The Dissertation Committee for Bassem H. Elkarablieh
certifies that this is the approved version of the following dissertation:

Assertion-based Repair of Complex Data Structures

Committee:

Sarfraz Khurshid, Supervisor

Adnan Aziz

Dewayne Perry

Kathryn McKinley

Keshav Pingali

Assertion-based Repair of Complex Data Structures

by

Bassem H. Elkarablieh, B.S., M.S.

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 2009

Dedicated to my family and friends.

Acknowledgments

I am deeply indebted to many people for their guidance and support during the three years I spent at the University of Texas at Austin.

I would first like to thank my advisor, Dr. Sarfraz Khurshid, for his continuous support, spectacular training and everlasting professional and technical advice. I will never forget him staying in office till 5 a.m. assisting me with writing my first paper. I feel extremely lucky to have him as my advisor and I wish him all the success in his life and career.

I am grateful to Dr. Kathryn McKinley. Kathryn was the instructor for my first lecture class at UT. Kathryn unselfishly provided me her guidance and assistance. Her comments and advice vastly improved the quality of my work.

I would like to thank the readers of my thesis. Dr. Adnan Aziz, Dr. Dwayne Perry and Dr. Kishav Pengali for their helpful comments on my work.

Special acknowledgements go to the Electrical and Engineering department, the Excellence in Distributed Global Environment center, and the National Science Foundation for providing me with the financial assistance throughout my research years at UT.

My heartfelt gratitude also goes to Dr. Adrian Nunez, my thesis advisor at Syracuse University, who currently works at Intel. Adrian aggressively campaigned for my fellowship at Syracuse University. Without his help and support I would not have made it this far. All the best wishes for him and his family.

Throughout my academic career, I had three summer internships. Special thanks to my mentors Laurant Chawaout at Intel, John Thomas at Google, and Patrice Godefroid at Microsoft Research.

I am proud to be a member of the Software Verification Validation and Testing group at UT. I was fortunate to work with very smart researchers including, Engin Uzuncaova , Danhua Shao, Daryl Shannon, Alison Lee, Zubair Malik, Yehia Zayour, and Shadi Abdul Khalek. Those guys provided the friendly environment that I needed to complete my research.

I could not complete my Ph.D. without the support of my friends. I would like to first thank my friends at UT, Shayak Banerjee, Veronica (Chen) Ding, Chihwen (Wendy) Kan, Shadi Abdul Khalik, Monica Coury, Michele Saad, Hala Nasser, Marcel Nassar, Mohammed, Ali, Surayah, and Sarah Fakhreddine, Omar El Ayach, and Salam Akoum. Special thanks to my friends in Syracuse University, Mario Tayah, Nancy Khoury, Jean Hannoush, Tingwei (Benson) Chiang , WeiYu (Trenton) Chen, Phoenix Huang, Chinyu Chin, Tsu Mei (Amy) Wie, WonKyung Park, Gulru Ustundag for helping me survive my first two years of graduate school in the freezing cold of the north. I would also like to thank my high school and college friends in Lebanon, Sami Azzam, Nicolas Tohme, Mazen Rashidi, Mohammad Sabbagh, Lara Semaan, Farah Eido, Zena Sheikh, Mohammad Mahjoub, Talal Kurdi, Lama Wazzan, Rola Haidar, Dana Sleiman and Ali Jaber.

A great owe of gratitude goes to Christine Kehyayan for being patient and understanding all these year. While she never asked me or pushed me to finish, I always felt the pressure to prosper and make her proud.

Finally, my deepest thanks go to my family, my mother Wafaa, my uncles Mowafak and Najib, and my brother Feras. They never doubted my abilities, never asked me“when are you going to graduate?”, always trusted my decisions, and prayed for my success and prosperity. From all my heart, Thank you!

Assertion-based Repair of Complex Data Structures

Publication No. ___________________

Bassem H. Elkarablieh, Ph.D.
The University of Texas at Austin, 2009

Supervisor: Sarfraz Khurshid

As software systems are growing in complexity and size, reliability becomes a major concern. A large degree of industrial and academic efforts for increasing software reliability are directed towards design, testing and validation—activities performed *before* the software is deployed. While such activities are fundamental for achieving high levels of confidence in software systems, bugs still occur after deployment resulting in costly software failures. This dissertation presents *assertion-based repair*, a novel approach for error recovery from insidious bugs that occur *after* the system is deployed. It describes the design and implementation of a repair framework for Java programs and evaluates the efficiency and effectiveness of the approach on repairing data structure errors in both software libraries and open-source stand-alone applications.

Our approach introduces a new form of assertions, *assertAndRepair*, for developers to use when checking the consistency of the data structures manipulated by their programs with respect to a set of desired structural and data properties. The developer provides the properties in a Java boolean method, *repOk*, which returns a truth value based on whether a given data structure satisfies these properties. Upon an assertion violation due to a faulty structure, instead of terminating the execution, the structure is repaired, i.e., its fields are mutated such that the resulting structure satisfies the desired properties, and the program proceeds with its execution. To aid developers in detecting the causes of the fault, repair-logs are generated which provide useful information about the performed mutations.

The repair process is performed using a novel algorithm that uses a systematic search based on symbolic execution to determine valuations for the structures' fields that result in a valid structure. Our experiments on repairing both library data structures, as well as, stand-alone applications demonstrate the utility and efficiency of the approach in repairing large structures, enabling programs to recover from crippling errors and proceed with their executions.

Assertion-based repair presents a novel post-deployment mechanism that integrates with existing and newly developed software, providing them with the defensive ability to recover from unexpected runtime errors. Programmers already understand the advantages of using assertions and are comfortable with writing them. Providing new analyses and powerful extensions for them presents an attractive direction towards building more reliable software.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 An Increasing Demand for Reliability

As software systems are growing in complexity and size, reliability is becoming harder to achieve. Software failures already cost the US economy tens of billions of dollars annually; and such cost is expected to rapidly increase with the rising dependency of the majority of businesses on software [62, 119]. To meet the ever-increasing demand for reliability, a great deal of progress is required in improving the current state-of-art and developing new techniques for delivering high quality, robust software.

Efforts for improving reliability in industry and academia are primarily directed toward design, testing, and validation [4, 21, 24, 95]—activities that are performed before the software is deployed. While such *pre-deployment* activities are fundamental for providing a certain level of confidence in program correctness and robustness, they do not prevent errors and anomalies from occurring dynamically in deployed software. For example, a single bit flip due to a cosmic ray can compromise the safety of a Java Virtual Machine with high probability and allow an intruder to run arbitrary code [52].

Bugs are inherent in deployed software systems resulting in errors which, if left untreated, can have serious consequences such as security breaches, loss of critical data, or unexpected failures. With error prevention being an arduous task, *post-deployment* techniques and approaches become necessary for systems to detect and adapt to unwanted erroneous scenarios. The ability to handle runtime errors on-the-fly can bring an unprecedented increase on reliability. Such increase is necessary to enable running mission-critical systems in unpredictable and hostile environments.

1.2 Improving Reliability by Utilizing Assertions

A powerful technique for detecting runtime errors in a program state is by using *assertions*—statements that evaluate a boolean expression representing a set of desired properties. Programmers have long used assertions to describe program properties, and most recent programming languages including C++, Java, C#, Ruby and Python have special support for assertions. An assertion violation in a running program is highly likely to indicate that the program has reached an erroneous state.

Usually, when an error occurs in a running application, programmers terminate the application, debug, test, and redeploy it. While this halt-on-error approach is sometimes necessary, e.g., during the execution of a security protocol, there are situations where alternative approaches are more desirable. For example, with corruption of persistent data, such as a file system, a simple reboot is unlikely to help. As another example, consider an intentional naming server for service location in a dynamic network [1]. If the server fails due to a malformed query, a continual subjection to the query will force perpetual failures. This problem compounds for deployed software, which cannot be promptly debugged and re-installed.

An attractive alternative to halt-on-error is to repair the state of the program and let it continue. In several cases, this alternative enables systems to resume their correct behavior. For example, a server that does not crash on a malformed query but repairs it, can continue to correctly resolve well-formed ones. Similarly, repairing a file system or a database can recover valuable data.

The traditional techniques for repair implement special repair routines [55, 93] which are triggered when specific problems occur during execution. These routines are domain specific and do not perform repair based on a well-formed description of the system. Thus, it is hard to build a robust generic repair framework using such approaches, since the developer must envision all possible bugs.

In this dissertation, we introduce *assertion-based repair*, a novel approach for error recovery from insidious bugs that occur after the system is deployed. We argue that assertions in a program represent a powerful form of specifications that hold key information for repairing corrupt program states and increasing program tolerance to runtime errors.

1.3 Assertion-based Error Recovery

Assertions have long been used to describe the properties of code. By writing assertions in a program, programmers intend to check the consistency of the program state with respect to a set of desired properties. Several static and dynamic analyses utilize assertions for checking programs [7, 22, 59, 113, 124]. While proving to be highly useful for error detection, we envision assertions to be likely powerful for error recovery.

The techniques we propose in *assertion-based repair* utilize violated assertions and use them as a basis for repairing corrupt program state. The key insight is that assertions include the properties that the program must satisfy to proceed with its execution. We utilize this insight as a basis to mutate the program state to satisfy these properties. These mutations not only serve as a mean to repair the state, but also, provide helpful information for developers to diagnose the causes of the inconsistencies.

We focus on repairing structurally complex data, which pervade object-oriented languages and are characterized by *class invariants* that represent structural integrity constraints. Good programming practice advocates the use of class invariant in assertions, by writing the invariant as predicates, named `repOk`, which return `true` if and only if their input satisfies its constraints [81]. The key idea is to repair data structures by analyzing the structural constraints described in the violated assertion. Given a structure that violates an assertion which represents its integrity constraints, we alter the structure such that the resulting structure satisfies the given constraints. Assertion-based repair does not aim to transform an erroneous state into one that a (hypothetical) correct program would have generated [31] because such state is unknown at the time of the violation. Instead, repair aims to generate a state that allows the program to recover on-the-fly from an erroneous state and resume its normal operation.

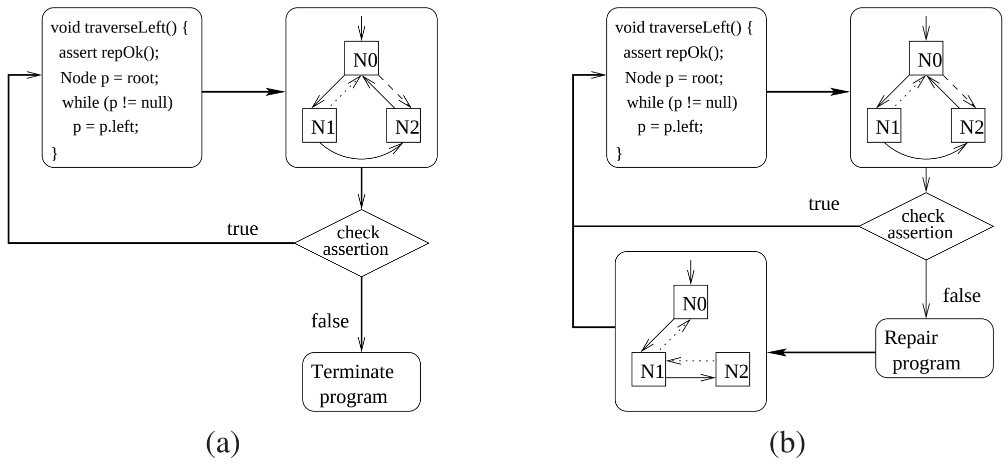


Figure 1.1: An illustration of assertion-based repair.

1.3.1 Illustrative Example

To elaborate an overall view of assertion-based repair, consider the example in Figure 1.1. The tree structure in Figure 1.1 contains a cycle along the `left` pointer, and thus violates the acyclicity constraint of a tree. The `traverseLeft` method performs a simple traversal of the tree following the `left` pointer. Before traversing the tree, the `traverseLeft` checks the validity of the structure by asserting `repOk`. It is always a good idea to assure the validity of the state before performing critical operations. Had the `assert` statement not been executed, the program would have gone into an infinite loop. However, when the assertion is executed on the corrupt structure the violation is detected.

Figure 1.1 (a) shows the traditional approach for handling assertion violations. Upon detecting an error through an assertion violation the program is terminated, debugged and re-executed. Figure 1.1 (b) shows the repair-based approach for handling assertion violations. Instead of terminating the program, the structure is repaired, the cycle along the `left` pointer is broken, and the program execution proceeds safely. Note that assertion-based repair utilizes the existence of the assertion check to perform repair. After repair, the program continues its execution from the statement right after the assertion check.

1.3.2 Why Assertion-based Repair?

Several features and properties make assertion-based repair a very attractive post-deployment analysis that can have a profound impact on improving software reliability.

Unlike hand written dedicated repair routines [55, 93] which describe *how* to generate a desired state, assertions describe *what* a desired state should be, i.e., its properties. Writing the repair routine already requires knowledge of desired properties. Moreover, it requires translating them into a procedure that correctly establishes them. To illustrate, consider red-black trees. Writing a repair routine involves implementing complex re-balancing operations to satisfy the constraints on height, color, etc. In contrast, writing an assertion requires writing conceptually simple tree traversals that *check* the constraints.

Unlike previous work on constraint based repair, assertion-based repair allows writing constraints using the language of implementation. Prior work [31–33] requires describing the properties in a declarative language. Although declarative languages provide a more succinct method for describing constraints, there is a large gap between the syntax and grammar of such languages and those of imperative programming languages which are commonly used by software developers and testers.

Assertion-based repair is a feasible candidate for integrating error recovery logic in software systems. Repair is only triggered on assertion violations and thus the repair logic incurs no extra cost on a running program. The only cost imposed by repair is the cost of writing assertions. Several applications already contain assertions, as is advocated by defensive programming [51]. In such cases, repair comes for free.

1.3.3 Applicability of Repair

The applicability of assertion-based repair on software applications highly depends on the properties that characterize these applications.

Applications that maintain persistent data are attractive candidates for repair; for example a file system or a database application where rebooting the system is highly unlikely to repair a fault. For another example, a data application where clearing and rebuilding the state of a database may take much more time than fixing a fault that occurs in the data structure at run-time.

Service oriented applications with very high downtime costs are other good candidates for repair. A widely used technique for increasing the reliability of such applications is by asserting the inputs, and ignoring any input that violates the application contract. Ignored inputs represent a loss of valuable transactions. Rather than ignoring such transactions, repair fixes the corrupt inputs to satisfy the given contract and enables processing them.

One property of repair is that the repaired structure might not be the one a program would have generated. These differences might further affect the execution of the program making it hard to reason about the acceptability of the repaired application. Some applications can tolerate such differences. For example in some application, like games, a deterioration in gameplay is much more preferable by users than a crash that terminates the game before even saving the progress. In some applications, data structures are mutually independent and not all the structures are affected by the changes. In other applications, changes in the data structure might only affect the performance of the program, for instance, consider a search application that stores information in a binary search tree. If the tree gets corrupted, it might be repaired into a linked list. Such a change might affect the performance of the search but not the correctness.

On the other hand, applications that consider exactness a more important requirement than durability might not be appropriate for repair and program termination might be their best solution. For example, consider a stock exchange application where a stock price is automatically changed by repair. Such result might create hazardous consequences on the users. Computational algorithms are other candidates that may not tolerate even a minimal deviation in data and thus may not be appropriate for repair.

It is therefore important to consider the risks, costs, and benefits before choosing whether repair is appropriate for an application. Our experience with repairing stand-alone applications including a program analysis tools, a calendar organizer and a database engine showed that repairing the corruptions enabled the applications to proceed with their executions in a manner that is more appropriate for the user than sudden crashes.

In summary, applications where uptime is important, reboot is a non-feasible solution, and which tolerate the variance between the repaired structure and the hypothetically correct one are targets for assertion-based repair. Applications where rebooting solves the problem, that are data sensitive, and cannot tolerate changes in the program state are not appropriate candidates for repair.

1.4 Thesis and Challenges

Our thesis is that assertion-based repair is a practical post-deployment technique for improving reliability through error recovery from data structure corruption errors: it is feasible to on-the-fly repair fairly large complex data structures while enabling applications to proceed with their executions, and providing developers with useful information to diagnose the error sources. This dissertation provides the evidence that supports our thesis. It presents an efficient and effective algorithm for repairing data structures based on violated assertions and describes the design and implementation of a framework for repairing Java programs.

We faced several challenges while developing this work. One of the challenges was the design of the repair algorithm. As a post-deployment analysis, the repair algorithm must be orders of magnitude faster than similar assertion-based testing and verification techniques [10, 29, 34, 91, 101]. At the same time, correctness is crucial for data structure repair; a repair algorithm that generates structures that do not satisfy the desired properties is not practical.

The main challenge that we faced in this work was evaluating the effect of repair on running programs. Recall that when repair is performed, there is no guarantee that the repaired structure would be the same as a hypothetically correct structure. While the repaired structure allows the program to proceed with its execution, there is no precise way to predict how the program behaves on the repaired structure, or evaluate how far does the repaired execution diverge from the original one.

The methodology we therefore use to evaluate the effect of repair is by measuring the acceptability of the repaired program from a user perspective based on empirical studies. We studied the behavior of a set of open-source applications after being repaired. The repaired faults were existing bugs that we found documented in bug repositories.

The case studies showed that in some applications where the data structures employ a certain degree of redundancy, the applications were completely repaired and the effect of repair was not noticeable. In some other cases the effect of repair was a modification in the order of the data elements. Nevertheless, in all the considered applications, repair enabled the program to avoid a potential crash.

1.5 Contributions

This dissertation makes the following contributions:

- **Repair assertions:** We present an alternative use of program assertions. Assertions already provide powerful tools for runtime checking. Repair assertions provide an effective and practical tool for error recovery from data structure errors. A convenient form of writing assertions is to use imperative predicates such as `repOk` methods that describe class invariants. We introduce the `assertAndRepair` construct for Java that allows both runtime checking of properties described as `repOk` methods as well as automatic repair in case of a violation.

- **Algorithm for efficient and effective repair:** We present a novel algorithm that uses violated assertions as a basis for repair. The repair algorithm employs a *systematic backtracking search* in conjunction with *on-demand symbolic execution* to mutate and fix corruptions in the data structure's fields. Our algorithm is sound and complete with respect to the given structural constraints under realistic assumptions. It repairs a corrupt structure into one that satisfies the given constraints if a valid structure can be constructed from the objects of the corrupt one and it does not generate a structure that violates any of the given constraints.

- **Optimizations:** We present two optimizations to scale the performance of the repair algorithm. The first optimization is based on a static analysis that guides the repair algorithm to the more likely candidates to repair corrupt fields. The second optimization is based on an efficient state exploration engine for repair that performs checkpoint-based backtracking by storing partial program states and performing *abstract undo operations*. These optimizations enable the repair algorithm to handle some structures with hundreds of thousands of objects.

- **Implementation:** We implemented a framework for assertion-based repair of Java programs. Our framework comprises of three main components: (1) A Java API to be used in programs to enable repair; (2) the core repair logic which consists of the search engine, the symbolic execution engine, and the constraint solver; and (3) a configuration module that enables user control to the repair algorithm.

- **Repair logs:** We abstract repair to report a log which summarizes the repair actions. We also allow custom abstraction functions provided by the user to compare the state before and after repair. These abstractions help the user understand the mutations performed during repair and assist in diagnosing the sources of the errors.

- **Evaluation:** We present an evaluation of assertion-based repair on both text-book data structures as well as stand-alone applications. We evaluate the efficiency of the repair algorithm by using it to repair fairly large data structures characterized with a variety of structural and data constraints. We evaluate the acceptability of the repaired structures by performing a set of case studies on repairing stand-alone applications. The experimental results show the feasibility of repairing some data structures with hundreds of thousands of nodes while enabling programs to safely proceed with their executions.

- **Application:** We present an application of repair to automated constraint based testing. We use the repair framework to generate large data structures by repairing randomly generated graphs to satisfy a set of desired properties.

Such structures are important for performing several testing tasks, including stress and load testing.

1.5.1 Impact

Data structures are ubiquitous in software systems. Programmers typically use library data structures or implement custom data structures that are specific for their programs. Recent work by Jump and McKinley [65] on dynamic shape analysis showed that 90% of the heap objects when running the SPECjvm [27] and DaCapo [9] benchmarks were part of recursive data structures; out of which 33% were application specific data structures. Assertion-based repair makes a fundamental impact on increasing the confidence in the validity of the data structures maintained by programs. Recursive data structures often include redundancies to enable performing efficient operations. Assertion-based repair enables repairing corrupt data structures while utilizing redundancies to perform effective and efficient repair that in most cases results in unnoticeable effect on running programs.

Developing robust software requires developing methodologies and paradigms that apply to the different levels of the software life cycle [10,38,66,84]. A large degree of effort on increasing reliability has focused on approaches for better requirement gathering, architecture, design, and testing. One key issue in these approaches resides in the lack of synergy among the used methodologies and the expensive cost of combining these methodologies. Another issue is that these approaches are applied before the software is deployed and do not provide software with a defense against errors that occur during runtime.

A key impact of assertion-based repair is its ability to integrate with existing approaches that utilize assertions for checking software before it is deployed [10], while providing software with the ability to tolerate errors and programmers with the ability to diagnose these errors. Assertion-based repair enables a unified framework for software verification and *resilient computing*—two software reliability methodologies that traditionally have employed very different algorithms. For example, using Korat [10,89,90], a constraint based testing framework, in conjunction with repair, a program annotated with assertions is (1) systematically tested before deployment and (2) trusted to execute without corruption once deployed—using the very same assertions. The unification has the potential to make a profound impact on improving the quality of software by providing software verification and resilient computing together at the cost of one.

A broader impact of this work is that it demonstrates the feasibility of performing post-deployment analyses without incurring large performance cost. It opens a promising direction for future work on applying a wide-spectrum of analyses on deployed software in an effort to improve reliability. Other researchers at academia [104] and industry [53] also showed interest in repair and are now are considering similar approaches of repair in their work. New software development paradigms are also being developed to enable automatic integration of repair logic in software [11].

Chapter 2

Overview

This chapter presents an overview of assertion-based repair, its components and applications. We start by listing the key requirements that we believe post-deployment analyses as well as error recovery techniques must satisfy to be attractive for practitioners; we then briefly overview the techniques we developed to target these requirements.

2.1 Requirements

The goal of our work is to provide programmers with a set of tools and techniques to easily incorporate data structure repair into their programs as an error recovery mechanism to improve reliability. We define our goal through the following requirements.

- *Correctness*: The repair logic must be sound and complete, i.e., it must not repair a corrupt data structure into one that does not satisfy the desired constraint; at the same time if a structure exists within given domains that satisfies the constraints, the repair logic must generate it.
- *Efficiency*: Being a post-deployment analysis, repair must be orders of magnitude faster than similar assertion-based techniques for testing, verification,

and error recovery. Since repair is only performed upon assertion violations, it must add minimal overhead on running applications, and must not deteriorate performance in case the repair logic is not executed.

- *Ease of integration*: Repair must be easy to incorporate in existing software systems. The developer need not expend large amount of effort to integrate repair into a software system. Rather, by following good programming practices such as writing program specifications and defensive code that frequently asserts its key properties, repair must come at minimal cost.

- *Transparency*: Repair must not keep the developers in the dark as to what is being changed in the program state. It must be *configurable*, giving the developer some control over the subset of the program state that repair alters.

With these requirements in mind, we first developed an effective algorithm to repair corrupt data structures. We then developed optimizations for the original algorithm in order to improve its efficiency. The efficiency of these optimizations directed our attention to alternative uses of repair in test case generation of large data structures. We finally implemented a configurable framework for integrating repair into existing code.

2.2 Repair Algorithm

The repair algorithm takes as inputs (1) an assertion that describes *what* properties the program state must satisfy and (2) a state that violates them, and generates a new state that satisfies the desired properties. The repair algorithm employs a backtracking search that performs systematic exploration of a neighborhood of the given state and uses *symbolic execution* [48, 75, 97, 106] as well as heuristics to perform efficient and effective repair.

Experiments using libraries and applications show that the repair algorithm effectively repairs complex structures with few thousand nodes, while enabling systems to recover from potentially crippling errors.

2.3 Repair Framework

We developed an assertion-based repair framework, *Juzi*, for repairing Java programs. Juzi introduces a new assertion statement `assertAndRepair`, to be used by developers for incorporating repair in programs. It provides an abstraction of the actions performed by the repair algorithm to help the user understand what is being altered by repair. It also supports custom abstraction functions provided by the users. The abstraction is presented in the form of repair-logs that are tailored towards helping users debug their programs. The level of detail in the logs is configured by the user. A key feature of Juzi is enabling the user to control the repair algorithm. Juzi provides the `Repairable` Java interface that allows the user to mark the classes to be considered for repair. Users also indicate what fields the repair algorithm is allowed to mutate, and what data values can be introduced by the repair algorithm.

2.4 Optimizations

To enhance the performance of the repair algorithm, we devised key optimizations that target the state exploration strategy and the backtracking engine.

2.4.1 Static Analysis for Repair

We developed STARC, which uses static analysis to scale the performance of the original repair algorithm to handle larger structures with complex structural constraints. The key idea behind STARC is as follows: the constraints described in `repOk` represent the desired properties of the program state that enable the program execution to proceed. By analyzing `repOk`, information can be extracted about the target data structure that can help guide the search during repair. STARC statically analyzes `repOk` to identify: (1) the *recurrent fields*, i.e., fields that `repOk` uses to traverse the structure; and (2) *local field* constraints, i.e., how the value of an object field is related to the value of a neighboring object field. The result of the static analysis is used to guide the original repair algorithm to the more likely candidates to repair a fault. Experimental results show that STARC scales the performance of the original repair algorithm by up to an order of magnitude.

