstein. Falling Back on Executable Specifications. In *Proceedings of 24th European Conference*

- on *Object-Oriented Programming (ECOOP)*, pages 552–576, Maribor, Slovenia, EU, May 2010.
- [83] Hesam Samimi, Max Schäfer, Shay Artzi, Todd Millstein, Frank Tip, and Laurie Hendren. Automated repair of HTML generation errors in php applications using string constraint solving. In *Proceedings of the 34th International Conference on Software Engineering (ICSE)*, pages 277–287, Zurich, Switzerland, 2012.
- [84] Alberto Sanfeliu and King-Sun Fu. Distance measure between attributed relational graphs for pattern recognition. *Systems, Man and Cybernetics, IEEE Transactions on*, 13(3):353–362, 1983.
- [85] Koushik Sen, Darko Marinov, and Gul Agha. CUTE: A concolic unit testing engine for C. In *Proceedings of European Software Engineering Conference and ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE)*, pages 263–272, Lisbon, Portugal, 2005.
- [86] Ajeet Shankar and Rastislav Bodik. DITTO: automatic incrementalization of data structure invariant checks (in Java). In *Proceedings of ACM SIGPLAN’07 Conference on Programming Language Design and Implementation (PLDI)*, pages 310–319, San Diego, California, 2007.
- [87] Junaid Haroon Siddiqui and Sarfraz Khurshid. An empirical study of structural constraint solving techniques. In *ternational Conference on*

- Formal Engineering Methods (ICFEM)*, pages 88–106, Rio de Janeiro, Brazil, 2009.
- [88] Emin Gün Sirer and Brian N. Bershad. Using production grammars in software testing. In *Proceedings of 2nd conference on Domain-specific languages*, pages 1–13, 1999.
- [89] Matt Staats and Corina Pasareanu. Parallel symbolic execution for structural test generation. In *Proceedings of the 19th International Symposium on Software Testing and Analysis*, pages 183–194, Trento, Italy, 2010.
- [90] Stefan Staber, Barbara Jobstmann, and Roderick Bloem. Finding and fixing faults. In *13th IFIP WG 10.5 Advanced Research Working Conference on Correct Hardware Design and Verification Methods*, pages 35–49, Saarbrücken, Germany, 2005.
- [91] Yuk Lai Suen. Automatically repairing structurally complex data. Master’s thesis, Department of Electrical and Computer Engineering, University of Texas at Austin, May 2005.
- [92] Nikolai Tillmann and Jonathan De Halleux. Pex–white box test generation for .NET. *Tests and Proofs*, 4966:134–153, 2008.
- [93] Emina Torlak and Daniel Jackson. Kodkod: A relational model finder. In *Proceedings of 13th Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS)*, pages 632–647, Braga, Portugal, March 2007.

- [94] Martin T Vechev and David F Bacon. Write barrier elision for concurrent garbage collectors. In *In Proceedings of 4th International Symposium on Memory Management*, pages 13–24, Vancouver, British Columbia, 2004.
- [95] Willem Visser, Klaus Havelund, Guillaume Brat, SeungJoon Park, and Flavio Lerda. Model checking programs. In *18th International Conference on Automated Software Engineering (ASE)*, pages 203–232, 2003.
- [96] Christian Von Essen and Barbara Jobstmann. Program repair without regret. In *25th International Conference on Computer Aided Verification*, pages 896–911, Saint Petersburg, Russia, 2013.
- [97] Yi Wei, Yu Pei, Carlo A. Furia, Lucas S. Silva, Stefan Buchholz, Bertrand Meyer, and Andreas Zeller. Automated fixing of programs with contracts. In *Proceedings of International Symposium on Software Testing and Analysis (ISSTA)*, pages 61–72, Trento, Italy, 2010.
- [98] Westley Weimer. Patches as better bug reports. In *Proceedings of 5th international conference on Generative programming and component engineering*, pages 181–190, Portland, Oregon, 2006.
- [99] Tao Xie, Darko Marinov, Wolfram Schulte, and David Notkin. Symstra: A framework for generating object-oriented unit tests using symbolic execution. In *Proceedings of 11th Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS)*, pages 365–381, Edinburgh, UK, April 2005.

- [100] Guowei Yang, Corina S Păsăreanu, and Sarfraz Khurshid. Memoized symbolic execution. In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 144–154, Minneapolis, MN, 2012.
- [101] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and understanding bugs in C compilers. In *32nd ACM SIGPLAN conference on Programming Language Design and Implementation (PLDI)*, pages 283–294, San Jose, California, 2011.
- [102] Razieh Nokhbeh Zaeem. Contract-based data structure repair using Alloy. Master’s thesis, Department of Electrical and Computer Engineering, University of Texas at Austin, May 2010.
- [103] Razieh Nokhbeh Zaeem, Divya Gopinath, Sarfraz Khurshid, and Kathryn S McKinley. History-aware data structure repair using SAT. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 2–17, Tallinn, Estonia, March 2012.
- [104] Razieh Nokhbeh Zaeem and Sarfraz Khurshid. Contract-Based Data Structure Repair Using Alloy. In *Proceedings of 24th European Conference on Object-Oriented Programming (ECOOP)*, pages 577–598, Maribor, Slovenia, EU, May 2010.
- [105] Razieh Nokhbeh Zaeem and Sarfraz Khurshid. Introducing Specification-Based Data Structure Repair Using Alloy. In *Proceedings of International Conference on ASM Alloy B and Z*, pages 398–399, Orford, Quebec, Canada, February 2010.

- [106] Razieh Nokhbeh Zaeem and Sarfraz Khurshid. Test input generation using dynamic programming. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, number 34, Cary, North Carolina, 2012. 11 pages.
- [107] Razieh Nokhbeh Zaeem, Muhammad Zubair Malik, and Sarfraz Khurshid. Repair abstractions for more efficient data structure repair. In *Fourth International Conference on Runtime Verification*, pages 235–250, INRIA Rennes, France, 2013.
- [108] Razieh Nokhbeh Zaeem, Mukul R. Prasad, and Sarfraz Khurshid. Automated generation of oracles for testing user-interaction features of mobile apps. In *7th International Conference on Software Testing, Verification and Validation*, pages 183–192, Cleveland, OH, 2014.

Vita

Razieh Nokhbeh Zaeem received the Bachelor of Science degree in Computer Engineering from Sharif University of Technology, Tehran, Iran in September 2006. In May 2010, she received the Master of Science and Engineering degree in Electrical and Computer Engineering from the University of Texas at Austin and was honored as a 2010 Google Anita Borg Scholarship Finalist. She interned at Rockwell Automation Inc. in Austin, TX in Summer 2010, and at Fujitsu Laboratories of America in Sunnyvale, CA in Summer and Fall 2012. She received her PhD in Electrical and Computer Engineering from the University of Texas at Austin in May 2014.

Permanent address: nokhbeh@gmail.com

This dissertation was typeset with L^AT_EX[†] by the author.

[†]L^AT_EX is a document preparation system developed by Leslie Lamport as a special version of Donald Knuth's T_EX Program.