2.4.2 Check-point Based Backtracking

We developed an efficient light-weight search engine for repair that performs check-point based backtracking by incrementally storing partial program states and performing *abstract undo operations* for restore. This approach is based on two key insights: (1) `repOk` methods check desired properties by traversing the given structure without mutating them; and (2) the traversals are over object graphs and are often implemented using standard work-list-based algorithms that keep track of sets of visited nodes to prevent infinite traversals. The first insight allows us to define a minimal set of state components to store, which reduces storage overhead. The second insight allows us to use our own library classes in place of Java libraries, such as sets and lists, that enable efficient backtracking. Our approach gains its efficiency by avoiding the performance overhead imposed by code re-execution based approaches [10,23,35,46] while reducing the overhead of maintaining the program state before and after backtracking. Experimental results show an order of magnitude speed-up when integrated with STARC.

2.5 Repair Based Generation

We envision an alternative use of repair for test input generation of large data structures. A key observation behind the generation approach is that while the problem of generating an input that satisfies all the given constraints is hard, generating a structure at random, which may or may not satisfy the constraints but has a desired number of objects is straightforward. Indeed, a structure generated at random is highly unlikely to satisfy any of the desired constraints. However, it can be *repaired* using STARC to transform it so that it satisfies all the desired constraints. A comparison with current search based as well as SAT based techniques for test input generation shows up to two orders of magnitude improvement in generation time.

2.6 Organization

The remainder of this dissertation is organized as follows. Chapter 3 presents the core repair algorithm. It argues the correctness of the algorithm and presents a snapshot of the results on repairing a set of library data structures. Chapter 4 presents the design and implementation of an assertion-based repair framework. Chapter 5 presents the optimizations performed on the core algorithm to scale its performance. An evaluation of the approach on repairing both library data structures and stand-alone applications is then presented in Chapter 6. Chapter 7 then presents an application of repair for test input generation. Chapter 8 compares our work with related work in the field of software testing, verification and validation. Chapter 9 presents key future directions and describes the work being performed on repair at the time of writing this dissertation. Finally, Chapter 10 concludes the dissertation.

Chapter 3

Assertion-based Repair

A form of specification that programmers use is assertions—statements that evaluate Boolean expressions that represent desired properties. If an assertion evaluates to `false` at run-time, the program is highly likely to have reached an erroneous state. Errors—however seemingly innocuous—in a program state, if left untreated, can have serious consequences. The standard approach when an error is detected at runtime is to terminate the program, debug it if possible, and re-execute it. An alternative to program termination is repair. Instead of terminating a program, repair its state and let it continue.

This chapter presents a novel algorithm for repairing corruptions that occur in complex data structures. Given a structure that violates an assertion that represents its integrity constraints, the repair algorithm uses a systematic search based on symbolic execution to repair the structure, i.e., mutates it such that the resulting structure satisfies the given constraints.

This chapter also argues the correctness of the repair algorithm and evaluates it on repairing a set of complex structures. Experimental results show that the algorithm effectively repairs corrupt structures with a small number of errors and with a few thousand objects.

3.1 Examples

We present two examples of repairing circular doubly linked lists and binary search trees to illustrate the repair algorithm and its key components. These examples demonstrate how the repair algorithm can on-the-fly repair faults in both the reference as well as the data members of the data structures.

3.1.1 Doubly Linked List

Consider the following class declaration of a circular doubly linked list:

```
class DoublyLinkedList {
    Node header;
    int size;

    static class Node {
        int element;
        Node next;
        Node prev;
    }
}
```

The `DoublyLinkedList` class declares an internal `Node` class that models the nodes of the list. Each list has a `header` field and stores the number of nodes reachable from `header` in the `size` field. Each `Node` instance holds two pointers, `next` and `prev`, and an integer field, `element`.

The structural integrity constraints (class invariant) of `DoublyLinkedList` are: (1) circular structure along `next`; (2) transpose relation between the `next` and `prev` fields; and (3) number of nodes reachable from the `header` following `next` cached in `size`. An empty list has a `null` `header` and its `size` is 0.

```
boolean repOk() {
L0     // If the header is null, size must be 0
L1     if (header == null) return size == 0;
L2     Set visited = new HashSet();
L3     visited.add(header);
L4     Node current = header;
L5     while (true) {
L6         Node n = current.next;
L7         // Next cannot be null
L8         if (n == null) return false;
L9         // Circularity constraint
L10        if (!visited.add(n)) {
L11            if (visited.size() != size) return false;
L12            if (n.prev != current) return false;
L13            else break;
L14        }
L15        // Prev is transpose of next
L16        if (n.prev != current) return false;
L17        current = n;
L18    }
L19    return true; }
```

Figure 3.1: Class invariant for the `DoublyLinkedList`.

The structural constraints of the `DoublyLinkedList` can be written as a Java predicate that returns `true` if and only if its input satisfies all the constraints. Following the literature, we term such a Java predicate `repOk` and for object-oriented programs, we term structural invariants, class invariants [81]. The class invariant for the `DoublyLinkedList` class is displayed in Figure 3.1.

An assertion can invoke `repOk` to check the structural constraints. For example, the following Java `assert` statement checks them at the beginning of the `add` method in `DoublyLinkedList`:

```
Object add(int element) {
    assert repOk();
    ...
}
```

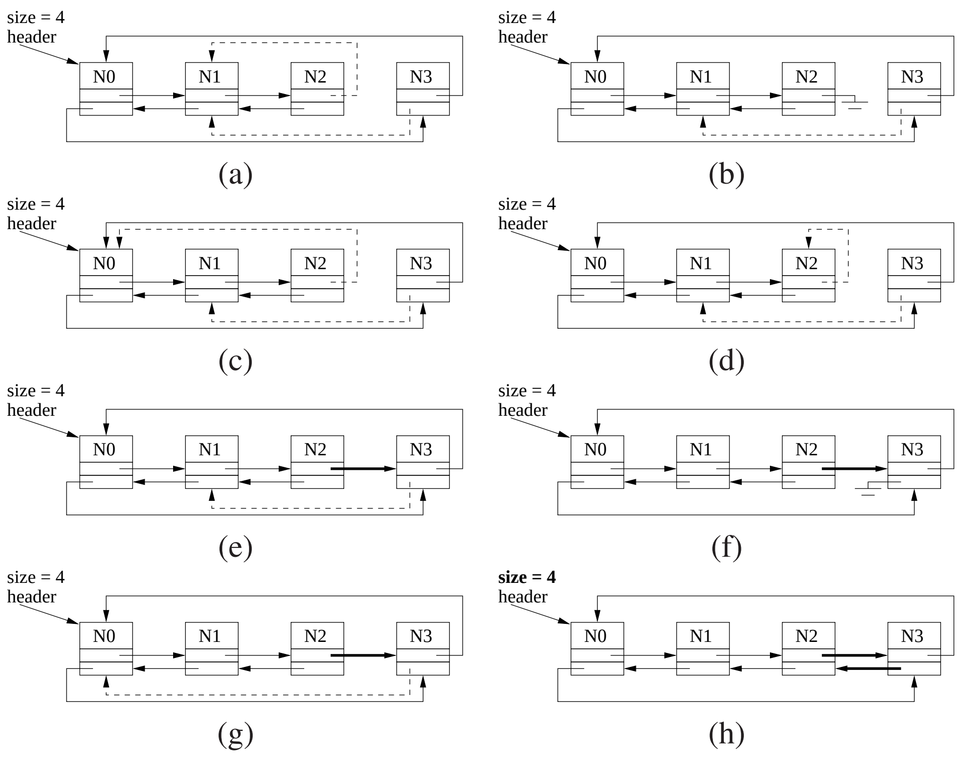


Figure 3.2: Repairing a circular doubly linked list.

To illustrate repair, consider the structures shown in Figure 3.2. The dashed arrows represent violations of the structural constraints. The bold arrows represent repaired fields. Figure 3.2 (a) shows a doubly linked list with two corruptions: (1) the `next` of `N2` is `N1` but the `prev` of `N1` is not `N2`, and (2) the `prev` of `N3` is `N1` but the `next` of `N1` is not `N3`. Figures 3.2 (b–h) show the mutations that the repair algorithm performs to repair the corrupt structure.

Given the corrupt structure in Figure 3.2 (a), and the `repOk` predicate, the repair algorithm first invokes `repOk` on the structure, and then monitors the field accesses during the execution of `repOk`. When `repOk` returns `false` due to a constraint violation, the repair algorithm systematically mutates the last field accessed by `repOk` (Section 3.3). For this example, we assume that the `size` field is correct and that it is not mutated by the repair algorithm. The next example illustrates how data fields are repaired.

Figures 3.2 (b–h) show the sequence of mutations that the repair algorithm performs to repair the faults in the structure of the list. During the first invocation of `repOk`, the last field accessed is the `next` field of `N2`. Thus, the algorithm systematically mutates this field to: (1) `null`; (2) list nodes already encountered during `repOk`'s execution (`N0` and `N2`); and (3) a list node not yet encountered during `repOk`'s execution (`N3`). (Note that the algorithm does not try node `N1`, since it is the original value of the `next` field and it has already been checked by the first execution of `repOk`.) The order of candidate selection for field mutations enables covering all non-isomorphic structures that can be generated by a program (Section 3.4). After each mutation the algorithm invokes the `repOk` predicate again to check for constraint satisfaction. Setting `N2.next` to `N3` allows the execution of `repOk` to proceed further.

The algorithm then detects the corruption in the `prev` field of node `N3`, and repairs it similarly (Figures 3.2 (e-h)). For this example, the repair algorithm performs a total of seven mutations to repair the violations in the structure of the initial corrupt list. We term these mutations *repair actions* in the rest of the document.

3.1.2 Binary Search Tree

The `DoublyLinkedList` example illustrates the use of the repair algorithm to repair faults that violate the structural constraints of a list. We now present an example that illustrates the use of the repair algorithm to repair faults that violate both the structural as well as the data constraints. We describe how the repair algorithm generates primitive values that satisfy the data constraints of a structure.

Consider the following class declaration of a binary search tree, i.e., an acyclic graph that satisfies the search constraints on the values of its nodes:

```
class BinarySearchTree {
    Node root;
    int size;

    static class Node {
        int element;
        Node left;
        Node right;
    }
}
```

Each `BinarySearchTree` object has a `root` node and stores the number of nodes in the `size` field. Each `Node` object has an integer value called `element` and has a `left` and a `right` child. The class invariant of `BinarySearchTree` can be formulated as follows.

```
boolean repOk() {
    if (!isAcyclic()) return false;
    if (!sizeOk()) return false;
    if (!searchConstraintsOk()) return false;
    return true;
}
```

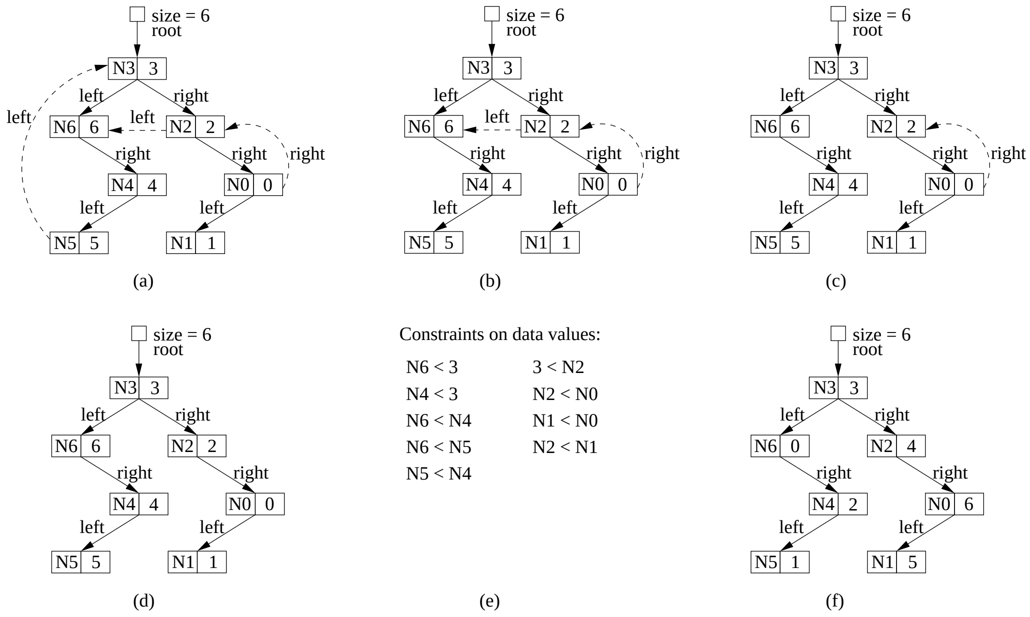


Figure 3.3: Repairing a binary search tree.

When invoked on a `BinarySearchTree` object *o*, the predicate `repOk` traverses the object graph rooted at *o* and checks all the constraints that characterize a binary search tree including, acyclicity along the `left` and `right` fields, consistency of the `size` field, and the correctness of the search order of the data members. If any constraint is violated the predicate returns `false`; otherwise, it returns `true`. The helper methods are implemented as standard work-list-based algorithms that keep track of visited nodes [85].

To repair faults in the primitive fields of the structure, the repair algorithm uses on-demand symbolic execution [69] where a corrupt field is treated symbolically, and a path condition is computed for that field during the execution of `repOk`. Once the path condition is computed, the algorithm then uses a decision procedure [6] to solve the path condition and determine the correct values to repair the field.

To illustrate, consider the binary search tree in Figure 3.3(a). The dashed lines represent fields that violate the acyclicity constraints. The elements in the tree are inserted in reverse order. Figures 3.3(b-d) show the steps that the repair algorithm takes to break the cycles in the structure. Following a depth first traversal which accesses the `left` field before the `right` field, the repair algorithm breaks a cycle each time it encounters an already visited node. Figure 3.3(e) shows the path condition after symbolically executing `repOk` on the repaired structure. The path condition contains the constraints on the order of the data values. Figure 3.3(f) shows the repaired structure after solving the path condition and reordering the values in the tree.

Using symbolic execution, the repair algorithm discovers the constraints that the data members need to satisfy for `repOk` to return `true`. By solving these constraints the repair algorithm determines values for the data members that repair the structure.

3.2 Background

In this section we give a brief description of the basics of symbolic execution and systematic searches, and illustrate how the repair problem relates to these techniques.

3.2.1 Symbolic Execution

The symbolic execution of a program is the process of running the program on symbols rather than concrete values [75]. These symbols represent arbitrary values that the input variables can take. To enable symbolic execution, the semantics of the program operations are defined on symbolic variables. Program instructions are abstracted as one of two types: *assignment* and *conditional* instructions. For assignment instructions, symbolic execution uses a dedicated memory model for transferring information between symbolic variables. For conditional instructions, symbolic execution considers both outputs of a condition and executes the two branches of the conditional instruction. This enables executing all the paths of a program.

In addition to executing all program paths, symbolic execution populates the necessary constraints on the input variables for an execution to take a specific path. Solving these constraints determines whether the corresponding path is reachable through a concrete execution or not. If the path is reachable, concrete values are generated for the inputs to cover that path. To illustrate, consider symbolically executing the method below that takes two integers x and y, and returns the absolute value of the difference:

```
int difference(int x, int y) {
L1.     int diff;
L2.     if (x > y)
L3.         diff = x - y;
L4.     else
L5.         diff = y - x;
L6.     return diff; }
```

Symbolic execution starts by assigning two symbols, say `X` and `Y`, to the input variables `x` and `y`. When executing the conditional statement on line `L2`, two cases are considered. In the first case, the `if` branch is taken, the path condition is updated with the constraint `X > Y`, and the `diff` variable is assigned to a symbolic expression corresponding to the difference operation `X - Y` on line `L3`. In the second case, the `else` branch is taken, the constraint `X <= Y` is added to the path condition, and the `diff` variable is assigned to the symbolic expression `Y - X` on line `L5`. The path conditions are then solved to generate concrete values for the input variables that allow covering all the method's paths.

3.2.1.1 Relationship with Repair

The repair algorithm uses symbolic execution to generate a set of constraints on the data members of a data structure. It then solves these constraints to determine values for the data members that satisfy the desired constraints. For example, consider the implementation of the helper method `isOrdered` from the `BinarySearchTree` example in Section 3.1.2.

```
boolean isOrdered(Node n, int min, int max) {
    // Check if root is less than the minimum of the right sub−tree
    // and greater than the maximum of the left sub−tree
    if ((n.element <= min) || (n.element >= max)) return false;
    // Check if the left sub−tree is ordered
    if (n.left != null) {
        if (!isOrdered(n.left, min, n.element)) return false;
    }
    // Check if the right sub−tree is ordered
    if (n.right != null) {
        if (!isOrdered(n.right, n.element, max)) return false;
    }
    return true; }
```

The above method checks if the elements in the binary search tree satisfy the correct search order. Using symbolic execution, the method is executed on the root node of the tree to determine the path constraints that result in `isOrdered` returning the value `true`. By solving these constraints values can be determined to fix corruptions in the data members.

3.2.2 Systematic Search

Search-based approaches have been employed is several techniques for checking software systems [3, 5, 30, 41, 58]. Three key components define a search process: (1) search variables, i.e., components that the search finds values for, (2) domains of values, i.e., the set of possible values that the search variables could take; these values determine the search space, and (3) the search algorithm which includes a search strategy that defines the order in which the search variables are assigned, the order in which values are assigned to the variables, as well as techniques and heuristics (if possible) to enhance the search efficiency.

As an example, consider solving the following equation using a search algorithm where $X1$, $X2$, $X3$ are integer values from the set $\{1, 3, 4\}$.

$$X1 + X2 + X3 = 10$$

In this problem, the search variables are $X1$, $X2$, and $X3$. The domain of values for each variable is the set $\{1, 3, 4\}$. The search space is represented as the n-ary tree in Figure 3.4. Nodes in the tree represent the search variables. We also term the nodes of the search tree as *choice points* since they represent events where the search

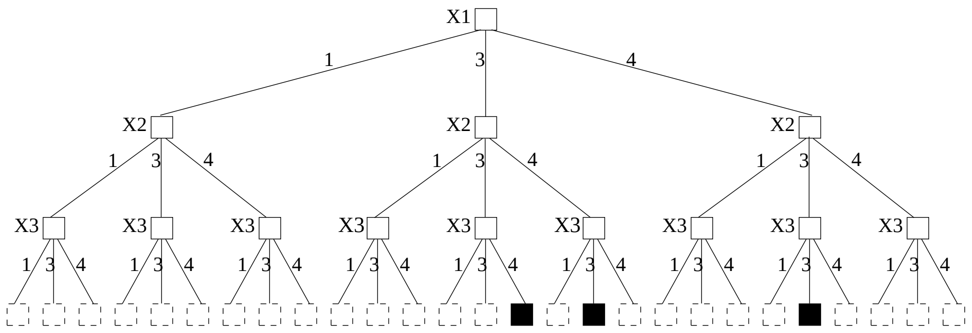


Figure 3.4: Example of a search tree.

algorithm chooses a value for a variable. Edges in the tree represent the possible values for the search variables. We term the edges as the *choices* at the *choice points*. A path from the root to a leaf represents a candidate solution. Solutions that solve the equation end at bold leaf nodes whereas solutions that do not solve the equation end at dashed leaf nodes.

The goal of the search algorithm is to find the paths that result in a valid solution of the problem by traversing the search tree. Several search strategies can be employed by the search algorithm including various traversal mechanisms such as, depth first, breadth first, or best first traversals. A search algorithm can also be configured to find all the solutions of the problem or to stop after finding the first solution. Since the state space is typically large, it is usually difficult for the search algorithm to cover all the search tree in a feasible amount of time. Several techniques can be used to target this problem and improve the search performance. For example, pruning techniques can be used to skip certain search paths that do not lead to a solution.

3.2.2.1 Application to Repair

The repair algorithm uses a systematic search to repair the values of the reference fields of a corrupt data structure. The repair problem is modeled as a search problem where (1) the search variables are the fields of the corrupt data structure; (2) the domain of values are the objects and values that compose the data structure; and (3) `repOk` executions are used to guide the search and perform the state space pruning. The goal of the search is to find values for the fields of the structure to satisfy `repOk`. To illustrate, consider the following declaration of a singly linked list.

```
class SinglyLinkedList {
    Node header;

    static class Node {
        Node next;
    }
}
```

A singly linked list has a `header` node, and each `Node` object in the list has a `next` field. The only constraint on the structure of the list is acyclicity along the `next` field. The search tree considered for finding a singly linked list with up to two nodes is displayed in Figure 3.5(a). The choice points are the fields of the linked list {`header`, `N0.next`, `N1.next`} and the choices are the nodes of the list `N0, N1` in addition to `null`. The repair algorithm employs efficient pruning techniques that are based on the Korat [10] tool for test input generation and that enable trimming large sections of the search tree. For example, when the `header` field is assigned to `null` the search does not consider any assignment for the `next` field of `N0` or `N1`.

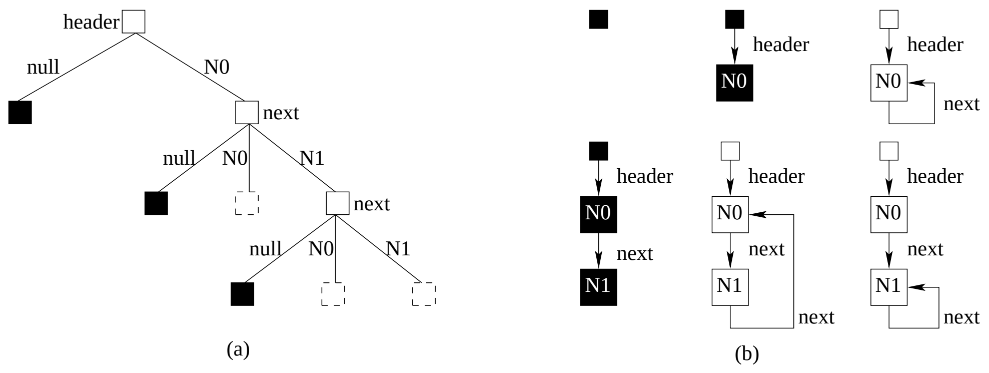


Figure 3.5: The search tree for repairing a singly linked list.

Moreover, only one node is considered for the `header` field, since considering the other node only generates isomorphic structures. Using a depth first traversal of the search tree, the search algorithm considers the lists in Figure 3.5(b) out of which three lists satisfy the constraints.

This section illustrated how symbolic execution and systematic searches can be separately used for repairing the data and the reference fields of a data structure. In typical data structures, however, these two problems are not independent. The next section describes the repair algorithm which combines the two techniques to repair data structures.

3.3 Repair Algorithm

We define repair as follows:

Definition. Given a data structure *s* such that *!s.repOk()*, i.e., *s* violates the structural integrity constraints, generate *t* such that *t.repOk()*, i.e., *t* satisfies the same constraints.

A key property of this definition is that the correctness specification is defined with respect to a single program state. The definition does not constrain repair to be performed with respect to a hypothetically correct reference state, but rather defines repair as a transformation from the corrupt state into one that satisfies the correctness specification. Another property of this definition is that it is *relaxed* with respect to the repaired structure, i.e., it does not require the repaired structure to satisfy any extra properties except what is defined in the correctness specification. For example, the definition leaves the notion of similarity between the original and repaired structure undefined.

This definition directs the design of the repair algorithm to primarily target generating a structure that satisfies the integrity constraints specified by the user. Any extra constraints on the generated structure are then added as an extension to the original algorithm. For example, similarity can be represented as a distance-metric between two graphs, one representing the initial corrupt structure, and the other representing the new repaired structure.

We present a repair algorithm based on Java assertions. The algorithm implements a dedicated solver for imperative constraints given as a `repOk` method. This solver only operates on a single program state. Two fundamental techniques that have been widely used in several techniques for testing and verification are structural constraint solving and symbolic execution. The repair algorithm combines these two techniques. It employs a systematic backtracking search for solving the structural constraints and symbolic execution for solving the data constraints. By solving the structural constraints the algorithm repairs faults in the reference fields of the structure and by solving data constraints, it repairs faults in the primitive fields of the structure.

Figure 3.6 shows the pseudo-code for the repair algorithm. The algorithm takes a `repOk` predicate and a structure `s` as inputs and returns a boolean value indicating whether the repair is successful or not.

Primarily, the algorithm performs a systematic search of the predicate space, while mutating the fields of the given structure to satisfy the given predicate. The algorithm first initializes the search by computing bounds on the search space. It does this by calling the method `getTypeDomains` which traverses the given structure and for each type T it records $domain(T)$ which holds all the values of type T that are stored in the structure. The recorded domains represent the candidates for repairing the corrupt fields in the structure.

The algorithm then proceeds with the search loop (the "SEARCH" label in Figure 3.6) and repeatedly invokes `repOk` on the given structure while monitoring its execution (the method `runAndMonitor`). During each invocation of `repOk` the repair algorithm records information about the data structure. First, it records the order in which the fields are accessed. This information is returned as a stack of fields indicated by the `stack` variable in Figure 3.6. Second, for each accessed field, it records a list of objects of the fields's type encountered when the field was accessed in `repOk`. When `repOk` reads an object field, the repair algorithm marks the value of the field as *visited.* The objects are stored in the `visited` map in Figure 3.6. Third, it records a path condition that holds the constraints on symbolic fields generated during the execution of `repOk`. While initially all the structure

```
boolean repair(Predicate repOk, Object s) {
L1      // Traverse the given structure to compute a map that holds domains of
L2      // values of all the types declared and used in the data structure.
L3      Map<Type, Set<Object>> typeDomains = Search.getTypeDomains(s);
L4      Stack<Field> stack = new Stack<Field>();
L5      Map<Field, List<Object>> visited = new HashMap<Field, List<Object>>();
L6      ConstraintSolver solver = new ConstraintSolver();
L7      PathCondition pc = new PathCondition();
L8
L9      SEARCH:  // The main search loop
L10     do {
L11         // Monitor the execution of repOk and record:(1) field access order
L12         // (2) objects accessed for each type, and (3) a path condition holding
L13         // constraints on the symbolic fields.
L14         boolean result = executeAndMonitor(repOk, s, stack, visited, pc);
L15
L16         // If repOk returns true and the constraints are solved, return TRUE
L17         if (result && solver.solve(pc)) return true;
L18
L19         // Iterate over the accessed fields and compute the repair candidates:
L20         // For a reference field, the choices are: (1) null, (2) visited objects
L21         // of its type, and (3) a non-visited object from its type's domain.
L22         //
L23         // For a primitive field, the only choice is a symbolic variable.
L24         for (Field f : stack) {
L25             Search.computeRepairCandidates(f, visited, typeDomains);
L26         }
L27
L28         BACKTRACK: // The backtracking loop
L29         while(!stack.isEmpty()) {
L30             // Get the last field accessed and mutate its value. If all
L31             // the mutations have been considered, restore the field's
L32             // original value and get the next field in the stack.
L33             Field f = stack.top();
L34
L35             if (Search.hasNextCandidate(f)) {
L36                 Object o = Search.getNextCandidate(f);
L37                 f.assign(o);
L38                 break;
L39             } else {
L40                 f.restoreValue();
L41                 stack.pop();
L42             }
L43         } // End backtracking loop
L44     } while(!stack.isEmpty()); // End search loop
L46
L47     // If all the search is exhausted, return FALSE
L48     return false;
   }
```

Figure 3.6: The repair algorithm.

fields have concrete values, during the search process, primitive fields can become symbolic.

If the execution of `repOk` returns `true` and the path constraints are satisfiable, the search terminates with a success and the corrupt structure `s` is repaired. If the execution of `repOk` returns `false` or the path constraints are not satisfiable the algorithm systematically backtracks (the "BACKTRACK" label in Figure 3.6) mutating the values of the fields in reverse order of field access.

To perform backtracking, the algorithm first computes the possible choices for each field in the search. The number of choices for each field depends on the type of the field as well as the number of objects of the field's type encountered when the field is accessed; this information is pre-computed in the `visited` map. For reference fields the number of candidates is equal to the number of encountered objects + 2 (corresponding to `null` and one object that has not been encountered). For primitive fields, there is one possible candidate; a symbolic value corresponding to the field. The algorithm iterates over all the fields returned in the `stack` variable when monitoring `repOk`, and uses the `computeRepairCandidates` method on each field to compute and track the choices for repairing a field. The order through which the field choices are enumerated is described in Section 3.3.1.

Backtracking occurs in reverse order of field access. The algorithm tries to mutate the value of the last field accessed during the execution of `repOk`. If the field has more candidates to try, its value is mutated to the next candidate and then `repOk` is re-executed on the structure. Since all the fields of the structure except the last field have the same value, the execution of `repOk` on the mutated structure has the same field access order for those fields. This is a property of re-execution-based backtracking where the state of the search is not saved but recomputed with every iteration of the search. Once all the candidates for a field have been checked, and do not repair the structure, the repair algorithm restores the field to its original value and systematically backtracks to update the value of the second to last field accessed and so forth until all the search candidates are discovered.

If all the field mutations do not repair the structure, the repair algorithm returns `false` declaring that the structure `s` is still corrupt.

3.3.1 Non-deterministic Field Assignments

While backtracking on a field f the repair algorithm non-deterministically mutates the value of the field according to its type. The algorithm considers two cases: reference fields and primitive fields.

References. For a reference field f of type T and of value v, the algorithm non-deterministically assigns f to candidates in the following order:

1. `null`, if $v \neq$ `null`;
2. all visited objects o of type T, such that $o \neq v$ and o belongs to the objects that have already been encountered during `repOk`'s invocation, i.e., objects maintained by the `visited` variable in Figure 3.6; and
3. a new (non-visited) object o of type T, such that o has not been encountered during the execution of `repOk`; the `computeRepairCandidates` method in Figure 3.6 computes this object. It takes the field `f`, the visited objects, and

the type domains, and selects an object from the field's domain that is not in the field's visited set.

When all the candidates have been considered for a reference field f, the original value of the field is restored.

Primitives. For a primitive field f of type T and value v, the algorithm non-deterministically assigns f to a new symbolic value V, and adds the constraint $V \neq v$ to the current path condition.

Notice that a primitive field access may introduce symbolic values. For these values, further invocations of `repOk` follow forward symbolic execution [72, 75]. Since all fields initially have concrete values, the first execution of `repOk` follows standard Java semantics for these values and does not generate any constraints in the path condition. Once a field becomes symbolic, `repOk`'s execution populates a set of constraints in the path condition which are solved to determine concrete values for the field. If the constraints are not satisfiable, then the algorithm restores the original value of the field. Therefore, when the algorithm terminates, all the fields of the structure will have concrete values.

The repair algorithm builds on an initial algorithm [69] which is based on the Korat test input generator [10] and symbolic execution [69, 72].

3.3.2 Complexity

The repair algorithm uses a systematic search algorithm where (1) the search variables are the fields of the corrupt data structure, (2) the domains of values are the objects of the corrupt data structure, and (3) search space pruning is performed according to the field access order in `repOk`. The worst case performance of the algorithm is therefore the time required to cover all the search space without any pruning, which is exponential in terms of the size of the corrupt structure. Additionally, the algorithm uses re-execution-based backtracking where the state of the data structure is not saved during the search but rather rebuilt with every repair action. This approach reduces the performance and memory space overhead of maintaining the state of the data structure upon backtracking, but requires re-executing `repOk` on the structure and re-constructing the structure state with every repair actions. The number of repair actions is proportional to the size and the number of faults in the data structure. As the number of required repair actions increases, the extra `repOk` executions become expensive.

In practice, however, the experimental results (Section 3.6) show that the repair algorithm can effectively repair complex data structures with up to a few thousand nodes and tens of faults in less than a minute. The experimental results also show that the performance of the repair algorithm depends on the implementation of `repOk`. Since search space pruning is primarily performed when `repOk` returns `false`, a `repOk` formulation that terminates as soon as a fault is detected in the data structure enables effective pruning of the search space and in turn more efficient repair.

In Chapter 5, we describe a set of key optimizations for the repair algorithm that target (1) reducing the number of repair actions required to repair the corrupt structure, (2) pruning more sections of the search space, and (3) performing efficient backtracking. While these optimizations do not improve the theoretical worst-case complexity, they make our approach scale to larger structures in practice.

3.4 Correctness

In this section we argue the correctness of the repair algorithm with respect to the given structural integrity constraints. We start by stating a set of key assumptions about the correctness of the symbolic execution engine as well as a set of conditions that the given predicate must satisfy for the algorithm to behave as described in Section 3.3. We then state the correctness statement and present a correctness argument for the repair algorithm.

3.4.1 Assumptions

We list the following assumptions about the symbolic execution engine and the implementation of `repOk`. If any of the assumptions is violated, then the repair algorithm may not repair correctly.

- **A1:** Symbolic execution of `repOk` is sound and complete. The symbolic execution generates inputs that cover all the reachable paths in `repOk`, and if the symbolic execution generates an input to cover a path P, then the concrete execution on that input must traverse P.
- **A2:** Any execution of `repOk` must terminate with a boolean return value. If this assumption is violated, then the repair algorithm may not terminate.
- **A3:** The implementation of `repOk` must not depend on the identity of the

objects in the data structure; i.e., the implementation must not include invocations to the `System.identityHashCode` utility method.

- **A4:** The implementation of `repOk` must be deterministic. All the executions of `repOk` on a structure must follow the same execution path, access the fields of the structure in the same order, and return the same value.

- **A5:** The implementation of `repOk` must not access global data or static fields. The fields accessed by `repOk` must either be fields of the data structure or local variables defined within `repOk`.

The assumptions about symbolic execution are necessary to state the correctness of the repair algorithm since the repair algorithm uses symbolic execution to determine conditions on the input structure that result in `repOk` returning `true`. These assumptions disregard any imprecision that the symbolic execution may incur due to any loop structures in `repOk`, or operations in `repOk` that result in generating constraints in a non-decidable theory, e.g., non-linear arithmetic, system calls, or resource allocations. The assumptions about `repOk`'s implementation are necessary to ensure that (1) the search algorithm does not hang due to a bug in `repOk` and (2) the utilized pruning techniques do not force the search to miss a valid structure.

3.4.2 Correctness Statement

Given a `repOk` method that satisfies the above assumptions, and an input data structure; we argue that the repair algorithm is:

- **Sound:** If the repair algorithm returns `true`, the output structure does not have any symbolic members and satisfies the given constraints.

- **Complete:** If at least one valid structure can be constructed using the objects of the input structure, the algorithm returns `true` and outputs a valid structure.

3.4.3 Correctness Argument

Soundness: We argue that the repair algorithm repairs a corrupt structure into one that satisfies the consistency constraints. If the repair algorithm returns `true`, then the algorithm terminates at line $L17$ in Figure 3.6. To satisfy the conditions on line $L17$, the execution of `repOk` on the structure returns `true` and the path constraints are satisfiable. Thus, the output structure satisfies the constraints. If the algorithm returns `false`, then it terminates at line $L48$ in Figure 3.6. To execute line $L48$ the `stack` variable is empty and all the structure fields are restored to their original concrete values. Thus, the output structure is equivalent to the original corrupt structure.

Completeness: We argue that the pruning techniques employed by the repair algorithm do not force the search algorithm to skip any valid structure, i.e., a structure that satisfies the constraints described in `repOk`.

Consider a naive search algorithm that takes a data structure p as input and enumerates all the possible candidates for the fields of the structure without any pruning. For a reference field f of type T, the candidate values are `null` in addition to all the structure objects of type T. For a primitive field f of type T and value v, the candidate values are v and a symbolic variable V corresponding to f.

The result of the naive search is a set of structures that can be categorized into two groups: (1) structures with all the fields having concrete values; we term these structures as the *concrete structures* and (2) structures with some fields having symbolic values; we term these structures as the *symbolic structures*. Running `repOk` on a concrete structure results in a truth value indicating whether the structure is valid or not. Symbolically executing `repOk` on a symbolic structure results in covering all the paths in `repOk` along with a path condition for each path. For repair, we are interested in the first reachable path in `repOk` that returns `true`. From assumption $A1$, the symbolic execution is complete and thus any valid structure is either a concrete structure or one that is generated by the symbolic execution.

The repair algorithm employs two pruning techniques on the naive search algorithm. The first pruning technique is through the execution of `repOk`. Once `repOk` returns `false`, the algorithm backtracks only on the fields accessed by `repOk`'s execution; the values of the rest of the structure's fields are not considered. The second pruning technique is through the non-deterministic field choices performed during backtracking. The algorithm only considers one object that has not been encountered during the execution of `repOk` instead of all such objects. We argue that the pruning techniques do not skip any valid structure from the ones generated by the naive algorithm.

The first type of pruning does not affect the completeness of the repair algorithm since only structures that result in `repOk` returning `false` are not considered. Consider an execution of `repOk` that returns `false` and with a field access order $\{f1, f2, .., fn\}$. From assumptions $A4$ and $A5$, since `repOk` is deterministic and does not access global data, any data structure with the same values of fields $\{f1, f2, .., fn\}$ results in the same execution of `repOk`, and thus all the rest of the fields do not affect `repOk`'s result.

The second type of pruning does not skip any valid data structure and only considers non-isomorphic structures. This follows from the proof of correctness and optimality of Korat [10] and it is described in detail in a technical report on the evaluation of Korat [86]. Briefly, form assumption $A3$, `repOk` does not depend on the identity of the assigned objects. Therefore, any two objects $o1$ and $o2$ of type T that have not been encountered during `repOk`'s execution are semantically equivalent since their fields are not yet accessed. Thus, using either $o1$ or $o2$ as candidates when assigning a field f of type T results in two isomorphic structures.

Given the above argument, any pruning performed by the algorithm does not ignore any valid structure. If a structure exists that satisfies `repOk` it is either a concrete structure, or it is a symbolic structures whose fields' values are generated by the symbolic execution.

Correctness: The correctness of the repair algorithm with respect to the given constraints follows from its soundness and completeness.

3.5 Limitations

Repairing data values in a structure requires care. For example, in repairing `DoublyLinkedList`, while we expect repair to re-establish structural constraints of a doubly linked list, we do not expect repair to modify any particular `element`. However, nothing prevents repair from introducing a new value for the `size` field of the `DoublyLinkedList` if such a value satisfies the desired constraints. This may cause a problem in some applications that do not tolerate any changes in the repaired data structure. We mitigate this problem by allowing users to specify fields that should not be mutated by repair. By declaring `size` as un-modifiable, the user is assured that repair will only re-structure the existing list entries to satisfy the invariants.

While proving the correctness of repair, we made strong assumptions about the completeness of symbolic execution. However, since `repOk` is an arbitrary Java method, finding an input for which the method returns `true` is undecidable. In fact, non-linear constraints over integers are undecidable. We have not found this to be a problem in practice. A reason for that is that `repOk` predicates are special methods that focus on structural integrity and the constraints of commonly used data structures seldom involve complex arithmetic. Even when `repOk` uses complex arithmetic, bounded enumeration enables exhaustive exploration of a bounded input space.

3.6 Preliminary Results

This section gives a quick overview of the evaluation of the repair algorithm and its ability to repair different types of constraints. A detailed evaluation of the repair algorithm is described in Chapter 6.

We present the results for applying the repair algorithm to the three data structures described in this chapter: singly linked list, circular doubly linked list, and binary search tree. For each subject structure, we evaluate the performance of the repair routine by *injecting errors*, i.e., corrupting object fields and repairing them as follows. Given s, the desired size of a structure, and e the desired number of fields to corrupt:

1. Generate a structure of size s.
2. Corrupt e fields at random in the structure; a corruption is a triple $\langle o, f, v \rangle$, where object o's reference field f is assigned value v, which is either `null` or a reference to an object of a compatible type.
3. Repair the corrupt structure.

For all subjects, we re-used `repOk` predicates that were developed previously [10, 30]. Re-use of specifications is a key strength of assertion-based repair. Indeed, if a `repOk` is already available for a subject, say because it was tested with Korat [10], it is used in repair for free. All experiments used a 1.7 GHz Pentium D with 2 GB of RAM.

Subject	Size	Repair (ms) [#errors≤1]	Repair (ms) [#errors≤5]	Repair (ms) [#errors≤10]	Repair (ms) [#errors≤15]	Repair (ms) [#errors≤20]
Singly linked list	100	≤ 2	not applicable	not applicable	not applicable	not applicable
	1,000	21				
	10,000	168				
	100,000	1,666				
Doubly linked list	50	170	204	239	261	293
	100	302	382	492	573	651
	200	2,721	2,961	3,240	3,417	3,755
	400	25,175	26,502	27,344	30,642	32,035
Binary search tree	500	≤ 2	47	62	141	219
	1000	14	109	156	218	392
	2000	33	266	516	762	968
	4000	68	703	1,797	3,031	4,218

Table 3.1: Results for applying the repair algorithm on three subject structures.

We evaluate how the repair time varies with the structure size and the number of corrupt fields. Table 3.1 tabulates the results for the three subjects. For each subject, we tabulate different structure sizes and the time to repair when there are 1, 5, 10, 15 and 20 errors injected. We choose these numbers of errors because in a real situation, we expect a small number of corruptions. The last five columns are labeled $[\#errors \leq n]$ since it is possible (though unlikely) for a randomly generated error to set the value of a field to its original value. The repair times for sizes less than 50 are negligible and not shown here.

For singly linked acyclic list, Table 3.1 shows times for $[\#errors \leq 1]$ only as a singly linked list either has zero or exactly one cycle, irrespective of how many `next` fields are mutated. Since at most one fault can occur is the structure of a singly linked list, the repair algorithm can handle lists with 100,000 objects within two seconds.

Doubly linked list was the most difficult among the above subjects to repair.

The complexity of the structure is in preserving reachability since each node must be reachable from any node in the structure. The first valid structure might not have the same number of nodes as the corrupt structure, and thus, the repair algorithm keeps searching for a valid structure with the original size, if possible. For these structures, the algorithm can repair structures with 400 nodes and $[\#errors \leq 20]$ in around thirty two seconds. Note that for the doubly linked list example, the variation of the repair time with respect to the number of faults is minimal. We explain this behavior in detail in Section 3.6.1.

For binary search tree the only structural constraint is acyclicity. The data constraint corresponds to the correct search order of the data elements. For these structures the algorithm repairs structures with up to 4,000 nodes in less that five seconds.

We point out that the injected errors cripple the subject implementations, causing failures ranging from un-handled exceptions to infinite loops. The repair algorithm successfully repairs the corrupt structures and enables the respective applications to continue to execute.

3.6.1 Understanding the Results of Repair

The previous section evaluated the performance of repair in terms of the size of the structure and the number of present faults. It computed the repair time in the presence of randomly injected faults which gives an overall idea of the average performance of the repair algorithm.

The repair time is affected by the number of repair actions performed during repair. From the algorithm in Figure 3.6, each repair action requires executing `repOk` on the structure to check if the performed action repairs a corruption. Several parameters affect the number of repair actions required to repair a data structure. These parameters include: (1) the location of the fault with respect to the root of the structure, (2) correlations that exist between the corruptions, and (3) the violated constraints. To further understand the results of repair, we study the effect of these parameters on both the number of repair actions performed during repair and the repair time. We use the doubly linked list example as a representative example since its structure is characterized by both *global* properties such as the circularity and reachability constraints and *local* properties such as the transpose relation between the `next` and `prev` fields.

The nature of the violated constraints is very important for analyzing the performance of the repair algorithm. Faults in a doubly linked list can occur in the `next`, `prev`, or `size` fields of the structure. Faults in the `next` field may break the circularity constraint and the transpose relation, faults in the `size` field may break the reachability constraint, and faults in the `prev` field may break the transpose relation. We study the result of repairing faults in each of these fields independently, and then consider the cases of simultaneous faults.

Repairing the `next` field: For this experiment, we first configure the repair algorithm not to mutate the value of the `size` field. We consider repairing a doubly linked list with 200 nodes. We select one node of the list and assign its `next` field to the `header` node. We vary the location of the selected node with respect to the `header` node and study the effect of the fault location on the repair time. We run

Fault location	*Node#40*	*Node#80*	*Node#120*	*Node#160*	*Node#200*
Repair actions	58,240	50,780	38, 520	21,460	2,002
Repair time (ms)	3,625	3,321	2,735	1,670	203

Table 3.2: Variation of the repair time with respect to the location of the fault in the `next` field.

the repair algorithm for 5 different locations and compute the repair time as well as the number of repair actions. The result is displayed in Table 3.2. The first row shows the distance from the `header` node following the `next` field. The second and third rows show the number of repair actions performed and the repair time in milliseconds.

Note that for repairing a fault in the `next` field, the number of required repair actions decreases as the fault moves away from the `header` node. For example, the number of repair actions required to fix a corruption at the end of the list is almost 20 time less than the number of actions required to repair a fault at the 40th node. This is reflected on the repair time which also decreases as the fault moves further away from the `header` node. This result is justified as follows. To repair a fault in the `next` field, the repair algorithm may require to reorder the nodes of the list depending on the choice of the non-encountered node during the non-deterministic choice assignments. If such a fault occurs early during traversal, the repair algorithm needs to reorder a large portion of the remaining nodes. As the fault moves away from the `header` node, the size of the list that needs to be reordered decreases and thus the number of required repair actions decreases.

We increase the number of faults injected at a time by injecting four faults

Fault location	*Node#400*	*Node#800*	*Node#1200*	*Node#1600*	*Node#2,000*
Repair actions	401	801	1,201	1,601	2,001
Repair time (ms)	110	360	719	1,503	2,688

Table 3.3: Variation of the repair time with respect to the location of the fault in the `prev` field.

in the `next` fields of nodes at distances 40, 80, 120, and 160 from the `header` node and run the repair algorithm. The algorithm performs 60,830 mutations on the list and repairs it in 3,870 milliseconds. Note that the repair time is similar to that of repairing a single fault at the 40th node. Since repairing the fault at the 40th node reorders the rest of the list, it automatically fixes the other faults.

We then release the constraint on the `size` field and allow the repair algorithm to mutate it. We inject a single fault in the `next` field of the 100th node. To repair the list, the algorithm performs only 103 mutations in 68 milliseconds. The repaired list, however, includes only 100 nodes. Instead of finding a list with the original nodes, the algorithm modifies the `size` field to reflect the number of nodes in the list (in this case 100 nodes reachable through `next`) and repairs the structure.

Repairing the `prev` field: We run similar experiments to study how the repair algorithm performs when repairing faults in the `prev` field. We consider a doubly linked list with 2,000 nodes, and inject faults in the `prev` field of a set of the list's nodes. We again study the effect of the fault location on the performance of repair. The results are tabulated in Table 3.3.

To repair a fault in the `prev` field, the repair algorithm searches all the list nodes encountered during `repOk`'s traversal for the right candidate to repair the corruption. As the distance of the faulty field from the `header` node increases, the number of repair actions increases since the algorithm needs to try more candidates before finding the right one. Moreover, there is no need to reorder the elements of the list while repairing the `prev` field, and therefore, unlike repairing faults in the `next` field, the number of non-encountered objects does not affect the performance of the repair algorithm. The repair time also increases as the location of the fault moves away from the `header` node. Each repair action requires traversing the structure from the `header` node, which adds a quadratic growth in repair time with respect to the location of the fault.

We then study the effect of repairing faults in the `prev` fields of multiple list nodes on the performance of the repair algorithm. We consider a list of 1,000 nodes and set the `prev` field of each of the nodes to `null`. To repair the list the repair algorithm performs 501,500 mutations in 3 minutes. Note that unlike faults in the `next` field where repairing the first detected fault dominates the performance of repair, faults in the `prev` field are independent, and the repair time is the cumulative time for repairing each fault in the order they occur while traversing the list.

The results of this section show that parameters such as the location of the fault and the violated constraint utterly affect the performance of the repair algorithm. The effect of these parameters differ depending on the nature of the structure and the fields being repaired. To generalize the evaluation as much as possible, we take such factors into consideration in our fault injection methodology (Chapter 6) and in addition to randomly injected faults, we introduce specific faults at different locations violating all the structural constraints that characterize the structures.

Chapter 4

Framework

Repair-based error recovery is viewed as a three stage process. The first stage is a pre-deployment stage. Programmers develop *repairable* classes, i.e., classes that include a `repOk` method that describes the class invariant, and use *repair assertions* instead of standard assertions to check for the consistency of the repairable classes instances with respect to the class invariant. The second stage is the integration stage. The repair logic is integrated into the program to generate *repairable programs*. The last stage is a post-deployment stage. The program from the second stage is executed. The program behaves similarly to the original program in case no assertion violation is detected, but automatically repairs itself upon a violation. Unlike standard assertions, the repair assertions trigger the repair algorithm upon a violation and enable error recovery.

To be attractive for practitioners, assertion-based repair must be easy to incorporate in existing software systems. This chapter presents an implementation of a framework, code named *Juzi*, for assertion-based repair of Java programs. Figure 4.1 shows the architecture of the Juzi framework for error recovery. Juzi uses code instrumentation to integrate repair into Java programs. Juzi takes as input a program that implements a set of repairable classes and uses repair assertions to

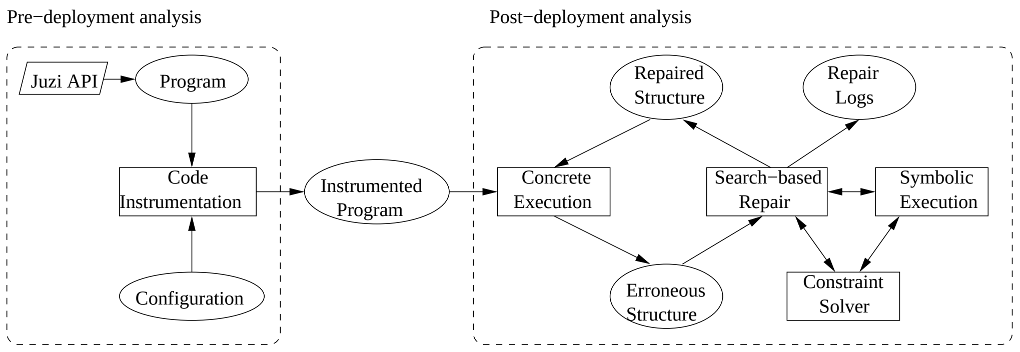


Figure 4.1: Juzi: a framework for repairing Java programs.

check for its properties (Section 4.1). Based on a set of user configurations (Section 4.3), Juzi instruments the program classes and generates a new program with embedded repair logic (Section 4.2). Once the instrumented program is executed the post-deployment analysis starts. Upon an assertion violation, the erroneous program state is detected and the repair loop is triggered. The repair loop consists of the search algorithm, the symbolic execution engine, and the constraint solver. The result of the repair loop is a repaired state that potentially enables the program execution to proceed.

Additionally, Juzi provides an abstraction of the actions performed by the repair algorithm to help the user understand what is being altered by repair (Section 4.3.3). The abstraction is presented in the form of repair-logs that are tailored towards helping users debug their programs.

Juzi consists of three key components: (1) a Java API for programmers to use while writing programs, (2) a code-instrumentation module, that performs source-to-source translation of the program to integrate the repair logic into the programs's code, and (3) a configuration module that allows user control of both the repair algorithm as well as the output of repair. We next describe each of the modules in detail.

4.1 Juzi API

Juzi provides a set of API methods and interfaces for programmers to use in order to write repairable data structures. Juzi introduces the `Repairable` Java interface for developers to identify the Java classes to be considered for repair. A repairable class is a class which implements a `repOk` method that describes its class invariant. Juzi also introduces a new assertion `assertAndRepair` to be used for asserting the consistency of a repairable class state with respect to `repOk`. The declaration of the components is shown below:

```
public interface Repairable {
  public boolean repOk();
}
public class Juzi {
  public static boolean assertAndRepair(Repairable obj) {..}
  public static boolean assertAndRepair(Repairable[] objs) {..}
}
```

The `Repairable` interface declares one method, `repOk`. Any class that implements the `Repairable` interface must define `repOk`. Juzi provides two versions of the `assertAndRepair` API method; one that takes a single repairable object and another that takes an array of repairable objects.

```java
public class DoublyLinkedList implements Repairable {
    Node header;
    int size;

    public static class Node {
        int element;
        Node next;
        Node prev;
    }

    public boolean repOk() {..}

    public void addFirst(int element) {..}

    public static DoublyLinkedList createList(int size) {
        DoublyLinkedList ll = new DoublyLinkedList();
        for (int i = 0; i < n; ++i) {
            ll.addFirst(i);
        }
        assertAndRepair(ll);
        return ll;
    }
}
```

Figure 4.2: A repairable `DoublyLinkedList` class.

To illustrate the use of the API, consider the `DoublyLinkedList` class in Figure 4.2. The class implements the `Repairable` interface and therefore implements a `repOk` method. The `addFirst` method adds an element at the beginning of the list and the `createList` method creates a list of a given size by successively calling the `addFirst` method. Good programming practices advocate asserting the integrity of the data structures at the border of public and factory methods [81]. To assert the integrity of the created list, the `assertAndRepair` method is called before returning from the `createList` method which ensures the consistency of the list with respect to the constraints described in `repOk`.

```
public static boolean assertAndRepair(Repairable obj) {
    // If the class is not instrumented,
    // then treat as a Java assertion
    if (!(obj instanceof JuziInstrumented)) {
        assert obj.repOk();
        return true;
    }

    // Execute the repair loop
    Juzi.initialize(obj);
    Search.initialize(obj);
    boolean done = false;
    do {
        PathCondition.initialize();
        if(obj.repOk()) {
            if (!PathCondition.isFeasible()) continue;
            done = true;
            break;
        }
    } while (Search.nextState());
    return done;
}
```

Figure 4.3: The implementation of the `assertAndRepair` method.

The implementation of `assertAndRepair` is displayed in Figure 4.3. The `assertAndRepair` method first checks if the class has been instrumented by Juzi or not. If the class is not instrumented, the method behaves as a standard assertion that wraps the Java `assert` statement on `repOk` and terminates the program in case of a violation. If the class is instrumented, the method executes the repair loop. Note that this implementation enables using the standard semantics of assertions before the program is deployed, and then the repair semantics can be used after deployment once the program is instrumented.

4.2 Instrumentation Engine

Juzi implements an instrumentation engine that takes the user program as input and instruments it into a functionally equivalent program that uses a set of libraries to enable repair. To compute the program classes that need to be instrumented, Juzi first detects the set of repairable classes (classes that implement the `Repairable` interface) and performs a reachability analysis to determine all the classes that the repairable classes reference, and the classes that reference the repairable classes. Juzi labels all these classes as "instrumented" by adding an empty interface `JuziInstrumented` to their interface list. This label allows runtime checking of whether a class is instrumented or not; for instance, the implementation of the `assertAndRepair` method does that. Juzi then instruments each class by (1) replacing field accesses with invocations to accessor methods that enable non-deterministic choice assignments, (2) adding boolean fields to monitor the initializations of the fields, and (3) inserting calls into the repair libraries. Juzi uses third party libraries for performing the code instrumentation at the bytecode level [12, 19, 98].

Two key classes enable repair: `Search` and `PathCondition`. Figure 4.3 shows the repair loop which performs a systematic search and uses symbolic execution [75].

The `Search` class provides a framework for state space exploration. It implements an `initialize` method which takes an input structure, and initializes the search space as described in Chapter 3. To initialize the search space, the `initialize` method traverses the structure and for each type it records the domains of values that the fields of that type could take. The recorded domains represent the candidates for repairing the corrupt fields in the structure. The `Search` class also keeps track of the current state of the search and provides the `nextState` method which puts the search into the next possible state. The `nextState` method returns `false` if the entire state space is explored.

The `PathCondition` class enables tracking the path conditions that arise in symbolic execution. The path condition is a set of constraints on program variables that, when satisfied, enable the program execution to take a specific program path. The `initialize` method clears the path condition; `isFeasible` checks whether the current path condition is satisfiable; and `update` adds a constraint to the path condition.

The repair loop starts by initializing the search environment and state by calling `Search.initialize` on the corrupt structure. The loop proceeds by repeatedly invoking `repOk` on the given structure. Each execution of `repOk` (1) monitors the order of field accesses and (2) non-deterministically updates the value of the last field accessed—if all values have been checked, systematically backtracks to update the value of the second to last field accessed and so forth (Section 3.3.1). The given `repOk` implementation, however, does not include the logic for backtracking, field monitoring, or non-deterministic field assignments. Juzi instruments `repOk` to allow such behavior.

4.2.1 State Space Exploration

To support non-deterministic choices, the `Search` class provides a `choose` method that takes an integer which represents the number of non-deterministic choices and returns an integer which represents one of these choices. For example, the assignment, `int choice = Search.choose(2);`, non-deterministically assigns the values $0, 1, 2$ to the variable `choice`.

To keep track of the current choice, `Search` holds a counter for each call site for the method `choose` in the program. At each call site, the first call to `choose` adds a counter in the `Search` class and initializes its value to 0. Further calls at a call site return the value of the counter. `Search` also provides a `nextState` method that increments the value of the last added counter. Once all the possible choices for a counter are explored, the corresponding counter is deleted. The `nextState` returns `true` if a counter is incremented, and `false` when all the counters are deleted (this indicates that the state space is explored). To illustrate, consider the following example:

```
void spaceExploration() {
L1.  Search.initialize();
L2.  do {
L3.     int i = Search.choose(1);
L4.     int j = Search.choose(2);
L5.     System.out.println(i + "  " + j);
L6.  } while(Search.nextState());
   }
```

The calls to the `choose` method at lines `L3` and `L4` set the search space by creating two counters that count from 0 to 1 and 0 to 2 respectively. The `nextState` increments the value of the counters. The `do..while` loop executes until both counters reach their maximum value and the `nextState` returns `false`. The output of executing the method `spaceExploration` is:

```
0 0
0 1
0 2
1 0
1 1
1 2
```

By associating field domain values with integer indices, the `choose` method enables non-deterministic field assignments.

4.2.2 Structure Mutation

The repair algorithm mutates the structure based on `repOk`'s executions. To enable non-deterministic field assignments, Juzi instruments the Java bytecode of both the repairable classes and the `repOk` methods.

Class instrumentation: For each field in a class, Juzi adds a boolean variable `field_is_initialized` that indicates whether a field is being accessed for the first time. To monitor field accesses and allow non-deterministic assignment, Juzi adds *get* and *set* accessor methods for each field. These methods provide the functionality for accessing and setting field values, and enable an observer to note the order of accesses. Additionally, for each class type, Juzi uses two new `java.util.Set` fields that represent the sets of visited and non-visited objects during the search. To illustrate, a snapshot of the instrumented `DoublyLinkedList` from Section 4.1 is displayed in Figure 4.4. A boolean field and two accessor methods are added for reference fields (to support non-deterministic field assignment)

```
class DoublyLinkedList implements Repairable {
    static Set visitedLists, nonVisitedLists;
    static Set visitedNodes, nonVisitedNodes;

    Node header; boolean header_is_initialized;
    SymbolicInt size; boolean size_is_initialized;

    void header(Node n) {...} // added set method
    Node header() {...} // added get method

    void size(SymbolicInt i) {...} // added set method
    SymbolicInt size() {...}  // added get method

    static class Node {
        SymbolicInt element; boolean element_is_initialized;
        Node next; boolean next_is_initialized;
        Node prev; boolean prev_is_initialized;

        void next(Node n) {...} // added set method
        Node next() {...} // added get method

        void prev(Node n) {...}  // added set method
        Node prev() {...} // added get method

        void element(SymbolicInt i) {...} // added set method
        SymbolicInt element() {...}  // added get method
        ...
    }
    ...
}
```

Figure 4.4: The instrumented `DoublyLinkedList` class.

and for primitive fields (to support symbolic execution). Two sets are added for each reference type to keep track of the visited and non-visited objects of that type.

`repOk` instrumentation: To monitor the order of field accesses in `repOk`, Juzi instruments the method's bytecode by changing all the field accesses to method invocations of the added accessor methods. To illustrate, the following bytecode corresponds to the statement "`Node current = header;`" from the `repOk` method of the `DoublyLinkedList`:

```
26:  aload_0
27:  getfield #24; //Field header: LDoublyLinkedList\$Node;
30:  astore_2
```

The instrumented bytecode is:

```
32: aload_0
33: invokevirtual #165; //Method header:() LDoublyLinkedList\$Node;
36: astore_2
```

For ease of understanding, Figure 4.5 illustrates the instrumented code at the source-code level. All the field accesses are transformed into method invocations. Operations on primitive values are changed into operations on symbolic primitives. Conditional statements are replaced with method invocations that update the path condition.

The non-deterministic assignment is performed by the methods added during class instrumentation. Figure 4.6 shows an example of the added methods for the `next` field of the `DoublyLinkedList` class. The first method (the set method) simply sets the value of `next` and marks it as accessed (initialized) by assigning the `next_is_initialized` variable to `true`. The second method (the get method) returns the current value of `next` if it is previously accessed (initialized). If it is not yet initialized, the get method non-deterministically chooses a value for `next`. The first choice is the original (possibly corrupt) value. This choice reflects the normal behavior of a field access and is made to maintain the normal execution of the program in case there is no error. The other choices are `null`, a visited `Node`, and a

```
boolean repOk() {
    if (header() == null)
        return (size().ifEQ(new IntConstant(0));
    Set visited = new HashSet();
    visited.add(header());
    Node current = header();
    while (true) {
        Node n = current.next();
        if (n == null) return false;
        if (!visited.add(n)) {
            // Symbolic check; updates the path condition
            if (size().ifNEQ(visited.size())) return false;
            if (n.prev() != current) return false;
            else break;
        }
        if (n.prev() != current) return false;
        current = n;
    }
    return true;
}
```

Figure 4.5: The instrumented `repOk` method.

new non-visited `Node` (as described in Section 3.3.1). Note that the algorithm keeps track of the visited and non-visited nodes for each field so that the correct choices are made according to the order in which the fields are accessed in `repOk`.

4.2.3 Symbolic Execution

To enable symbolic execution, instrumentation replaces type declarations of primitive types with custom library classes. For example, each integer declaration is replaced with the library class `SymbolicInt`. The `SymbolicInt` defines the semantics of the operations on symbolic integers. All the operations over primitive integers are replaced with invocations of library methods that are members of the `SymbolicInt` class. A key requirement of symbolic execution is the ability

```
// Set method for next
void next(Node n) {
  next = n;
  next_is_initialized = true;
}

// Get method for next
Node next() {
  if (!next_is_initialized) {
    next_is_initialized = true;

    // Non-deterministic choice based on the number of visited nodes
    int i = Search.choose(visitedNodes.size() + 2);

    // Return the original value and
    // add the node object to the visited node set
    if (i == 0)  {
      if (next != null)
        if (visitedNodes.add(next))
          nonVisitedNodes.remove(next);
    }
    // Assign null to next
    else if (i == 1) {
      if (next == null)
        Search.backtrack();
      next = null;
    }
    // Assign an already visited node to next
    else if (i > 1 && i < visitedNodes.size() + 2) {
      Node temp = getVisitedNodeAt(i - 2);
      if (next == temp)
        Search.backtrack();
      next = temp;
    }
    // Assign a new non-visited node to next
    else if (i == visitedNodes.size() + 2) {
      Node temp = getANonVisitedNode();
      if (temp != null) {
        next = temp;
        visitedNodes.add(next);
        nonVisitedNodes.remove(next);
      }
    }
    else
      Search.backtrack();
  }
  return next;
}
```

Figure 4.6: The added accessor methods for `next`.

to cover different program paths. To allow covering the two branches of a conditional statement, Juzi replaces conditional statements with method invocations on `SymbolicInt` variables. These methods perform non-deterministic Boolean choices and consider both results of a conditional statement. For example, the `ifNEQ` method in Figure 4.5 compares two symbolic integers and considers the two cases for equality and inequality. To keep track of the path conditions generated using symbolic execution, Juzi uses the `PathCondition` class described earlier and that is updated with every non-deterministic choice. At the end of each program path, the repair algorithm checks the satisfiability of the path condition. If the path condition is satisfiable, it solves the path condition and assigns values to the symbolic variables.

4.3 Configurations

We next describe the different ways the user can configure the repair algorithm.

4.3.1 Controlling the Fields to Repair

Juzi provides a configuration file for the user to specify what classes to instrument and what fields to repair. This feature allows the user to add more constraints on the repair algorithm, which may be needed in some cases. For example, when the structure needs to have a certain number of nodes, the user can specify not to repair the `size` field and keep it concrete rather than symbolic. In this case, Juzi cannot modify the `size` field to satisfy other constraints and if the size property is not satisfied, it reports the structure as non-repairable. We have already used this feature while running the experiments in Section 3.6.1 where we configured Juzi to treat the `size` field concretely to study the effect of the fault location on the performance of the repair algorithm.

4.3.2 Controlling Data Repair

Data repair is a challenging problem. To illustrate, consider the problem of repairing the data elements of a binary search tree. While re-ordering the data elements to satisfy the search order may be a good choice for repair, the repair algorithm may set new values to the data elements and declare the tree as repaired. Juzi gives the user some control on how to repair the data. (1) The user can specify domains of values for data fields. Juzi will then use these domains while repairing data corruptions. (2) The user can select which type of constraint solver to use for solving the path conditions. Constraints on the order of data can be solved using a difference constraint solver which reorders the data elements without mutating the values. Other complex constraints require more complex solvers to repair the data values. (3) The user can configure Juzi to only use the values present in the data structure to repair the structure. For a primitive type T, the repair algorithm computes $domain(T)$ by traversing the given (corrupt) structure, collecting all values of type T encountered during the traversal, and re-using these values to solve the data constraints.

4.3.3 Abstractions

To assist the users in understanding the mutations performed during repair, Juzi provides them an abstraction of the performed repair actions, if they so desire. The abstraction specifies the set of fields that are mutated. Such information can help the users debug their program (if the corrupt structure was a result of a bug in the program). Moreover, Juzi also provides the users with a pair of abstract values that represent the structure before and after repair. The users can choose to provide their own abstraction functions if they like. As a default, Juzi provides a function, α, that counts the number of values of each type reachable from root o: $\alpha(o) = \{\langle n, T\rangle | n \text{ is number of values of type } T\}$. Juzi also reports a difference report between the structure state before repair and after repair which indicates the fields that have been mutated along with their original and new values.

4.3.4 Visualization

Juzi can be configured to visualize the repair process to provide a graphical representation of the structure before and after repair as well as the steps taken to go from the initial (corrupt) to the repaired structure. This feature (1) helps users understand the behavior of the repair algorithm, and (2) provides a debugging tool as it visually portrays the repair process suggesting ideas as to what might have gone wrong during the program execution. Moreover, the visualization also assists the user in finding bugs in `repOk` itself and guides the writing of `repOk` methods.

Juzi uses the Jung network layout framework to visualize the data structures [82]. Jung provides a generic graph class and a customizable utility package

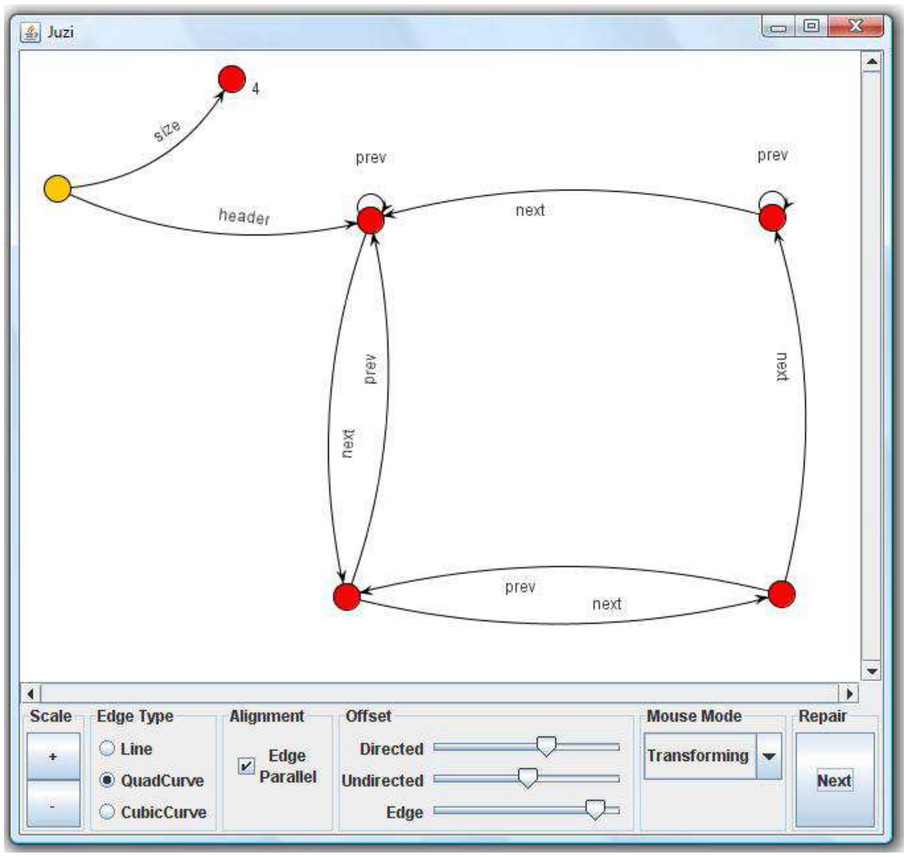


Figure 4.7: Example of the Juzi visualization module.

for displaying the graphs. Upon an assertion violation, Juzi translates the corrupt data structure into a Jung graph and displays it. The display GUI is interactive and allows the user to manually go through the repair actions one step at a time. Once the structure is repaired the program proceeds with its execution using the repaired structure. Figure 4.7 shows an example of the visualization provided by Juzi for repairing a doubly linked list with four nodes. By clicking the next button the user can iterate through the repair actions.

Chapter 5

Optimizations

This chapter presents two key optimizations for the core repair algorithm presented in Chapter 3. These optimizations target the search exploration strategy and the backtracking engine, and aim at enhancing the performance of the repair algorithm to repair larger structures with more faults. The first optimization utilizes a static analysis of `repOk` to capture information about the target data structure and uses the information to guide the search to the more likely candidates to repair faulty fields. The second optimization devises a *checkpoint-based* backtracking engine that enables the search to efficiently backtrack to the next candidate structure without having to rebuild the structure state.

5.1 STARC: Static Analysis for Repairing Complex Data

We presents STARC, a static analysis for efficient and effective repair of large data structures. Similar to the original repair approach, STARC systematically explores a neighborhood of the given corrupt structure using a backtracking search [23, 46, 56, 59] and performs *repair actions*, to transform the structure into one that satisfies the desired assertion. STARC, however, draws its key strength from a static analysis that enables it to perform repair actions that are more likely to

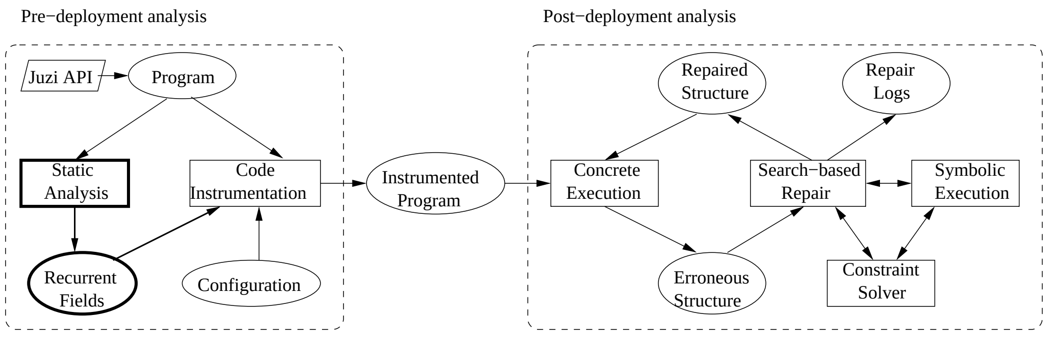


Figure 5.1: Static analysis for repair.

correct the corruption. The key idea behind STARC is as follows: `repOk` describes the constraints that the corrupt structure must satisfy. Statically analyzing `repOk`'s implementation enables extracting information about the target data structure that can help guide the search during repair.

Given a `repOk` method, the static analysis identifies two key characteristics. One, it identifies a set of *recurrent fields* [16], i.e., fields that `repOk` primarily uses to traverse its input structure. Two, it identifies a set of *local field constraints*, i.e., how the value of an object field is related to the value of a neighboring object field. STARC uses the result of the static analysis to (1) prioritize the order of repair actions based on the role of the corrupt field (recurrent or not), which makes efficient local repairs of corrupt fields and (2) monitor field accesses based on their relationship with their neighboring fields, which enables effective pruning of the search space. Figure 5.1 shows STARC as an extension of the repair framework described in Chapter 4. The bold components represent the new optimizations. These optimizations enables STARC to feasibly repairs faulty structures with tens

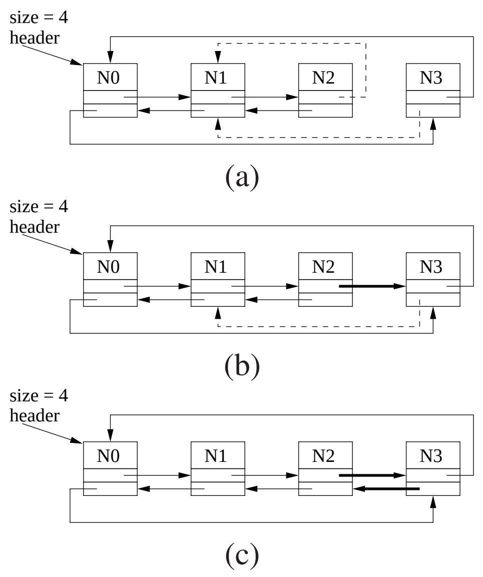


Figure 5.2: Repairing a circular doubly linked list using STARC.

of thousands of nodes—up to ten times larger than those possible using the original repair algorithm.

5.1.1 Illustrative Example

This section illustrates STARC using the doubly linked list example. It illustrates how STARC uses the static analysis to improve the performance of repair.

Consider the `DoublyLinkedList` class and the `repOk` method described in Section 3.1.1. To repair a doubly linked list, STARC first analyzes `repOk` and detects that the `next` field is the field used to traverse the doubly linked list, i.e., the recurrent field of the structure, and that the `prev` field is always equal to the transpose of the `next` field. Then given an erroneous structure STARC orders the repair actions according to the role of the field being repaired and guides the search algorithm to the choices that are more likely to repair the corruptions.

To illustrate, consider repairing the `DoublyLinkedList` instance in Figure 5.2 (a). Figures 5.2 (b-c) show the mutations performed by STARC to repair the corrupt structure. STARC uses the static analysis result to prioritize the order of the mutations on the corrupt fields. For a recurrent field, STARC gives a higher priority for selecting a non-visited node over a visited one or `null`, since recurrent fields are used for traversal and are highly likely to point to a new node. For this example, STARC first sets the `next` field of node `N2` to the non-visited node `N3`, and thus, repairs the `next` field in one try. STARC then utilizes the information about the transpose relation between the `prev` and the `next` fields to repair faults in the `prev` field. STARC directly sets the `prev` field of node `N3` to node `N2`, and repairs the structure.

In comparison with the original repair algorithm, instead of performing seven mutations to repair the list the optimizations enable STARC to repair the structure using only two mutations.

5.1.2 Detecting the Recurrent Fields of a Structure

The performance of the repair algorithm is highly dependent on the number of *repair actions* that are required to repair a structure. To scale the performance of the search algorithm, STARC first implements a static analyzer that detects the recurrent fields of the data structure. A key observation behind finding the recurrent fields is that such fields satisfy the reachability constraint of the structure. A recurrent field is more likely to point a non-visited object rather than a visited one.

Cahoon and McKinley proposed a data flow analysis framework for detecting the recurrent fields for prefetching of linked structures [15, 16]. We use this analysis to prioritize the repair actions. The problem is modeled as a forward data flow analysis problem. We first define some terms that we use to describe the components of the framework, then we describe the data-flow framework and illustrate how STARC uses the recurrent field information to prioritize repair actions.

5.1.2.1 Terminology

We define the following terms:

- *Information unit (IU)*: An information unit is the left hand side of an assignment operation on objects or object fields. An information unit is used to save and propagate information in the data flow framework.
- *Object field (F)*: An object field is a reference field in the data structure.
- *Recurrent status (RS)*: The recurrent status of an object field can have one of three values: non_recurrent (nr), possibly_recurrent (pr), and recurrent (r), where the elements are ordered such that $nr \leq pr \leq r$.
- *Information site (IS)*: An information site is a program statement of interest.

For example, consider the `repOk` method for the `DoublyLinkedList` class (Section 3.1.1). The information units are the local variables `visited`, `current`, `n`, and the implicit variable `this`. The object fields are the `header` field of the variable `this`, and the `next` and `prev` fields of the variables `current` and `n`. The information sites are lines `L2, L4, L6, L17,` and `L0`, the entry of the method.

5.1.2.2 Data-flow Framework

The basic data unit in the data flow framework is the *information tuple* T:

$$T \subset (IU \times F \times IS \times RS)$$

The data flow framework includes:

Initialization: When initializing an information tuple, all the components of the tuple are initialized to the bottom element of their lattices. For an information unit, iu, the bottom element is iu, for an object field, the bottom element is `null`, for an information site, the bottom element is `L0`, and for the recurrent status, the bottom element is *nr*.

The data-flow functions: The propagation of information in the framework occurs at the information sites. We consider two types of information patterns: equality patterns and access patterns. Given an input set of information tuples, Rin, we compute:

- **Equality patterns**: ⟨*information unit*⟩ = ⟨*information unit*⟩′

$$\begin{aligned} GEN(iu = iu', Rin) &= \{(iu, f, is, rs) | (iu', f, is, rs) \in Rin\} \\ KILL(iu = iu', Rin) &= \{(iu, f, is, rs)\} \end{aligned}$$

- **Access patterns**: ⟨*information unit*⟩ = ⟨*information unit*⟩′.⟨*object field*⟩

$$GEN(iu = iu'.f, Rin) = \left\{ \begin{array}{l} \text{if } \{(iu', null, L0, nr)\} \in Rin \\ \qquad (iu, f, is, pr) \\ \text{if } \{(iu, f, is, pr)\} \in Rin \\ \qquad (iu, f, is, r) \end{array} \right\}$$

$$KILL(iu = iu'.f, Rin) = \{(iu, f, is, pr), (iu, null, L0, nr)\}$$

The meet operation: The meet operation ($\sqcup$) is defined on sets of tuples. Given two sets T1 and T2, the meet operation is defined as follows:

$$\begin{aligned} T1 \sqcup T2 = {} & \{t | t \in T1 \wedge t \notin T2\} \cup \{t | t \notin T1 \wedge t \in T2\} \\ & \cup \{(iu, f, is, rs_1 \sqcup rs_2) | (iu, f, is, rs_1) \in T1 \\ & \wedge (iu, f, is, rs_2) \in T2\} \end{aligned}$$

The transfer functions: The transfer functions are:

$$A_{in}(ip) = \bigsqcup_{p \in pred(ip)} A_{out}(p)$$

$$A_{out}(ip) = (A_{in}(ip)/KILL(ip, A_{in}(ip))) \sqcup GEN(ip, A_{in}(ip))$$

Starting at the entry of the analyzed method (`repOk`), all the tuples are initialized. The algorithm proceeds by propagating information and iterating until a fixed point is reached. To illustrate, the first three iterations of the data flow analysis for the `DoublyLinkedList`'s `repOk` are displayed in Table 5.1. The bold tuples indicate an update in the tuple information during successive iterations. The tuples for the information units `this` and `visited` are never updated and thus they are omitted for brevity. The framework converges in the fourth iteration.

stmt	RA	Iteration 1	Iteration 2	Iteration 3
L4	in	(current, null, L0, nr) (n, null, L0, nr)	(current, null, L0, nr) (n, null, L0, nr)	(current, null, L0, nr) (n, null, L0, nr)
L4	out	**(current, header, L4, pr)** (n, null, L0, nr)	(current, header, L4, pr) (n, null, L0, nr)	(current, header, L4, pr) (n, null, L0, nr)
L6	in	(current, header, L4, pr) (current, null, L0, nr) (n, null, L0, nr) **(n, next, L6, pr)**	(current, header, L4, pr) **(current, next, L17, pr)** (n, null, L0, nr) **(n, next, L6, r)**	(current, header, L4, pr) **(current, next, L17, r)** (n, null, L0, nr) (n, next, L6, r)
L6	out	(current, header, L4, pr) (current, null, L0, nr) **(n, next, L6, pr)**	(current, header, L4, pr) (current, next, L17, pr) **(n, next, L6, r)**	(current, header, L4, pr) (current, next, L17, r) (n, next, L6, r)
L17	in	(current, header, L4, pr) (current, null, L0, nr) (n, next, L6, pr)	(current, header, L4, pr) (current, next, L17, pr) (n, next, L6, r)	(current, header, L4, pr) (current, next, L17, r) (n, next, L6, r)
L17	out	**(current, next, L17, pr)** (n, next, L6, pr)	**(current, next, L17, r)** (n, next, L6, r)	(current, next, L17, r) (n, next, L6, r)

Table 5.1: The first three iterations of the data flow framework.

At the end of the analysis, the object fields in the tuples that have a recurrent status r are considered the recurrent fields as used by `repOk`. For example, as expected, in Table 5.1 all the tuples that have r as a recurrent status have `next` as an object field. Thus, `next` is the field used for traversing the list in the `repOk` method of the `DoublyLinkedList` class. The `prev` field is not reported by the analysis as a recurrent field since it is not used by `repOk` to traverse the structure.

Note that interprocedural analysis is performed similarly. At the call site, information is propagated to the entry of the called method by following a set of *equality patterns* for each argument in the method signature. At the return site, information is propagated from the called method to the caller by following an equality pattern at the caller side. Details about the interprocedural analysis are found elsewhere [16].

5.1.2.3 Prioritizing Repair Actions

STARC uses the information about the reference fields to prioritize the candidates for repairing the structure fields. The repair algorithm follows the same search pattern when taking repair actions to fix an error in a reference field (Section 3.3). The recurrent fields of a linked data structure are used to traverse the structure starting from a given root node. For traversing a structure, recurrent fields are more likely to point to new (non-visited) nodes or `null` rather than pointing to previously visited nodes. STARC orders its repair candidates based on the type of the faulty field (recurrent or not). For recurrent fields, STARC gives higher priority for choosing a new (non-visited) candidate over choosing a visited one or `null`. For the non-recurrent fields, STARC chooses the same order presented in Section 3.3; STARC gives higher priority to choosing a visited node over a new node. This optimization not only improves performance (Section 5.1.5) but also guarantees that the reachability of the structure is preserved by repair.

5.1.3 Detecting Constraints on References

Structural properties often constrain aliasing possibilities, e.g., $o.f == p \Leftrightarrow p.g == o$ for objects o and p, and fields f and g. Solving such constraints can be efficiently performed symbolically without enumerating the search space.

STARC implements a static constraint solver that repairs particular fields instantaneously; without triggering the search algorithm. Some of the imperative constraints on reference fields take the following pattern:

```
if (iu != iu') {
    ...
    return false;
}
```

For example, the transpose relation between the `next` and `prev` fields of the `DoublyLinkedList` class takes the following form:

```
Node n = current.next;
if (n.prev != current) {
    return false;
}
```

The solution of such constraints is embedded in the negation of the condition. STARC performs static analysis on the control flow graph of the `repOk` method to detect these patterns. Once these patterns are detected, the solver injects the solution of the constraints into the `repOk` method.

All the analysis that STARC performs is at the Java bytecode level. To detect patterns in a method, STARC builds the control flow graph (CFG) and searches for basic blocks where the entry instruction is a conditional branch and the exit instruction is an integer return. To detect the items being compared in the conditional statement, STARC uses the JVM specification [80] to trace the last two items produced on the stack. For example, consider the bytecode example of the transpose constraint of the `DoublyLinkedList` as described in the `repOk` method in Section 3.1.1:

```
// Compare prev to current
42: aload_3
43: getfield #32; //Field DoublyLinkedList\$Node.prev;
46: aload_2
47: if_acmpeq 52
// return false
50: iconst_0
51: ireturn
```

STARC detects the parameters of the conditional statement by following the consumer/producer chain of the previous instruction until two items are produced in the stack. In the above example, the instructions used to produce the comparison objects are:

```
42: aload_3        // consume: 0   produce: 1
43: getfield #32; // consume: 1   produce: 1
and
46: aload_2        // consume: 0   produce: 1
```

These instructions are then used to produce the solution for the constraint and add the solution to the bytecode as follows:

```
42: aload_3
43: getfield #32; //Field DoublyLinkedList\$Node.prev;
46: aload_2
47: if_acmpeq 57
// set the field of prev to current
51: aload_3
52: aload_2
53: putfield #32; //Field DoublyLinkedList\$Node.prev;
```

Using this solver, STARC identifies equality constraints and directly solves such constraints without using any non-deterministic search. This optimization enables highly efficient solving of a variety of *local* constraints. To illustrate, STARC automatically detects the transpose relation of the `DoublyLinkedList` and fixes any violation in the `prev` field by setting it to the transpose of its predecessor's `next` field.

5.1.4 Characteristics

We next discuss some characteristics of the STARC framework.

5.1.4.1 Completeness of the Repair Algorithm

The original repair algorithm uses a systematic search that explores all the non-isomorphic structures that satisfy the integrity constraints. Thus, if there exists a structure that satisfies the integrity constraints, the algorithm will find it. We point out that the optimizations added in STARC do not affect the completeness of the original algorithm. Using the recurrent field information, STARC only changes the order of the search and does not skip any valid structure from being explored. The reference constraint solver statically detects and solves constraints that are not yet initialized by the search algorithm. Thus, the instrumentation of `repOk` does not affect its behavior.

5.1.4.2 Reachability of the Repaired Structure

An important characteristic of STARC is that it solves two problems: the structural constraints as described in `repOk` and the reachability of the original structure nodes. Recall that STARC prioritizes the order of choices according to the type of the faulty field. Using the recurrent analysis information, the recurrent fields are assigned to new non-visited nodes. Thus, STARC first solves the reachability problem and then satisfies `repOk`. This feature is not present in the original repair algorithm, which usually finds the first structure that solves the constraints disregarding the original size and the number of nodes reachable from the root of the structure. For example, as discussed in Section 3.6.1, a faulty doubly linked list of 200 nodes may be repaired into a list with 100 nodes that satisfies the structural integrity constraints. Since STARC's algorithm is complete, and prioritizes reachability, it first tries to find a solution with all reachable nodes. If none exists, it will satisfy `repOk` with a smaller structure, if possible.

5.1.5 Preliminary Evaluation

This section presents a preliminary evaluation of STARC by applying it to faulty implementations of the three subject structures used to evaluate the original

Subject Structure	*Size*	*# of faults*	*Time (ms)*		*# of repair actions*	
			core	*STARC*	*core*	*STARC*
Singly linked list	1,000	1	23	21	1	1
	10,000	1	178	148	1	1
	100,000	1	1,648	1,581	1	1
Doubly linked list	100	10	516	16	1,525	85
	1,000	10	212,203	152	539,300	749
		100	$\geq \tau$	382	$\geq \delta$	992
	10,000	10	$\geq \tau$	14,925	$\geq \delta$	4,022
		100	$\geq \tau$	23,778	$\geq \delta$	9,710
Binary search tree	100	10	51	60	113	113
	1,000	10	153	160	443	443
		100	2,363	2,422	9,554	9,554
	10,000	10	15,435	15,711	5,618	5,618
		100	161,110	160,845	67,017	67,017

Table 5.2: Results for repairing data structures using STARC.

repair algorithm. This evaluation demonstrates the efficiency of the optimizations integrated in STARC. We report the average repair time and the number of repair actions required to fix the errors and compare the results of STARC with those of the original repair algorithm. We set a threshold time of ten minutes to repair a faulty structure, and stop the execution after that period.

Table 5.2 displays the repair time and the number of repair actions taken by the original repair algorithm and STARC to repair the three subject structures; τ represents a time threshold of 10 minutes, and δ represents a threshold one million repair actions. Singly linked list has the simplest of the constraints and the least number of faults and its repair is therefore the fastest. For the binary search tree where acyclicity is the only constraint, the performance of both approaches is indistinguishable since breaking cycles is achieved by setting the value of the corrupt field to `null`.

The doubly linked list results show that for solving constraints like transpose and circularity, STARC outperforms the original repair algorithm by more than two orders of magnitude. The original algorithm did not finish the execution within the ten minutes threshold when repairing a doubly linked list with 1,000 nodes and 100 faults, whereas STARC was able to repair a doubly linked list with 10,000 nodes in less than thirty seconds. Note that although the repair algorithm is complete for both approaches, the former took 539,300 repair actions to repair ten faults in a doubly linked list of size 1,000 whereas the latter only took 749 actions. The static analyzer in STARC bias the repair algorithm toward solving the reachability constraint while repairing a recurrent field. The original repair algorithm on the other hand repairs the faults in the structure, yet the repaired structure might not satisfy the reachability constraint, thus it keeps searching for a structure that satisfies all the constraints. STARC also enables overcoming the disadvantage of the original algorithm when maintaining the reachability of the elements in the data structure. When using STARC, there is no need to configure the `size` field as `un-modifiable` by the repair algorithm.

Note however that STARC still requires to traverse the data structure for every repair action since it uses a re-execution-based backtracking approach. The next section presents an efficient backtracking that enables performing mutations on the structure without having to recreate its state by re-executing `repOk` from the beginning.

5.2 Checkpoint-based Backtracking for Efficient Repair

We present a novel backtracking approach that significantly improves the efficiency of the core repair algorithm, and in general systematic search engines. Specifically, most existing approaches, including the one we used in the repair algorithm, use backtracking through code re-execution to explore their search space. For example, the core repair algorithm (Figure 3.6) explores the state space of `repOk` using a backtracking search where program states are not stored to enable backtracking; instead, the state at a backtracking control point is re-created by re-executing `repOk` from the beginning and monitoring its execution.

An alternative to code-execution is performing stateful searches, such as those used in some model checkers, e.g., SPIN [59] and Java Pathfinder(JPF) [114], which store (hash) program states and retrieve them for backtracking. Both approaches have complementary strengths and traditionally model checkers are based on one of the two approaches [23, 46, 59, 114].

In contrast to these approaches, our approach uses a checkpoint-based backtracking that employs efficient state manipulations based on selective storing of program components, and *abstract undo operations* for retrieving the program state. Our approach is based on two key insights: (1) `repOk` implementations check desired properties by traversing the given structures without mutating them; and (2) the traversals are over object graphs and often use standard worklist-based algorithms that track sets of visited nodes to prevent infinite traversals. The first insight allows us to define a minimal part of state to store, which reduces storage overhead. The second insight allows us to use our own library classes in place of the standard

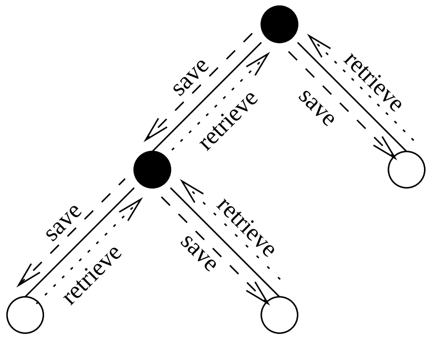


Figure 5.3: The stateful backtracking process.

Java libraries, such as sets and lists, that are commonly used in graph traversals; in contrast with the standard libraries that optimize program execution, our libraries optimize backtracking. Checkpoint-based backtracking combines the benefits of a re-execution-based search with those of a stateful search by avoiding rebuilding program states while at the same time not imposing the large overheads of state storage and retrieval.

There are two key requirements for stateful backtracking: a mechanism for switching the *execution control* to specific statements in a program, and an approach for storing and retrieving the *program state* at those statements. The performance is highly dependent on the efficiency of the aforementioned operations. For instance, our experiments with STARC show that the overhead imposed by a naive approach of saving and restoring that takes a snapshot of the heap at every choice point, is similar to that imposed by rebuilding the program state for most structures.

We next describe the backtracking algorithm. We first describe how to maintain the program state, i.e., the stack, static, and heap memory, and then describe how to efficiently maintain the program counter when backtracking.

5.2.1 State Storage and Retrieval Algorithm

Systematic search algorithms [10, 36, 72, 106] operate on *choice points*, i.e., program statements where non-deterministic choices are performed on the search variables, and *termination points*, i.e., program statements that specify the end of a search path or a choice. For instance, the choice points in STARC are the field access statements in `repOk` where the search variables are the fields of the structure, and the choices are the members of the field domains. The termination points are the `return` statements of `repOk` where a structure is declared as valid or not.

Backtracking occurs between a termination point and a choice point. To maintain the correctness of a program execution, a backtracking approach must save the program state at each choice point, and upon backtracking, must retrieve the saved state and proceed with the next choice. To illustrate, Figure 5.3 gives an abstraction of the search process. Black nodes represent choice points, and white nodes represent termination points. As the program executes (following the dashed arrows in Figure 5.3), the search algorithm saves the state between choice points. Once a termination point is reached, the search algorithm backtracks to the last choice point (following the dotted arrows in Figure 5.3), retrieves the saved state and proceeds with the next choice.

Several approaches exist for state storage/retrieval [74, 122]. A simple, yet expensive, approach for storing the state is by taking a snapshot of the heap at every choice point. This approach is (1) expensive regarding memory requirements, and

```
// An interface for undo commands
public interface UndoCommand {
    public void execute();
}

// Declaration of the stack used for storing the undo commands
Stack<UndoCommand> undoStack = new Stack<UndoCommand>();

// Store method for saving the undo commands at a choice point
public void store() {
    saveUndoStack(undoStack);
    undoStack = new Stack<UndoCommand>();
}

// Retrieve method for restoring the program state
// at the backtracking target
public void retrieve() {
    undoStack = getLastUndoStack();
    while (!undoStack.isEmpty()) {
        UndoCommand uc = undoStack.pop();
        uc.execute();
    }
}
```

Figure 5.4: Components for maintaining the program state.

(2) inefficient as it stores a lot of unnecessary redundant states. A more efficient approach for state storage is by using state comparisons [79, 117]. This approach efficiently hashes the heap at the first choice point, and then incrementally updates it by comparing the state at the current choice point with the stored one.

We propose an alternative approach for state storage/retrieval. Rather than taking a snapshot of the heap at the choice point, or performing state comparisons to update the snapshot, we incrementally store the program state as the changes occur during execution. To enable efficient state retrieval, along with every stored change, we save a corresponding *undo command* [44] that enables retrieving the original state when the command is executed. Undo commands are implementations of the "Command" design pattern [44] where each command object saves the necessary information for undoing the effect of an action performed on the object. Undo commands have been previously used in software model checkers [100]. However, our use of the undo commands is different. Rather than performing the operations at the concrete heap level, we introduce *abstract undo commands* which perform these operations at an abstract object level. We further describe this idea later in this section.

The search algorithm maintains the undo commands in a stack which we term "undo stack". As the program executes, the undo stack is populated with undo commands. At each choice point, the undo stack is saved. Upon backtracking, the last saved undo stack is retrieved, and its commands are executed to restore the state to the previous choice point. To illustrate, Figure 5.4 shows the implementation of the `store` method which is invoked at choice points, and the `retrieve` method which is invoked when backtracking. The `UndoCommand` interface defines the common behavior of all undo commands. The `undoStack` saves the undo commands that occur between two choice points in the program. The `store` method saves the undo stack and clears it, and the `retrieve` method retains the last undo stack and executes its commands to retrieve the program state.

5.2.1.1 Undo Commands

We next describe the undo commands. Undo commands are considered in the methods of interest for the search algorithm. For instance, in STARC, undo commands are inserted in `repOk` and any helper method invoked by `repOk` that accesses the target structure fields, i.e., contains choice points. Undo commands are inserted at method statements that cause a change in the program state. The statements of interest for inserting the undo commands are the following:

- store operations on the local variables,
- store operations on instance or static fields of a class, and
- method invocations.

We create undo commands that (1) save the original value of the modified object, and (2) enable retrieving the state of the modified object when the undo command is executed. We next describe the undo operation for each of above statements.

Local variable stores: Store operations on local variables are treated as field store operations. Since Java doesn't support pointer creations to elements in the JVM registers' stack, our approach replaces local variables with static fields and uses field undo commands (described in the next paragraph) to undo local changes. This approach adds some overhead as *`X`*`STORE` instructions which access the variables from the method's stack are replaced with `PUTSTATIC` instructions which access the variables from static memory. Note that our transformation replaces stack frames with static fields and as such cannot support recursive methods; to support recursion, our transformation would need to replace stack frames with (appropriately linked) heap objects.

Field stores: Field store operations are the simplest to save and undo. Before each field store operation, we create an undo command object that takes as input the field's owner object and the field's value. When the command is executed, it reassigns the field to the saved value. To illustrate, consider the example of the `repOk` method for the `DoublyLinkedList` class in Figure 5.5(a). The method has two fields to store, `n` and `current` (the `visited` variable is never reassigned and thus it is not saved). To store `current`, we push a new instance of the `CurrentUndoCommand` class (Figure 5.5(b)) which takes the list object and the field's value, onto the undo stack. Upon backtracking, when the `execute` method of the `CurrentUndoCommand` is invoked, the field `current` retrieves its old value. The field `n` is similarly stored. The example in Figure 5.5(b) describes the general implementation of an undo command to restore the value of a field. We point out, however, that there is no need to save the owner object when restoring static fields.

Note that executing the undo commands upon backtracking restores the heap, static, and stack memory since local variables are transformed into static fields.

Method invocations: A straightforward way to handle method invocations is to instrument the invoked method's code and add undo commands before changes to its local variables and fields accesses. We use this approach on `repOk` (the method of interest of STARC's search algorithm) and any helper method invoked

```
// A static field to replace the local variable current
public static Node current;

// RepOk method with undo commands, and undoable containers
public boolean repOk() {
    ...
    // The HashSet is replaced with an undoable hash set
    Set<Node> visited = new UndoableHashSet<Node>(undoStack);
    ...

    while (true) {
        ...

        // Undo command added to retrieve the value of current
        undoStack.push(new CurrentUndoCommand(this, current));
        current = n;
    }
    ...
}
```

(a)

```
// The undo command for field accesses
public class CurrentUndoCommand implements UndoCommand {
    DoublyLinkedList list;
    Node value;

    public CurrentUndoCommand(DoublyLinkedList list, Node value) {
        this.list = list;
        this.value = value;
    }

    public void execute() {
        list.current = value;
    }
}
```

(b)

Figure 5.5: An undo command class to restore the value of the variable `current` in `repOk`.

by `repOk` which contains choice points. However, we treat other method invocations differently depending on the type of the method, its effect on the caller object, and the type of its caller object. We first check if the method is *pure*, i.e., does not mutate the state of its caller, and if so, there is no need to instrument the method's code. We then check the method's caller object type. If the caller object's type is a container type, i.e., its class implements the `java.util.Collection` or the `java.util.Map interfaces`, we use *abstract undo* commands to reverse the effect of the method on the container (Section 5.2.1.2). If the method's caller object type is not a container, we use the straightforward approach, i.e., instrument the method and add undo operations on its field accesses.

5.2.1.2 Abstract Undo Operations

Container types are widely used in Java programs. For example, `repOk` predicates are typically implemented as standard work-list algorithms that traverse the object graph, keep track of visited nodes, and check for the validity of the structural integrity constraints [85]. Collection classes provide powerful utilities for performing such traversals and checks, for example, a `LinkedList` object can be used for the work-list and a `HashSet` object can be used for saving the visited items. These classes maintain complex data structures to enable efficient operations, such as adding, removing, or checking the occurrence of an element. This makes it expensive to store and retrieve their states using standard approaches. To illustrate, a snapshot approach requires iterating over the container elements at each choice point to save the state. A state comparison approach requires traversing the

```
// A snippet of the UndoableHashSet class
public class UndoableHashSet<T> implements Set<T> {
    Stack<UndoCommand> undoStack;
    Set<T> container;
    ....

    public boolean add(T e) {
        if (container.add(e)) {
           undoStack.push(new AddUndoCommand<T>(container, e));
           return true;
        }
        return false;
    }
}
                          (a)

// Implementation for the abstract add undo command
public class AddUndoCommand<T> implements UndoCommand {
    Set<T> container;
    T val;

    public AddUndoCommand(Set<T> container, T val) {
        this.container = container;
        this.val = val;
    }

    public void execute() {
        container.remove(val);
    }
}
                          (b)
```

Figure 5.6: Abstract undo commands on sets.

container to perform state comparisons. Even the undo approach that we presented in the previous section may be expensive due to the complex implementation of the operations on containers. For example, a `HashSet` implementation uses a `HashMap` which saves its elements in an internal array. Therefore adding undo commands for all the internal state changes involves several operations, especially for operations that dynamically resize the containers.

We present an efficient way for undoing changes on containers. We perform the undo operations at the abstract level of the container rather than at the concrete container implementation. For example, instead of adding field undo commands in the implementation of the `addFirst` method of a `LinkedList` class, we add one undo command that reverses the effect of the `addFirst` method, i.e., the undo command calls the `removeFirst` method on the `LinkedList` object.

To apply this abstraction, we implement undoable versions of the container classes and replace all the instances of the concrete versions with the new ones, e.g., the `visited` variable in the `repOk` method in Figure 5.5(a). The undoable versions are simple adapters for the original containers where the methods' implementations push the appropriate undo command to the program undo stack. To illustrate, consider the code snippet of the `UndoableHashSet` class in Figure 5.6(a). The add method of this class adds an object to the internal wrapped `HashSet` object. If the `add` operation is successful, an `AddUndoCommand` object is created and pushed onto the undo stack. The implementation of the `AddUndoCommand` class is displayed in Figure 5.6(b). Instances of the class are constructed using the container and the object added to the container. The execute method simply removes the added object from the container.

Abstract undo operations achieve their efficiency by providing a way to undo the effect of complex operations that are frequently invoked and that involve large state changes.

5.2.2 Monitoring the Program Counter

We next describe how to maintain the program counter and change its value between choice points to automatically switch the program control without special JVM support.

We start by identifying the *backtracking sources*, i.e., the program statements to backtrack from, and the *backtracking targets*, i.e., program statements to backtrack to. We then instrument the program to enable branching from the sources to the targets while restoring the state of the program at those targets.

The backtracking sources are the termination points of the program. For instance, the return statements in `repOk`. The backtracking targets are the choice points of the program.

To enable efficient backtracking, we instrument the method under analysis, e.g., `repOk`, by adding labels at the backtracking targets, and `TABLESWITCH` instructions at the backtracking sources. The branch targets for the `TABLESWITCH` instructions are the labels inserted before the backtracking targets. The `TABLESWITCH` condition checks an integer value returned by search algorithm that identifies the label of the target choice point (this information is already maintained by the search algorithm). Note that this is a non-trivial use of table switches as the targets of the `TABLESWITCH` instructions occur at arbitrary points in the method code.

At the backtracking sources and targets, we also add a call to the `retrieve` and `store` methods described in Section 5.2.1 to maintain the program state when backtracking.

```
public boolean repOk() {
// L0:
//   store();
     Node header = getHeader();

// L1:
//   store();
     int size = getSize();

     System.out.println(header + "  " + size);

//   retrieve();
//   int index = Search.getTargetId();
//   TABLESWITCH \\(index)
//        0 : L0
//        1 : L1
//        2 : L2

// L2:
     return true;
}
```

Figure 5.7: An example of the backtracking implementation.

To illustrate the backtracking approach, consider the example in Figure 5.7 of a simplified `repOk` method that accesses two fields from the `DoublyLinkedList` and always returns `true`. The instructions added by the instrumentation are displayed in the commented portion of the code. Our use of the `TABLESWITCH` statements cannot be expressed in Java source and therefore, they are expressed in Java bytecode. The method in Figure 5.7 is simple and does not require adding undo commands.

The code example has two choice points. A label is added (L0 and L1) before each choice point, as well as a call to the `store` method which is used to save any undo commands performed before the choice point (in this case none). The added labels are the backtrack targets.

The method has one backtracking source which is the `return` statement. Before this statement, a label is added (L2) in addition to a call to the `retrieve` method which is used to execute the saved undo commands. A `TABLESWITCH` is also added before the return statement. The branching labels are L0 and L1, with the default label L2. For illustration, the domain of values we use in this example are [null, N0] for the `header` field, and [0, 1] for the `size` field. The output of executing `repOk` is as follows:

```
null  0
null  1
N0  0
N0  1
```

The execution works as follows. The first pass on `repOk` assigns `header` and `size` to `null` and 0 respectively. Before the method returns, the search algorithm returns 1 as the `id` for the last choice point, and the `TABLESWITCH` branches to label L1, assigns `size` to 1, and prints the values. At the next encounter of the return statement, the search algorithm returns 0 as the last field initialization since all the values in the `size` field domain are considered. The program backtracks to label L0, assigns `header` to `N0`, assigns `size` to 0, prints the values and so on. When all the choices are considered, the search algorithm returns 2 as the branch target, which causes the `TABLESWITCH` to branch to label L2, and the method's execution then terminates.

The above discussion illustrated backtracking within a single `repOk` method.

However, normally as the complexity of the structural constraints increases, it is typical to represent the class invariant as multiple small helper methods with one executive `repOk` method that invokes the helper methods. Such cases might require backtracking to choice points that reside in the helper methods from the return statements in `repOk`. To handle such scenarios, the call sites of the helper methods are considered backtracking targets, and `TABLESWITCH` statements are added at the entry points of the helper methods to enable branching to the destination choice point. Upon backtracking from `repOk`, the control point is changed to the helper method's call site, the method is invoked, and then the `TABLESWITCH` at the entry of the method directs the control to the target choice point. Note that there is no need to restore the local variables at the target choice point, since restoring the values is automatically handled by executing the undo commands.

The described backtracking mechanism adds minimal overhead since it primarily adds table switches at method entries and return statements. Backtracking within `repOk` requires one switch, while backtracking for the cases of helper methods, requires two switches per invocation to reach the target choice point.

5.2.3 Characteristics

We discuss some characteristics of the checkpoint-based approach and address its limitations.

5.2.3.1 Overhead of the Checkpoint-based Backtracking

The checkpoint-based approach removes the overhead of rebuilding the program state from scratch after each backtracking operation (as in re-execution-based backtracking). This overhead includes re-initializing the object fields in every iteration of the search algorithm. However, it introduces the overhead of maintaining the program state by saving and executing the undo commands. Our experiments with the new backtracking engine (Section 5.2.4) show that the checkpoint-based approach reduces the number of field initializations performed in STARC, while introducing a set of undo commands. The number of such commands, however, is relatively an order of magnitude less than the reduction in field initializations, resulting in faster generation time.

A key reason for this improvement relates to the nature of the `repOk` methods used by STARC to build and explore the search space. Such methods are typically pure methods, i.e., they check for the structural properties without mutating the structure. Thus, we expect state changes between 2 consecutive choice points to be minimal, which results in less undo operations to retrieve the state and in turn a better performance than code re-execution.

5.2.3.2 Soundness of the Approach

The search presented in this chapter is purely performed through code instrumentation of the class under analysis. This entails some modifications in the structure of the class, including adding fields to replace local variables when performing undo operations. Such modifications may affect the soundness of the approach on some Java programs. For example, consider a `repOk` method that reflectively accesses the fields of its declaring class. Such method might have a different behavior because of the changes performed on the structure of the class.

Other factors that might break the soundness of the approach are the abstract undo operations. These operations might not be equivalent to the exact inverse of the corresponding forward operations. While executing these commands reverses the effect at the abstract container level, the internal structure of the container might have changed. For example, adding and removing methods in a balanced tree may involve some reordering operations that result in a different structure. A `repOk` method that accesses the internal implementation of the container may have a different behavior after running the undo operations.

While the described scenarios may break the soundness of repair, we do not expect these cases to happen in practice. For example, we never encountered or wrote a `repOk` method that reflectively reasons about its own class, or uses the implementations of the container libraries rather than the well defined interfaces.

5.2.3.3 Abstract Analysis on Containers

Although presented in the context of a Java implementation, the proposed technique is not limited to Java or its containers. Undo operations can be applied to different languages and on any (well-specified) container written in that language. For example, similar containers can be implemented for the C++ standard library.

We believe that extending current program analyses to handle libraries opens more opportunities for reasoning about programs. For example, *abstract symbolic*

Subject structure	*Size*	*Faults*	*Time(ms)*		*Field initializations*		*Undo*
			STARC	*Checkpoint*	*STARC*	*Checkpoint*	*operations*
Doubly linked list	1,000	100	382	31	482,544	1,872	1,988
	10,000	100	23,778	182	32,118,651	19,210	18,418
	100,000	100	$\geq \tau$	1,142	-	199,808	199,614
	1,000,000	100	$\geq \tau$	8,640	-	1,999,995	1,999,988
Binary search tree	1,000	100	2,391	121	8,126,303	12,552	9,529
	10,000	100	1 60,648	937	228,499,900	97,015	67,051
	100,000	100	$\geq \tau$	3,184	-	331,607	629,673
	1,000,000	100	$\geq \tau$	32,009	-	5,922,174	2,922,274

Table 5.3: Results for repairing two structures with up to a million nodes and 100 faults.

execution has been previously introduced in a workshop paper [73] which treats containers as symbolic objects. By treating containers symbolically, the approach was able to test programs that manipulate such containers, an analysis that was not feasible if the implementation of the container was to be considered.

5.2.4 Performance Improvement

We evaluate checkpoint-based backtracking by integrating it in STARC and using it to repair data structures with up to a million nodes. To demonstrate the efficiency, we compare the repair time taken by the checkpoint-based approach with that taken by the re-execution-based approach originally used in STARC. We use STARC to refer to the original algorithm. The checkpoint-based approach gains its efficiency by reducing the number of field initializations performed by re-executing `repOk` on the structure after each repair action. We compare the number of field initializations performed by the checkpoint-based approach with those performed by STARC. Additionally, we report the number of undo operations, which represents the overhead of the checkpoint-based approach to maintain the program state.

Table 5.3 shows the repair results. The table displays the repair time in milliseconds taken by STARC and the checkpoint-based approach for repairing corrupt binary search trees and doubly linked lists with up to a million nodes. The original re-execution-based search does not terminate in a threshold of ten minutes for repairing structures with a hundred thousand nodes and a hundred faults. The checkpoint-based backtracking enables repairing structures with up to a million nodes in less than 10 seconds for the doubly linked list and 33 seconds for the binary search tree. Moreover, when the re-execution-based backtracking terminates within the threshold time, the checkpoint-based backtracking achieves more than two orders of magnitude speedups when repairing the corrupt structures.

To study the speedups obtained by using the checkpoint-based backtracking approach, we perform a comparison between the number of field initializations performed by STARC and the checkpoint-based approach when repairing the structures. We also study the number of undo operations required by the checkpoint-based backtracking approach to maintain the program state. The field initialization results in Table 5.3 show that for the studied subjects the checkpoint-based approach reduces the number of field initializations required by STARC by more than two orders of magnitude, while the number of undo commands performed is less than an order of magnitude than the number of field initializations performed by the re-execution-based backtracking. For example, when repairing a doubly linked list with ten thousand nodes, the field initialization ratio is 1,617X while the number of undo commands is comparable to the number of field initializations performed by the checkpoint-based approach.

Note that the speedup factors increase with the size of the structure. For example, for the binary search tree example, the speedup factor increased from 19X when repairing a corrupt structure with 1,000 nodes to 171X when repairing a structure with 10,000 nodes. This increase in the speedup factor relates to the nature of the backtracking search used in STARC. The original search in STARC is re-execution-based and thus every mutation in the structure requires traversing the structure from the root to check the class invariant. As the size (number of faults) of the structure increases, such traversals become more expensive. The checkpoint-based approach, on the other hand, incrementally checks for the class invariant and requires a single traversal of the structure to perform all the mutations.

The experiment on repair presented in this section demonstrated that integrating the checkpoint-based approach in STARC scales its performance for repairing larger data structures more efficiently. Chapter 6 presents more empirical evidence of the efficiency of the optimizations on repairing a wider variety of data structures and stand-alone applications.

Chapter 6

Evaluation

This chapter presents an evaluation of assertion-based repair on repairing inconsistencies in data structures. Two types of experiments are presented: (1) experiments for measuring the efficiency of the repair algorithm, where we repair a diverse set of library data structures and (2) experiments for evaluating the effect of repair on running applications, where we repair inconsistencies in the data structures of three stand-alone applications. We next describe each of these experiments and discuss some interesting results.

6.1 Experiments on Library Data Structures

In chapters 3 and 5 we presented a preliminary evaluation of the repair algorithm and the subsequent optimizations on three subject data structures. The evaluation demonstrated the ability of the repair framework to efficiently repair large data structures that occur in real applications. In this section we extend the evaluation to a more diverse set of subjects with a wider spectrum of structural and data constraints. We start by describing the subject structures, and then we state the evaluation methodology and present the experimental results.

6.1.1 Benchmarks

The subjects we choose to evaluate repair are primarily textbook data structures [26] that are characterized by a set of *local* constraints, i.e., constraints relating objects in the structure to their neighboring objects, and *global* constraints, i.e., constraints relating all the objects of the structure. These structures are also characterized by a set of data constraints that include both structurally dependent constraints, i.e., constraints relating the data elements to structural properties, and structurally independent constraints, i.e., constraints relating the data elements of an object to the data elements of neighboring objects. The subject data structures are described below:

- **Binary tree.** A binary tree object has a `root` node and a `size` field caching the number of nodes in the tree. Each node in the tree has a `left` and a `right` child node in addition to a `parent` node. The `root` node has no parent. Integrity constraints include acyclicity along `left` and `right`, the transpose relation between parent and children nodes, and consistency of the `size` field with respect to the number of reachable nodes from the `root`.

- **Sorted linked list.** A sorted linked list is an acyclic linked list whose nodes have integer elements. Integrity constraints include acyclicity as well as ordering of elements: all elements appear in sorted order.

- **Disjoint set.** The Disjoint set data structure is a linked-based implementation of the fast union-find data structure [26]; this implementation uses both path compression and rank estimation heuristics to improve efficiency. A Disjoint set object has a `header` and a `tail` node as well as a `size` field that represents the size of the set; each set node has a `child` and a `parent` field. Structural integrity constraints are acyclicity and reachability to the sentinel header node (the `parent` field of each node should point to the header node).

- **AVL tree.** An AVL tree is a balanced binary search tree. The integrity constraints are the same as those of the binary search tree as well as the balance property where the height of the left and the right sub-trees does not differ by more than one.

- **Ranked tree.** A ranked tree is an augmented AVL tree where each node in the tree is labeled with a `rank` value defined as follows: given a node n and a function $size$ that returns the number of nodes in a sub-tree, the rank of n is equal to $size(n.left) + size(n.right) + 1$. Integrity constraints are the same as those of an AVL tree in addition to the consistency of the `rank` field.

6.1.2 Methodology

To evaluate the efficiency of the repair algorithm, we apply it to repair the subject structures described in the previous section. We study the repair time and the number of repair actions with respect to the size of the structure.

We generate a set of data structures for each subject and use fault injection to introduce inconsistencies in the structural and data constraints. Recall that several parameters can affect the performance of repair (Section 3.6.1), e.g., the location of the fault with respect to the root of the structure. To incorporate these parameters into the evaluation, we design several techniques for fault injection and apply the appropriate technique on the subject structures.

The first technique is *random fault injection* where we select a field of the structure at random and assign it to a node of a compatible type that is also selected at random. This technique may break the reachability of the structure since assigning fields randomly may cause the loss of some nodes due to garbage collection. We therefore apply this technique to inject faults in the data elements of the data structures where we randomly select a structure node and assign its data fields to randomly selected integer values. We do not expect such a scenario to happen in practice, but it helps assessing the performance of the repair algorithm on unexpected behavior.

The second technique is *location-based fault injection* where we select equidistant nodes on a path from the root object along a specified set of fields and randomly assign the fields to nodes in the structure that are selected at random. We use this technique to perform an experiment where the faults are not clustered in a single region of the data structure but rather spread over the whole structure. We have used this technique to study the effect of the location of a fault on repairing doubly linked lists (Section 3.6.1) .

The last technique is *ad hoc fault injection* where we manually identify the relationship between the fields of the structure and distribute the faults accordingly. This is not a generic approach and it differs according to the subject structure. This technique is effective for understanding the performance results of the repair algorithm. We already used this technique in section 3.6.1 to study the dominance of repairing a fault in the `next` field of a doubly linked list on the overall repair time. We use this technique in situations where random selection might break structure reachability. For example, to introduce cycles in a tree structure, we only consider leaf nodes.

We conduct experiments on structures with sizes ranging from ten thousand nodes to one million nodes, and with 100 faults injected using the three described fault injection techniques. All experiments use a 1.7 GHz Pentium D with 2 GB of RAM.

6.1.3 Results

Table 6.1 shows the repair time and number of repair actions performed by the repair algorithm for repairing the subject structures. The table also shows the fault injection technique used for each subject.

For the sorted list, we use random fault injection where a hundred data fields are assigned to random integer values. Since the only structural constraint is acyclicity, we do not inject any faults in the `next` field of the list since changing any field breaks the reachability of other elements in the list. For repairing a list with up to a million nodes, the repair algorithm took less than 20 seconds and performed around seven hundred thousand repair actions. The actions performed by repair include shifting the elements of the list to retain the sorted order.

For the binary tree example, we use location-based fault injection. We generate a complete binary tree where each node has two children except nodes at the last level of the tree which can have either one or no children. We choose a hun-

Subject structure	Size	Number of faults	Repair time (ms)	Repair actions	Fault injection
Sorted list	10,000	100	122	5,188	random
	100,000	100	1,552	60,757	
	1,000,000	100	18,542	680,153	
Binary tree	10,000	100	140	4,257	location-based
	100,000	100	1,500	42,752	
	1,000,000	100	19,094	406,837	
Disjoint sets	10,000	100	212	24,520	ad hoc
	100,000	100	2,947	256,526	
	1,000,000	100	35,857	3,715,811	
Avl tree	10,000	100	265	67,106	ad hoc
	100,000	100	2,406	324,674	
	1,000,000	100	28,266	2,422,175	
Ranked tree	10,000	100	294	85,434	ad hoc
	100,000	100	3,219	454,633	
	1,000,000	100	37,548	3,233,502	

Table 6.1: Results for applying the repair algorithm on five subject structures.

dred nodes at the last level of the tree and introduce cycles as well as corruptions in the `parent` field of the nodes. For the binary tree example, the repair algorithm performed four hundred thousand repair actions on the fields to generate a correct binary tree with up to a million nodes in nineteen seconds. The repair actions include breaking cycles in the tree and searching for the nodes that correct the `parent` field.

For the disjoint sets, we use ad hoc fault injection. We first generate a valid structure and then select a hundred equidistant nodes along the path following the `child` field. For each of these nodes, we set the `parent` field to a randomly selected node. We then split the set into two structures; one with nodes reachable through the `tail` field and the other with nodes reachable through the `header` field. To successfully recombine the two sets and satisfy the constraints, the repair algorithm performed around three million mutations in less than forty seconds. The repair was performed in three stages. The first traversal of the structure connected the two structures along the `child` field. Further traversals took care of fixing the `parent` field and assigning the `tail` of the set.

For the last two structures, we combine two fault injection techniques. We use random fault injection to corrupt the `rank` field of the nodes of a `ranked` tree and data fields of the `avl` tree, and ad hoc fault injection to insert cycles in the trees where we only introduce cycles in the fields whose original value is `null`. Similar to the other structures, the repair algorithm repaired trees with a million nodes in less than a minute.

These results show the applicability of the approach to various complex structures. For all the studied structures, the repair algorithm was able to perform repair in less than a minute for data structures with up to one million objects.

6.2 Experiments on Stand-alone Applications

To study the acceptability of assertion-based repair, we use it to repair inconsistencies in the data structures of stand-alone applications and we observe the effect of repair on the running applications. We choose a diverse set of open-source applications including a software analyzer [12], a calendar application [8], and a database engine [112].

Challenges while conducting the experiments include studying the applications' source code and documentation, locating the key data structures, identifying the consistency constraints, formulating `repOk` methods to describe the constraints in terms of the application constructs, and modifying existing assertions (or adding new assertions) to enable repair.

To generate inconsistencies at runtime, we mine the bug repositories of these applications to detect known bugs that either resulted in data structure corruption errors in the applications themselves or indirectly resulted in errors in programs that use the buggy applications. For the experiments in this section, we do not use fault injection.

For each application, we develop an input scenario that results in a corrupt data structure. We run the application on the input and compare its behavior with and without the repair capabilities.

6.2.1 ASM

ASM [12,76] is an open-source framework for manipulating Java bytecode. ASM provides an efficient engine for both code instrumentation and code generation making it a powerful utility for software analysis. Several open-source projects including, language interpreters [99] and software development and testing frameworks [40] use ASM to perform code transformation. Juzi also uses ASM to perform bytecode instrumentation to incorporate repair in Java programs.

6.2.1.1 Bug Report

ASM supports frame manipulation for programs complying with the Java specifications. A program compiled using Java 6 (or a higher version) includes stack frame information in addition to bytecode instructions to enable efficient class verification [80]. At every label in the program representing a branch target or an exception handler, a stack frame is saved holding the types of the objects present in the local variables and in the operand stack. While analyzing a method's bytecode, ASM provides any registered client *visitor* with the frame information and the object types. According to the ASM documentation, the object type information provided by ASM is interpreted as follows: primitive types are represented by an integer encoding the type, reference types are represented by a `String` object representing the type's internal name, and uninitialized types (that correspond to objects that are created using `NEW` but whose constructor has not yet been invoked) are represented by a `Label` object that points to the location of the `NEW` instruction that created the type.

A bug in version 3.1 causes ASM to announce inconsistent type information when analyzing frames with uninitialized types. The frames announced by ASM mistakenly point to labels that do not exist in the program. This bug surfaces when analyzing a corner test case that involves the tertiary operator $(expr?op1 : op2)$ as a parameter of a constructor call. To illustrate, Figure 6.1 shows a Java class that triggers the bug, along with a snapshot of the reported bytecode. Note in the bytecode, the `FRAME` statements include a pointer to label `L2` that does not exist in the method `foo`. This bug is reported in the ASM bug repository with `id#312464`.

```
// The Java class that triggers the ASM bug
public class FrameBug {

    public boolean foo(int x, int y) {
        Integer m = new Integer(x == 0 ? 0 : 1);
        Integer k = new Integer(y == 0 ? 0 : 1);
        return m > k;
    }
}

// The corresponding bytecode reported by ASM
public foo(II)Z
   NEW java/lang/Integer
   DUP
   ILOAD 1
   IFNE L0
   ICONST_0
   GOTO L1
  L0
  FRAME FULL [FrameBug I I] [L2 L2]
   ICONST_1
  L1
  FRAME FULL [FrameBug I I] [L2 L2 I]
   INVOKESPECIAL java/lang/Integer.<init> (I)V
   ASTORE 3
  L3
   NEW java/lang/Integer
   DUP
   ILOAD 2
   ...
```

Figure 6.1: A bug in the ASM framework.

6.2.1.2 Bug Effect

In Juzi, we use ASM to manipulate the bytecode of Java classes to enable repair. For each method, we build the control flow graph (CFG) where the first instruction in a code block corresponds to the first instruction of the method, branch targets, exception handlers, or instructions that follow branch instructions. A CFG block that corresponds to a branch target holds the frame information, where a frame containing a `Label` object points to the CFG block that contains the target label.

Figure 6.2 shows the declaration of the data structure used to represent the basic blocks in a CFG. Each CFG instance has an `entry` block and a list of `exit` blocks. Each basic block has a list of predecessor and successor basic blocks as well as a reference to the first and last instruction in the block. A basic block also has a reference to a frame instance. Each instruction node is a doubly linked list node that holds two references to the next and previous instructions in sequential order. A frame instance has a set of types for the local variables and the variables in the operand stack. Each uninitialized type is associated with a pointer to the basic block that holds the `Label` corresponding to the `NEW` instruction responsible for creating the object of that type.

We used Juzi to analyze the program that triggers the bug in Figure 6.1. The bug in ASM causes Juzi to build an inconsistent CFG where a frame instance has a `Label` object for `L2` that corresponds to a `null` basic block. After building the CFG, Juzi throws a `NullPointerException` when trying to access the information of the basic block.

6.2.1.3 Repair Result

We modified the code in Juzi by formulating a `repOk` method that describes the consistency constraints of the CFG class and added assertions to check for the consistency after building the CFG. The integrity constraints that we considered for

```
public class CFG {
    BasicBlock entry; // The entry block
    BasicBlock[] exitBlocks; // The exit blocks

    public class BasicBlock {
        BasicBlock[] pred; // The predecessor blocks
        BasicBlock[] succ; // The successor blocks
        AbstractInsnNode first; // The first instruction
        AbstractInsnNode last; // The last instruction
        Frame frame; // The frame information
        int type; // The block type
    }

    public class Frame {
        int type; // The frame type
        Object[] localTypes; // Local variable types
        Object[] stackTypes; // Operand stack types
        // Blocks holding uninitialized types
        BasicBlock[] unInitializedTypes;
    }

    public class AbstractInsnNode {
        int type; // The instruction type
        AbstractInsnNode next; // Next instruction
        AbstractInsnNode prev; // Previous instruction
    }
}
```

Figure 6.2: The declaration of the CFG class in Juzi.

the CFG class are the following: (1) the entry block has no predecessors, (2) exit blocks have no successors, (3) if a block $B1$ is a predecessor of block $B2$ then $B2$ is a successor of $B1$, (4) instruction nodes must satisfy the constraints of a doubly linked list, and (5) uninitialized types in a frame must point to a `Label` object corresponding to a `NEW` instruction.

We instrumented the CFG class to enable repair and ran the version of Juzi with the repairable CFG on the program that triggered the error. The new version of Juzi detected the inconsistency in the CFG and repaired the CFG as follows. The repair algorithm traversed all the instruction nodes searching for a pattern of two instructions where the first is a `Label`, and the second is `NEW`. The algorithm found the instructions corresponding to label `L3` and assigned it to the frame. The algorithm then found the basic block that contained the two instructions and assigned it to the `null` pointer that caused the error in the first place. The generated CFG enabled the execution of Juzi to proceed and generated a log-report with the changes that occurred. The repair process was performed in less than 100 milliseconds and 15 repair actions were considered.

6.2.2 Borg Calendar

Borg [8] is an open-source calendar and task tracking system written in Java with over a hundred thousand downloads on SourceForge. Borg provides several features including a to-do list, several calendar views, popup reminders, repeating appointments, as well as importing/exporting data from/to various calendar formats. To maintain the user data, Borg interacts with a database management system (DBMS) and maintains a database holding the user projects, tasks, appointments, and memos.

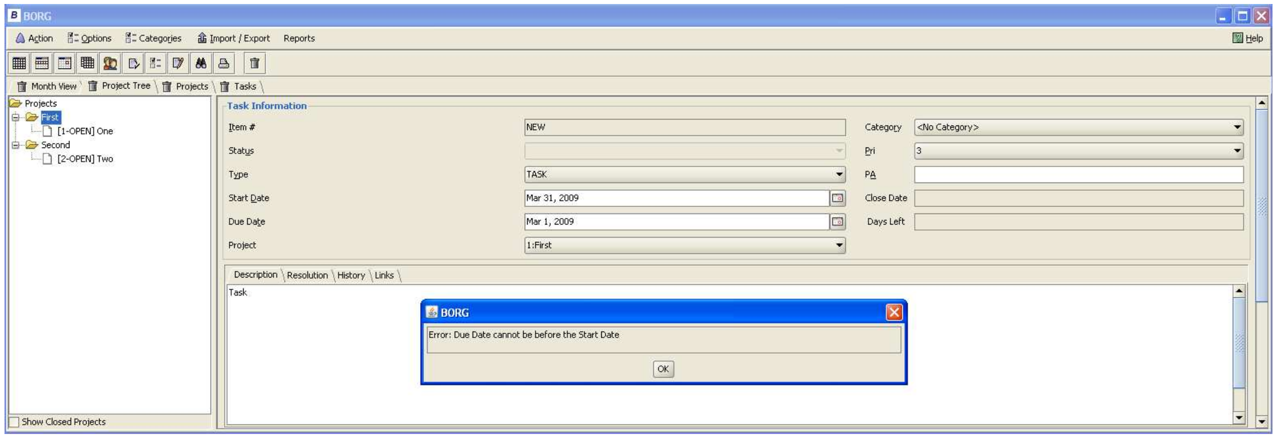
6.2.2.1 Case Study

Borg provides two methods for creating calendar projects. The first method is interactive, where the user creates projects and tasks by filling up forms that popup upon request. The second method is batch, where the user provides an XML file describing the projects and the tasks and imports the file into Borg. After creating each component, Borg saves/updates the database with the calendar information.

Borg's implementation supports a fair amount of error checking when manipulating the calendar components interactively. For example, when creating a project in the calendar, Borg asserts that the due date of a project is assigned after the start date, and that a description string is provided for a project. If these properties are violated, Borg reports the error in a message box and does not allow the project creation. Figure 6.3(a) shows a snapshot of Borg's error message.

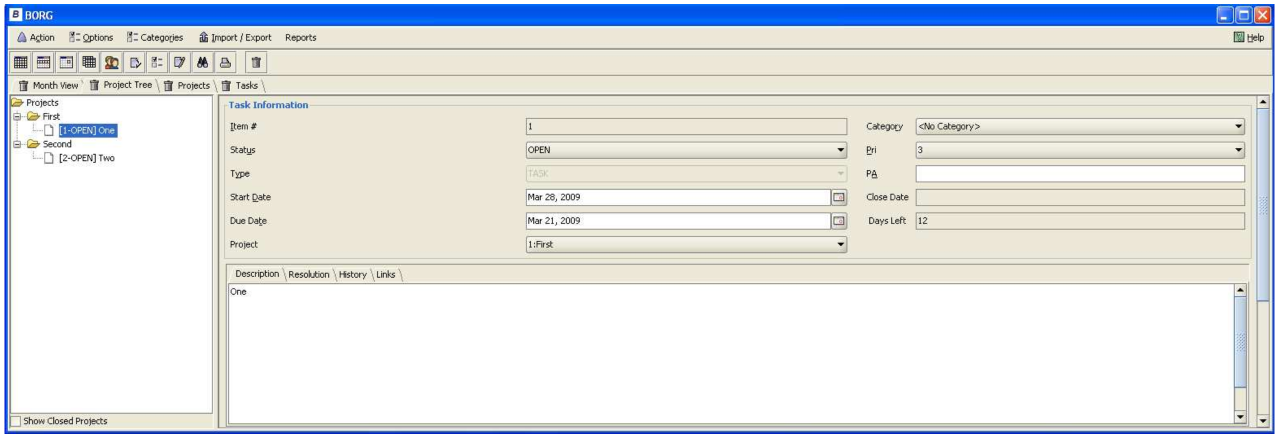
These consistency checks, however, are not performed when the calendar data is imported from an XML file. An inconsistent XML description file can result in an inconsistent calendar. For instance, consider the following XML data that represents a project with one task:

```
<TASKS>
 ...
<Project>
    <KEY>1</KEY>
    <Id>1</Id>
    <StartDate>03/01/09 12:00 AM</StartDate>
    <DueDate>03/31/09 12:00 AM</DueDate>
    <Description>First</Description>
    <Status>OPEN</Status>
</Project>
<Task>
    <KEY>1</KEY>
    <TaskNumber>1</TaskNumber>
    <StartDate>03/28/09 12:00 AM</StartDate>
    <DueDate>03/21/09 12:00 AM</DueDate>
    <State>OPEN</State>
    <Type>TASK</Type>
    <Description>One</Description>
    <Project>1</Project>
</Task>
 ...
</TASKS>
```


Item #: NEW
Status:
Type: TASK
Start Date: Mar 31, 2009
Due Date: Mar 1, 2009
Project: 1:First
Category: <No Category>
Pri: 3
Description: Task

Error: Due Date cannot be before the Start Date

(a)


Item #: 1
Status: OPEN
Type: TASK
Start Date: Mar 28, 2009
Due Date: Mar 21, 2009
Project: 1:First
Category: <No Category>
Pri: 3
Days Left: 12
Description: One

(b)

Figure 6.3: (a) Borg error message indicating an inconsistency. (b) An inconsistent task imported from XML.

Figure 6.3(b) shows a snapshot of Borg's state after importing the file into the calendar. Notice the corruption in the task information; the start date is assigned after the due date without any complaints from Borg.

6.2.2.2 Repair Result

We consider two ways to prevent the faulty scenario. The first is by asserting the consistency of the loaded data before updating the database as performed when creating tasks interactively, and the second is by performing repair.

To import the data from an XML file, Borg implements a parser that parses the XML into a simplified DOM tree. The data is then analyzed and translated to SQL queries into the underlying database. The declaration of the tree structure is shown below:

```
public class XTree {
    // Each XML element has a name and a value
    String name;
    String value;

    // Each XML element has children, siblings, and a parent
    XTree firstChild;
    XTree lastChild;
    XTree sibling;
    XTree parent;
}
```

Each `XTree` object represents an XML element. Each element has a `name` and an optional `value`. An `XTree` object also has pointers to the `parent` XML element, the `first` and the `last` children elements, and the `sibling` element which resides at the same tree level. Figure 6.4 illustrates the data structure shape.

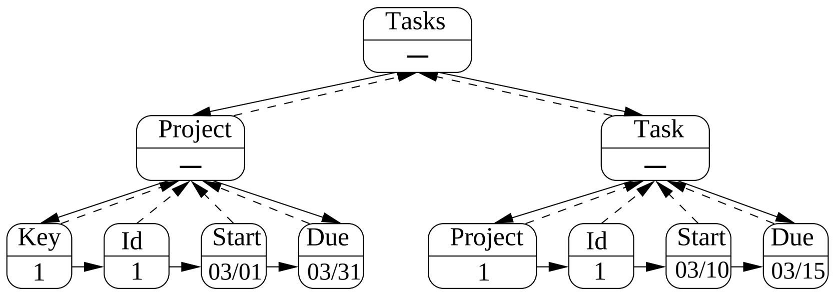


Figure 6.4: The `XTree` data structure in Borg.

We check for two types of consistency constraints: constraints on the structure of the `XTree` nodes, and constraints on the contents of the `XTree` nodes. For the structural constraints, we consider the standard tree constraints which include acyclicity along the children pointers, correctness of the parent pointer, and the validity of the list along the siblings. For the data constraints, we consider the validity of the relationships between the data elements. For example, the start date element of a task must happen before the due date element of the same task. For another example, the duration of a task must not exceed the duration of its project.

We formulate a `repOk` method that describes the above properties and modify Borg's source code to check for the consistency of the constraints after parsing the XML file. We report an error message similar to the one in Figure 6.3(a) when an inconsistency is detected.

For the first experiment, we disable repair. We import a faulty XML file with two faults: (1) a task with a duration that expands beyond its project's duration, and (2) a task with a due date that happens before its start date. Borg fails to import the XML file due to the inconsistency in the calendar information. Instead of loading the calendar, Borg announces an error message indicating the load failure and proceeds with its execution. Note that this problem persists if we try to load the same file again.

For the second experiment, we instrument the `XTree` class and use the instrumented class in Borg. We run Borg and import the faulty XML file. Borg now shows a warning message indicating that repair was performed and points to the log-file which shows the repair steps. The repair algorithm detects the inconsistencies in the data elements. For the first fault, it sets the start date to the project's start date and the due date to the project's due date. For the second fault, it sets the start and due dates to the same value. The repair process took less than 50 milliseconds. Repair allows loading the calendar. While it is highly likely that the repaired tasks do not have the original intended dates, the repair logs point out the errors while allowing Borg to process the file.

6.2.3 HSQLDB

The HSQL database engine [112] is a popular relational database for Java with over one million downloads on SourceForge. HSQL supports a rich subset of the SQL language and provides both in-memory as well as disk based queries. HSQL is being used in several open-source projects. We used HSQL while experimenting with Borg to maintain the calendar state. We also used HSQL in our research on DBMS testing [68].

6.2.3.1 Specifications

Database tables in HSQL are represented as set of rows holding the data elements. To enable efficient operations on the tables, HSQL supports various indexing schemes for the data. Figure 6.5 shows the declaration of the data structures used by HSQL to maintain the tables of the database[1]. For efficient in-memory indexing, HSQL maintains the indices of the database records in an internal data structure that implements a balanced binary search tree. The class `Index` implements AVL trees and the class `Node` represents the nodes corresponding to the rows of the table. Each node has a `left`, a `right`, and a `parent` field defining the backbone of the tree and an integer field defining the tree `balance` at a node. Additionally, a node has an instance of the `row` object it represents, and a `next` field representing a node in another index tree that corresponds to the same row. We will shortly elaborate on the use of the `next` field.

Multiple indices can be created for a database table. A primary index is created for a table to identify unique rows. This index is created by default in case the user does not specify it. Other indices include foreign key indices that relate a table to other tables and user created indices (using the CREATE INDEX SQL expression) that enable faster processing of desired columns. Figure 6.6 shows the shape of a `Table` data structure with n indices. Each index is represented as a separate AVL tree. A row that belongs to the table is represented as n nodes in the data structure where each node belongs to a different index tree and all the nodes

[1]We modified the original field names for the clarity of the description.

```
public class Table {
    Index[] indexList;
    int indexCount;
    ...
}

public class Index {
    Node root;
    ...
}

public class Node {
    Node left;
    Node right;
    Node parent;
    Node next;
    int balance;
    Row row;
    ...
}

public class Row {
    Object data[];
    Node primary;
}
```

Figure 6.5: Data structures to maintain the data in an HSQL database.

point to the same row (we omit the arrows from the nodes to the rows in Figure 6.6 for clarity). These nodes are connected in a linked list where the `primary` node represents the header node of the list in the primary index (I_1). The list elements can then be accessed following the `next` field of the nodes. An update on a row may require changes on the nodes pointing to that row. The linked list structure enables quick access to all such nodes.

The structural integrity constraints for the `Table` data structure are the following: (1) acyclicity along `left` and `right`, (2) correctness of `parent`, (3) the

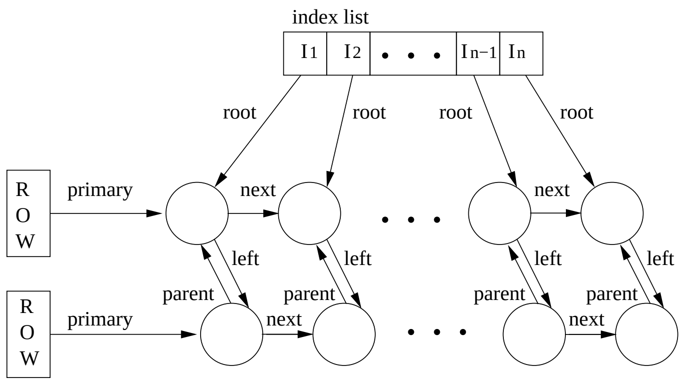


Figure 6.6: The data structure to maintain table indices in HSQL.

height-balance property where the number of nodes in the `left` and `right` subtree do not differ by more than one, (4) the linked list constraints along the `next` field where the primary index points to the head of the list and the last index points to the last node in the list, and (5) the order constraint where the order in which the nodes are inserted in the linked list must match the order in which the corresponding indices are inserted in the `indexList`.

6.2.3.2 Bug Report

Earlier versions of HSQL contained a bug that corrupted the consistency constraints of the data structures and caused loss of data when updating the database. This is a known bug `id#878288` that is reported in HSQL's bug repository and is fixed in later versions.

When creating table indices, HSQL inserts every new index at the end of the `indexList` in the `Table` class. For each row in the table, it creates a node object for the new index, inserts the node in the index's AVL tree, and adds the node at the end of the linked list along the `next` field. An optimization was added to the `Table` class that clusters the indices in the `indexList` according to their types (primary, foreign, or user created) and not in the order they are created. The optimization, however, was not reflected on the order in which the elements are inserted in the node's linked list, which corrupted the order constraint of the data structure and caused multiple index trees to cross-link.

6.2.3.3 Repair Results

We formulated a `repOk` method to describe the structural integrity constraints of the `Table` data structure in HSQL. We asserted the properties at the exit statements of the `createIndex` method in HSQL that is responsible for creating the table indices. We used the following SQL script to corrupt the data structure:

```
// Create table ``users'' with a primary index
CREATE TABLE users (id INTEGER primary key, name TEXT, age INTEGER)
INSERT INTO users VALUES(1, 'n1', 10)

// Create table ``foreign'' with a primary index
CREATE TABLE foreign (id INTEGER primary key)
INSERT INTO foreign VALUES(1)

// Add an index on the age field as a foreign key
ALTER TABLE users ADD FOREIGN KEY (age) REFERENCES foreign(id)

// Add an index on name column
CREATE INDEX name ON users (name)
```

The SQL statements perform the following: (1) create two tables `users` and `foreign` and insert one row of data in each table, (2) create a foreign key constraint which inserts an index on the `age` column in the `users` table, and (3)

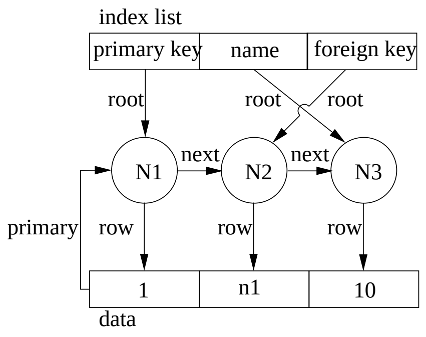


Figure 6.7: A corrupt HSQL index structure.

create a user index on the `name` column of the `users` table. Since HSQL puts user created indices before foreign key indices, the created data structure is corrupted. Figure 6.7 shows the state of the data structure after creating the last index. Note the corruption in the order constraint where the order in which the indices are placed in the `indexList` does not match order in which the corresponding nodes are inserted in the linked list.

We executed the script in the presence of the assertion (but without repair). HSQL detected the corruption and threw an exception while creating the last index. The exception allowed HSQL to roll back and stop the execution of the script.

We then executed the script with the repair enabled. When executing `repOk` on the corrupt structure, the repair algorithm is triggered and the structure is repaired. The repair algorithm detects the inconsistency in the linked list and reorders the nodes of the list to satisfy the order in which the indices are inserted. The repaired structure is similar of the structure that is generated by HSQL after applying the bug fix. For the example above, the repair was performed in 40 milliseconds.

6.3 Discussion

We next discuss our experience with using assertion-based repair while designing and analyzing the experiments. The experimental results demonstrated the ability of the approach to repair complex data structures while enabling programs to proceed with their execution. The repair results, however, varied among different structures. Three key factors affected the results of repair: (1) the nature of the faults in the data structures, (2) the implementation of the `repOk` methods, and (3) the program tolerance to internal state changes. We discuss the effect of each of these factors and draw general conclusions about the repair results.

6.3.1 Faults in Data Structures

We use an abstraction of data structures to interpret the repair results. We view three different roles for the fields of a data structure. Some fields define the *backbone* of the data structure. These fields typically connect the structure nodes and define the structure reachability. Examples of such fields are the `next` field of a linked list, the `left` and `right` fields of a `BinaryTree`, and the `child` field of a disjoint set. Other fields in the structure enable efficient operations. These fields add *redundancy* to the data structure, yet, they are necessary for practical usage. For example, in a doubly linked list, the `next` field enables traversing the structure and reaching all the nodes. However, efficient delete operations require quick access to the node before the node to be deleted. The `prev` field provides such a quick access. The rest of the fields define the *data* stored in the data structure, e.g., the `element` field of a binary search tree.

The experiments showed the following behavior of the repair algorithm when repairing fields with similar roles.

- Repairing faults in the fields that define the structure's backbone resulted in reordering the objects of the data structure.
- Repairing faults in the redundant fields resulted in the hypothetically correct data structure, i.e., the structure before fault injection.
- Repairing faults in the data fields resulted in either duplicating a value that existed in the structure, swapping the values between fields, or introducing new values that did not originally exist in the structure.

This generalization of behavior was observed in the majority of the studied structures. For example, repairing faults in the `next` field of a doubly linked list or the database structure resulted in reordering the nodes in the list. Repairing the `parent` pointer in a disjoint set, or the `rank` field of the ranked tree resulted in the original structure since these fields define redundant information. Repairing data fields resulted in duplicating existing fields as in the calendar example or introducing a new value as in the binary search tree.

6.3.2 Implementation of `repOk`

Our experience with implementing `repOk` methods showed that implementing `repOk` methods to check for the global constraints first, i.e., constraints on the backbone of the structure, followed by the local constraints, i.e., constraints on the redundant fields, and then the data constraints resulted in a more efficient repair than any other order. This follows from the intuition of incremental constraint solving and separation of concerns. By generating a valid backbone of a data structure, then fixing all other fields as an incremental addition to the backbone, and finally repairing the data fields, we separate solving for reachability from solving the rest of the constraints. Moreover, by representing independent constraints separately in `repOk`, we decouple the structural constraint solver in the repair algorithm from the symbolic execution engine. This, in general, reduces the number of wrong decisions made during the search and in turn improves performance.

6.3.3 Program Tolerance to Changes

To reason about the acceptability of the generated structures, we consider an alternative view of the repair problem.

Consider the heap state of a program right before executing a repair assertion. The heap can be viewed as a graph where the vertices are the objects created during execution and the edges are the fields of those objects. Let S be the set of all the valid graphs that are the result of the program execution, and let S' be the set of all the valid graphs that can be formed using the object fields. The set S is a subset of the set S'.

Now consider the result of repairing an invalid graph using the same assertion. The result of repair is a valid graph that either belongs to S, i.e., a graph that the program can build; or belongs to S' but not S, i.e., a graph that the program cannot build. In the former case, we expect the program to tolerate the divergences from the original graph. For example, this scenario occurred in the calendar example, where the repaired structure was one that sets a task duration to its project duration; this can be generated using a valid user input. In the latter case, we cannot draw conclusions regarding the behavior of the program after repair.

Chapter 7

Repair-based Test Case Generation

Software testing, the most commonly used technique for validating the quality of software, is a labor intensive process, and typically accounts for about half the total cost of software development and maintenance [7]. Automating testing would not only reduce the cost of producing software but also increase the reliability of modern software.

For programs that take as inputs structurally complex data, which pervade modern software, test generation is particularly hard. Desired inputs must satisfy complex structural integrity constraints that characterize valid structures.

There are two fundamental approaches for generating structurally complex tests: one, representation-level generation by explicitly allocating objects and setting values of their fields such that the underlying constraints are satisfied; two, abstract-level generation by a sequence of method invocations using the API. The two approaches are complementary and have their advantages and disadvantages. For example, while concrete-level generation requires the user to a priori provide constraints, abstract-level generation requires the user to first correctly implement the methods used in a sequence.

Recent years have seen a significant progress in automating both these approaches. Constraint-based techniques are able to provide efficient test enumeration at the representation level using off-the-shelf SAT solvers [87] as well as using novel search algorithms [10, 72, 106]. Efficient state matching algorithms are able to provide test enumeration at the abstract level by pruning redundant method sequences [115, 120, 121].

Much of the prior work, however, has focused on systematic generation of small structures. The motivation—inspired by traditional model checking—for that is to enable bounded exhaustive testing, where a program is tested on all (in-equivalent) inputs within a small input size. While bounded exhaustive testing does increase a developer's confidence in their software, it is not prudent to altogether ignore testing the program on larger inputs.

This chapter presents a novel algorithm for constraint-based generation of large inputs that represent structurally complex data. We view structures as object graphs whose nodes represent objects and edges represent fields. A key observation behind the generation algorithm is that while generating an object-graph that satisfies desired structural constraints is hard, generation of a connected graph at random with a desired number of nodes is straightforward. Of course, a graph generated at random is highly unlikely to satisfy any of the desired constraints and would therefore represent an invalid structure. However, we can systematically *repair* it, using STARC, such that it satisfies all the constraints.

Experimental results using a prototype implementation show that the generation algorithm can generate structures that are 100 times larger than those possible with previous constraint-based generation techniques, such as Korat [10] that implements a dedicated search, or TestEra [87] that uses the Alloy Analyzer and off-the-shelf SAT solvers, such as mChaff [92].

7.1 Repair Based Generation

This section describes the test generation algorithm. The prototype implementation utilizes three main engines: *Egor*, a random graph generator, *STARC*, the data structure repair framework described earlier, and *Dicos*, a difference constraint solver.

We describe the algorithm for generating a structure that has a unique root; structures that have more than one root are handled similarly [10]. Figure 7.1 shows the generation framework, which takes three inputs: (1) `clazz` that represents the class of the structure's root, (2) predicate `repOk` representing the structural integrity constraints, and (3) `size`, a set of pairs, which defines the number of objects for each class in the structure. To illustrate, consider the declaration of the class `BinarySearchTree` from Section 3.1.2. To generate tree objects with 100 nodes, we set `size =` {`<BinarySearch Tree, 1>, <Node, 100>`}.

The generation algorithm performs the following steps:

- Allocate appropriate objects using the field declarations in `clazz` and generate a random graph using these objects; indeed, this graph may not satisfy any of the desired constraints yet;
- Repair the reference fields of the random graph such that all constraints on

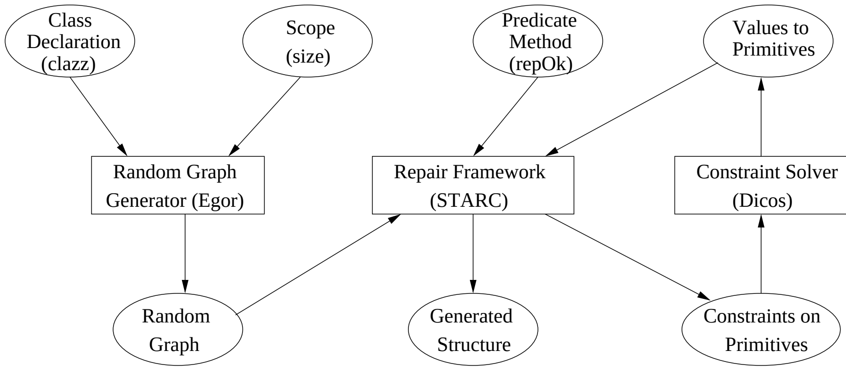


Figure 7.1: A framework for generating large data structures.

these fields are satisfied; STARC returns the constraints on the primitive variables;

- Solve the data constraints; Dicos returns a complete solution;
- Assign each data field its value; the resulting graph represents a concrete object-graph that satisfies all the desired invariants.

The rest of this section describes the details of the algorithm and its main modules.

7.1.1 Generating a Random Graph

Egor takes an object representing the class declaration of the structure's root class, and the desired size as inputs, and generates a random graph that is allocated on the heap. The vertices of the graph are new objects of the given classes. The edges of the graph represent the reference fields. Figure 7.2 shows the pseudo-code for the Egor random graph generation algorithm.

Intuitively, the algorithm starts with an empty graph. It then allocates new objects as required to generate a graph of the desired size. For each object, the algorithm randomly assigns values to the object's reference fields, ensuring at each step that the graph can further be extended if necessary. The algorithm terminates when the graph has the desired number of objects and their reference fields have been initialized.

To explain the algorithm, we first explain the notation we use in Figure 7.2:

- **`clazz`** is an object representing the container class of the structure (for example the `BinarySearchTree` class).
- **`size`** is a set of pairs representing the desired size of every class in the structure. Egor provides a helper method `desiredSize` that takes a field `f` and `size`, and returns the desired size of the class that is the declared type of `f`.
- **`liveObjectWorkList`** is a list of objects whose reference fields are yet to be assigned a value.
- **`assignedObjectSet`** is a set of objects whose reference fields have already been assigned a value. Egor provides a helper method `getRandomObject` that randomly returns an object from the `assignedObjectSet`.
- **`LiveFieldCount`** is a class that represents for each class the number of object fields, i.e., *live count*, that have not yet been assigned values in the structure. The live count of every class is initially set to zero. `LiveFieldCount` provides three helper methods: `get`, `update`, and `decrement`. The method

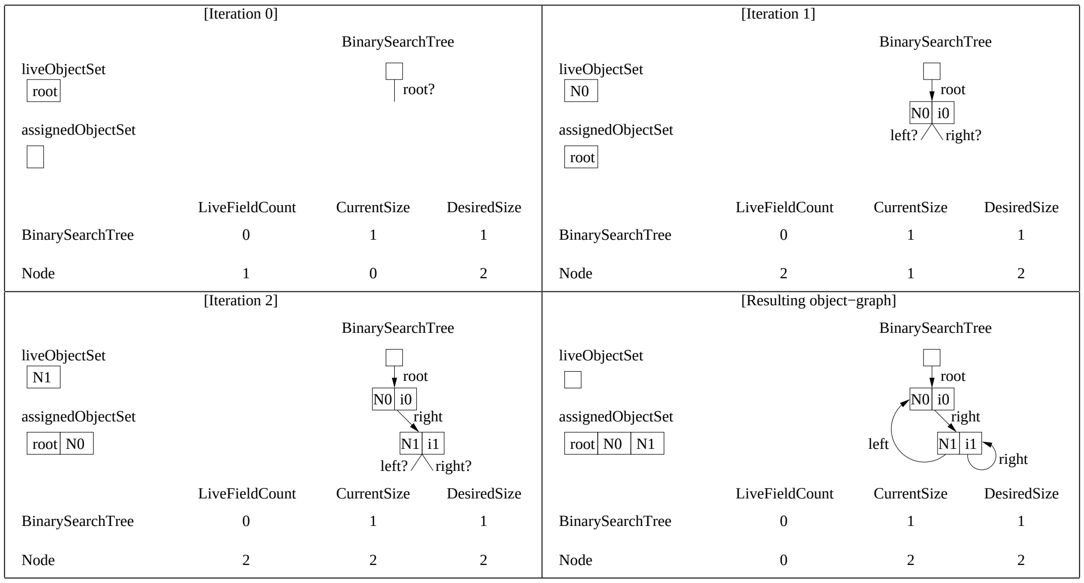
```
Object generateRandomGraph(Class clazz, Set<Pair<Class, int>> size) {
    // Initialize the structures for maintaining the algorithm state
    Random rand = new Random();
    LinkedList liveObjectWorkList = new LinkedList();
    Set assignedObjectSet = new HashSet();
    LiveFieldCount liveFieldCount = new LiveFieldCount(clazz);
    CurrentSize currentSize= new CurrentSize(clazz);

    // Create an instance of the root class, add it to the worklist, and
    // update the field info depending on the fields to be assigned
    Object root = clazz.newInstance();
    liveObjectWorkList.add(root);
    liveFieldCount.update(root);

    // Iterate until the work-list is empty, at each step pick up an
    // object and assign its reference fields
    while (!liveObjectWorkList.isEmpty()) {
        Object o = liveObjectWorkList.removeFirst();
        for (Field f : fields(o)) {
            // If the desired number of object is created, then assign
            // the field to null or a previously created object
            liveFieldCount.decrement(f);
            if (currentSize.get(f) == desiredSize(f,size)) {
                int i = rand.nextInt(2);
                if(i == 0) f.setValue(null);
                if(i == 1) f.setValue(getRandomObject(assignedObjectSet));
            } else {
                // If f is the last field to be assigned of a type t, assign f
                // to a new object of type t to enable extending the graph
                if (liveFieldCount.get(f) == 0) {
                    Object o' = newInstance(f);
                    f.setValue(o');
                    liveObjectWorkList.add(o');
                    liveFieldCount.update(o')
                    currentSize.update(f);
                } else {
                    // Randomly assign f to (1) null, (2) a previously created
                    // object, and (3) a new object of the field's type
                    int i = rand.nextInt(3);
                    if(i == 0) f.setValue(null);
                    if(i == 1) f.setValue(getRandomObject(assignedObjectSet));
                    if(i == 2) {
                        Object o' = newInstance(f);
                        f.setValue(o');
                        liveObjectWorkList.add(o');
                        liveFieldCount.update(o');
                        currentSize.update(f);
                    }
                }
            }
        }
        assignedObjectSet.add(o);
    } }
```

Figure 7.2: The Egor algorithm for generating random graphs.

`get` takes a field object and returns the live count of the field's declared class; `update` takes an object, and for each of its fields, increments the live count of the field's declared class; `decrement` takes a field object, and decrements the live count value of the field's declared class.

- **`CurrentSize`** is a class that represents the number of objects for each class in the structure. For each class, the current size is initially zero. The class `CurrentSize` provides two helper methods: `get` and `update`. The method `get` takes a field and returns the current size of the field's declared class; `update` takes a field and increments the size for the field's declared class.

The Egor generation algorithm first initializes its variables. Next, it creates an instance of the root class (`clazz`), adds it to the `liveObjectWorkList`, and updates the `liveFieldCount`. Next, Egor iterates until the `liveObjectWorkList` is empty. In each iteration, Egor removes the first object from the work list and assigns values to each of the object's reference fields as follows. When assigning a field f of type t, Egor first checks the `currentSize`, and the `desiredSize` for t. If `currentSize` is equal to the `desiredSize`, Egor randomly assigns f to `null`, or to an object from the `assignedObjectSet` since new objects of class t can no longer be added to the graph. If the current size is less than the desired size, Egor checks t's `liveFieldCount`. If it is zero, i.e., the graph can only be extended further by assigning a new object to f, Egor allocates a new object o' of type t, assigns o' to f, and updates the `liveFieldCount` and `currentSize` for t. If the live field count is greater than zero, Egor randomly assigns f to `null`, an object from



[Iteration 0]

	LiveFieldCount	CurrentSize	DesiredSize
BinarySearchTree	0	1	1
Node	1	0	2

[Iteration 1]

	LiveFieldCount	CurrentSize	DesiredSize
BinarySearchTree	0	1	1
Node	2	1	2

[Iteration 2]

	LiveFieldCount	CurrentSize	DesiredSize
BinarySearchTree	0	1	1
Node	2	2	2

[Resulting object−graph]

	LiveFieldCount	CurrentSize	DesiredSize
BinarySearchTree	0	1	1
Node	0	2	2

Figure 7.3: Generating a random `BinarySearchTree` object with two nodes.

the `assignedObjectSet`, or a new object of a compatible type. After assigning all the fields of an object, Egor adds the object to the `assignedObjectSet`.

As an example, consider generating a `BinarySearchTree` structure with two nodes. The algorithm takes three iterations of the while-loop. The algorithm state at the beginning of each iteration as well as the resulting object-graph are shown in Figure 7.3. The reference fields are labeled with the field name; a `'?'` indicates the field has not yet been assigned a value by the algorithm; fields that have the value `null` are omitted for clarity. Each node is labeled with its identity (`N0` or `N1`) and a symbolic integer value (`i0` or `i1`). Initially the `BinarySearchTree` root object is the only live object with one field (the `root`) to be assigned. Since assigning the `root` field to `null` prevents extending the graph to the desired size,

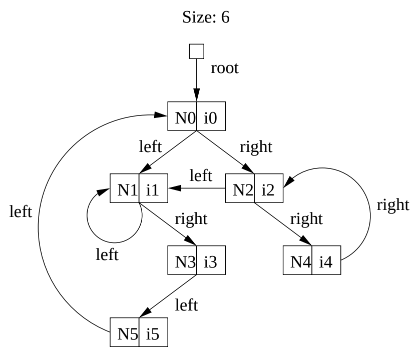


Figure 7.4: Random graph with six nodes generated by Egor.

the field is assigned to a new node `N0`. Node `N0` has two fields to be assigned `left` and `right`. Egor randomly sets the `left` field to `null` and the `right` field to a new node `N1` in order to satisfy the reachability constraint. Finally, Egor sets the `left` and `right` fields node `N1` to previously encountered nodes, and completes the graph. Note that in the last iteration Egor does not create any new nodes since the graph already contains the desired number of objects of type `Node`.

The generated graph satisfies two key properties: reachability, i.e., all the objects allocated are reachable from the root object, and randomness, i.e., the assignment to each field is made at random (using the Java API). Note that primitive data is left uninitialized. Determining the values for the primitive fields is performed using Dicos after the random structure is repaired by STARC. Figure 7.4 shows an example of a six node `BinarySearchTree` graph generated using Egor.

7.1.2 Completing the Structure

To complete the structure, STARC takes the random structure generated by Egor and `repOk`, and generates a repaired structure as well as the set of data

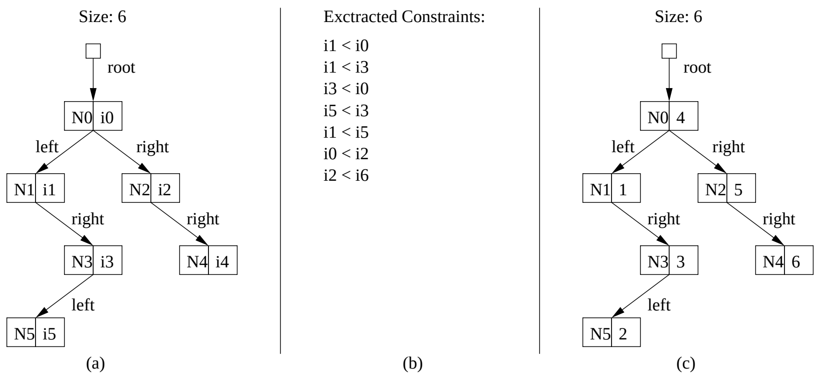


Figure 7.5: Completing the randomly generated graph.

constraints, which constrain the primitive fields of the resulting structure.

To illustrate, Figure 7.5(a) shows the repaired structure for the randomly generated structure in Figure 7.4. Figure 7.5(b) shows the data constraints extracted from the `BinarySearchTree` in Figure 7.5(a). The constraints returned by STARC are solved by Dicos (Section 7.1.3). The solution returned by Dicos is used to assign values to the data fields and complete the structure (Figure 7.5(c)).

7.1.3 Dicos: Difference Constraint Solver

Dicos is a difference constraint solver for primitive integers. Dicos handles integer constraints that take the form $x < y$ and $x \leq y$ as well as simple equality constraints of the form $x == y$. Following a textbook algorithm [26], the current implementation builds a constraint graph where the vertices are the primitive fields, and the edges are the constraints. Dicos adds a *root* node in the graph that is a predecessor of all the nodes. Once the graph is built, the problem simplifies to finding the single source shortest path from the added *root* node. To check the satisfiabil-

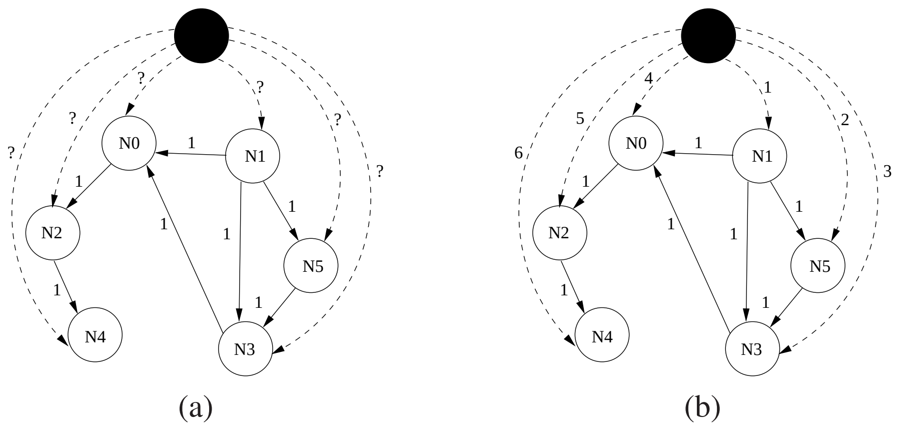


Figure 7.6: Solving difference constraints using Dicos.

ity of the constraints, Dicos checks for negative cycles in the graph. A negative cycle indicates a contradiction in the constraints. Dicos implements the Bellman-Ford [26] algorithm to find the shortest path in time $O(v.e)$. Since the complexity of the data constraints varies between structures, Dicos uses faster algorithms for handling simple constraints. For example, the data integrity constraints of the binary search tree example are translated into a directed acyclic graph (DAG) rather than a cyclic one. For a directed acyclic graph with v nodes and e edges, Dicos can compute the primitive values in $O(v + e)$ using a topological traversal.

To illustrate, Figure 7.6 shows the data constraint graph for the path condition in Figure 7.5(b). The dotted lines are the edges from the newly added root. Figure 7.6(a) shows the state of the graph before solving the constraints, all the dotted edges are labeled with `'?'`. Figure 7.6(b) shows the solution for the difference constraints; each `'?'` has been replaced with a value that satisfies the constraints. The topological distance from the added root node to each node determines the or-

	Repair-based			*Korat*	*TestEra*
Singly Linked List	Generation Time(ms)	Repair Time(ms)	Total Time(ms)	Total Time(ms)	Total Time(ms)
10 Nodes	≤ 1	≤ 1	≤ 2	37	3,000
100 Nodes	≤ 1	≤ 1	≤ 2	334	-
1,000 Nodes	2	5	7	-	-
10,000 Nodes	18	51	69	-	-
100,000 Nodes	199	583	782	-	-
Doubly Linked List	Generation Time(ms)	Repair Time(ms)	Total Time(ms)	Total Time(ms)	Total Time(ms)
10 Nodes	≤ 1	4	5	82	8,000
100 Nodes	≤ 1	44	45	3,204	-
1,000 Nodes	3	271	274	-	-
10,000 Nodes	34	3,718	3,752	-	-
100,000 Nodes	396	43,174	43,570	-	-
Binary Tree	Generation Time(ms)	Repair Time(ms)	Total Time(ms)	Total Time(ms)	Total Time(ms)
10 Nodes	≤ 1	5	6	21	5,000
100 Nodes	≤ 1	60	61	512	-
1,000 Nodes	3	372	375	-	-
10,000 Nodes	38	3,672	3,710	-	-
100,000 Nodes	402	45,267	45,669	-	-

Table 7.1: Results on solving constraints on the structure.

der of the data and solves the path condition. Dicos keeps track of the nature of the graph being constructed and then decides on which algorithm to use. Dicos even performs some simplifications on the path condition that might solve satisfiability without the need of a solver.

7.2 Evaluation

We evaluate the generation algorithm by applying it to generate six subject structures. For each subject, we evaluate the time it takes to generate one valid structure for sizes: 10, 100, 1,000, 10,000, and 100,000.

For solving purely structural constraints, two of the previous tools that have been shown to provide efficient solving are TestEra [87], which uses the Alloy Analyzer [61] and off-the-shelf SAT technology, and Korat [10], which implements an imperative constraint solver. We present a comparison of the repair-based generation approach with these two tools when generating structures with purely structural constraints. For data constraints TestEra and Korat are unable to compete with the repair-based approach because they require explicit enumeration of primitive values and checking of their constraints. The comparison with TestEra and Korat shows that the repair-based approach can generate structures of sizes that are 100 times larger. All experiments used a 1.7GHz Pentium D with 2GB of RAM.

7.2.1 Solving Constraints on Structure

Table 7.1 shows the results for the data structures with purely structural constraints. All tabulated times are in milliseconds. A time of `'-'` indicates failure to generate in 20 minutes. Singly linked list has the simplest of the constraints and its generation is therefore the fastest. For generating doubly linked lists and binary trees, repair-based generation can generate structures with up to 100,000 nodes in less than a minute.

We gave TestEra and Korat 20 minutes to generate one structure. Overall, Korat performs better than TestEra. However, Korat is unable to generate any subject structure with more than 800 nodes within the given time. The generation results show that the repair-based approach enables generating structures that are up to 100 times larger than those feasible with Korat and TestEra.

	Repair-based			
Sorted Linked List	Structure Generation Time(ms)	Structure Repair Time(ms)	Data Generation Time(ms)	Total Time(ms)
1,000 Nodes	3	11	27	41
10,000 Nodes	31	121	296	537
100,000 Nodes	338	1,423	2,571	4,332
Binary Search Tree	Structure Generation Time(ms)	Structure Repair Time(ms)	Data Generation Time(ms)	Total Time(ms)
1,000 Nodes	4	422	41	467
10,000 Nodes	42	4,008	389	4,439
100,000 Nodes	446	48,401	4,067	52,914
Avl Tree	Structure Generation Time(ms)	Structure Repair Time(ms)	Data Generation Time(ms)	Total Time(ms)
1000 Nodes	7	2,765	35	2,807
10000 Nodes	76	10,984	376	11,436
100000 Nodes	901	72,593	4,844	78,338

Table 7.2: Results on solving constraints on the structure as well as the data.

Notice that the random graph generation time is essentially proportional to the size and the number of fields in the target structure. The repair time dominates the total generation time as expected. Since the corrupt structure is generated at random, and only satisfies the reachability constraint, we expect the number of faults to be proportional to the size of the structure. Moreover we expect the faults to be distributed among all the fields of the structure. For repairing such random graphs, STARC took around 45 seconds to repair graphs with a hundred thousand nodes.

7.2.2 Solving Constraints on Structure as well as Data

Structures generated in this section have constraints on the order of the data. For a sorted list, the elements are ordered in a strictly increasing/decreasing order along the `next` field. For a binary search or avl trees the element in the root of a tree is larger than all the elements in the left sub-tree, and less than all the elements in the right sub-tree. We used TestEra and Korat to generate these structures, and both failed to generate the first structure with 400 nodes within 20min. TestEra and Korat use a search algorithm to solve the reference constraints as well as data constraints whereas we try to solve the two problems separately if possible which allows us to use a dedicated solver for data constraints.

Table 7.2 tabulates the results for three subject structures. Since the constraints are on the order of the data elements, the performance of Dicos scales essentially linearly with the size of the generated structures. For test generation, the performance of the random graph generator is linear with respect to the number of fields in the generated structure, and generates random graphs with a hundred thousand nodes in less than a second. The dominance of the repair time on the overall result is observable on the `BinarySearchTree` and the `AvlTree` examples which include complex constraints on the structure of the tree. For the sorted list, the repair time is primarily the time taken to build the constraints on the primitives. Note that for structures with mixed constraints, repair-based generation still scales and generates structures with a hundred thousand nodes in less than two minutes.

7.3 Characteristics

7.3.1 Generate Large Inputs from Specifications

Several techniques have been recently developed for performing scope-based testing, i.e., exhaustively testing a program on all inputs within a desired size.

Scope-based testing aims at verifying the correctness of programs when manipulating small size input, while projecting a similar behavior of the program for larger inputs. Software, however, may behave differently when subjected to inputs of different sizes. For example, a sorting process may use one algorithm for sorting a small number of elements, and a more efficient (and complex) algorithm for sorting a larger number of elements. Testing such a program on small inputs may not exercise the code of the efficient algorithm, and thus any anomaly in the implementation of the efficient algorithm may not be captured.

Testing software systems on large size input is crucial for detecting bugs and unexpected system behavior. It is typical in industry to have a team of testers dedicated towards stress and load testing [37]. Stress testing is the process of exposing the program to a heavy input load that does not occur in normal circumstances in an effort to detect any undesired behavior. This process is typically performed by (1) writing test scripts where the tester replicates small input scenarios to generate a large test, (2) running the program on the test cases, and (3) checking for abnormal behavior such as a program crash or memory problems. This approach has two key problems: (1) regression is very expensive; changes in the program may require large changes in the test scripts (or even rewriting the scripts from scratch) due to the replication, and (2) the tests that result from replicating or combining small input scenarios are usually skewed.

The approach presented in this chapter tackles these two problems. First, it generates large inputs using specifications. As long as the specification does not change, no changes are required in the tests despite modifications in the implementation. Second, the tests are generated automatically and at random providing a diverse suite for exercising possibly different scenarios. Moreover, with some tuning, the presented approach can provide a powerful tool for *negative testing*, i.e., generating inputs that violate a programs pre-condition and check how the program reacts to such inputs. The random graph generator is generic and generates a random heap which is more likely to be inconsistent. The generated heaps can be used to mimic the behavior of a program in the presence of a bug that puts the system in a similar state.

7.3.2 Test Case Enumeration

We have illustrated repair-based generation for generating one structure of a desired size. The same approach can also be used to systematically enumerate a given number of structures. We expect a typical usage of repair-based generation to be to generate a small set of large test inputs; for inputs of large size, exhaustive generation is infeasible in principle due to the enormous number of valid structures.

Chapter 8

Related Work

Assertion-based repair [35, 69] introduces a novel use of assertions for repairing data structure corruptions in deployed software. While primarily designed for error recovery, the techniques used in assertion-based repair are closely related to techniques used in specification based testing, dynamic test input generation, and software model checking.

This chapter reviews the work related to assertion-based repair. It first reviews related work on error detection, traditional error recovery, and constraint-based repair and then it relates repair to common assertion-based techniques for testing and verification.

8.1 Error Detection

Detecting errors using specifications at runtime is one of the earliest techniques for checking program correctness. There is a large body of research on behavioral specification languages [54, 61, 64, 78], both in their own regard and as annotations for code. In Larch [77], programmers write annotations in an abstract mathematical notation. The Java Modeling Language (JML) [78] is a general purpose specification language that combines Eiffel's [88] approach of basing the an-notation language on the underlying programming language with Larch's algebraic basis; several tools support JML [18, 94, 123]. The Unified Modeling Language (UML) [96, 102] is the de facto industry standard for object modeling. Another widely used specification language is Z [109], which allows constructing mathematical models of dynamic systems. An objected-oriented extension of Z has been developed to allow modeling systems as interactions between objects [108]. Alloy [60] is a relational, first-order logic suitable for expressing software designs. It builds on Z's mathematical basis to provide a small yet expressive language. Alloy's key strength is its analyzability. The SAT-based Alloy Analyzer performs scope bounded checking.

Runtime verification (RV) [57] allows synthesizing monitors from specifications for debugging as well as checking safety properties at run-time. Monitoring-oriented programming (MOP) [17] generalizes RV by supporting logic plug-ins that allow users to specify both monitor deployment and error recovery. MOP enables repair but requires the user to provide repair routines [17]. The SETL [42] programming language is based on a logic of sets; it provides sets and relations as basic datatypes and supports quantifiers. SETL users can manually direct the compiler to choose appropriate concrete datatypes for efficient execution of SETL programs. The JML checker [13] translates annotations written in the Java Modeling Language (JML) into runtime checks. The checker handles several JML constructs and can check post conditions that relate post-states with pre-states, but does not enable repair.

Assertion-based repair is closely related to specification-based analyses. (1)

Repair uses specifications for recovering from errors rather than detecting errors. (2) Unlike specification-based techniques which perform reasoning on two program states, repair reasons on a single program state, and performs mutations based on a general description of a valid state. We describe in Chapter 9 how repair can be extended to relate multiple program states and utilize existing pre-deployment techniques to perform post-deployment software analysis.

8.2 Error Recovery

Error recovery has been part of software systems for a couple of decades [63, 107]. System reboot is a traditional error recovery mechanism. In this approach, the user reboots the system when it crashes, uses system logs to analyze the cause of the problem, and creates patches to fix the errors. One disadvantage of this approach is that the system state before the crash is lost and the system returns to its initial state. Additionally, if the problematic scenario recurs, then the program is likely to reach the same corrupt state and crash again.

Check-pointing and rollback [74,122] tackles the problem of state loss when rebooting by recovering the program state to the last saved state rather than the initial one [67]. One drawback still exists when persistent, rather than volatile, faults occur in a system. In this case, it is very difficult to automate recovery using traditional approaches.

Repair is another mechanism for fault-tolerance and error recovery; several systems have featured repair over the last couple of decades [20, 43, 55, 93]. The `fsck` and `chkdsk` file system utilities check the consistency of the file structure upon booting a system and repair possible corruptions. Commercial tools such as the IBM MVS operating system [93] and the Lucent 5ESS telephone switch [55] implement dedicated components for monitoring and maintaining the consistency of the system state. DIRA [107] combines check-pointing mechanisms with dedicated routines to detect buffer overflow attacks and repair the structures damaged by the attack. A fundamental problem with the traditional approaches to repair is that repair is based on dedicated repair routines, which must be implemented for each system they are intended for. As a result, these routines are ill-understood, mostly ad-hoc, and are program specific. The problem is compounded by the absence of any text-book algorithms for repairing erroneous program states.

The use of structural integrity constraints to perform repair is relatively new. Demsky and Rinard [31, 33] are the first to use constraints as repair routines. Their framework performs repairs based on constraints written in a new declarative language that is similar to the first-order relational language Alloy [61]. Repair is performed by translating the constraints to disjunctive normal form and solving them using an ad-hoc search. To help the user formulate constraints, they have taken a promising approach [32] of integrating repair with dynamic invariant generation using Daikon [39].

Garcia [45] and Suen [110] made an initial investigation of using assertions for repair [69]. Our core repair algorithm is based on that work. The use of assertions for repair differs from the use of declarative constraints in several ways. Our work allows writing constraints using the language of implementation rather than using a declarative language which is semantically different from common programming languages and may require users to learn a new language. This allows performing repair on the heap itself and not on an abstraction of the heap and enables providing users with meaningful feedback in the form of repair logs.

8.3 Test Input Generation

Specification-based testing techniques has been present in the testing literature for a long time [50]. Many approaches automate test generation from specification languages provided in the form of program annotations [34,41,71,111,123]. These approaches are typically search-based. For example, the ASMLT [41] test input generator translates ASML specifications into finite state machines (FSM) and generates test cases by traversing the FSM states.

Korat [10] is similarly a search-based test generation tool that exhaustively enumerates all non-isomorphic instances of a data structure up to a bound on the size. Korat accepts the constraints written as a `repOk` predicate. TestEra [70] is a test generation tool that uses the Alloy Analyzer to generate all the structures that satisfy the integrity constraints. TestEra translates the class declarations of a structure into an Alloy model and the Java predicate into an Alloy formula which is then fed into the Alloy Analyzer.

Handling data constraints is always a challenge for search and SAT based approaches. Korat treats data members the same way it treats reference values, and even if the structural integrity constraints are solved, Korat still needs to perform the search to complete the structure. TestEra does not provide an efficient way to handle data elements due to the way primitive data types are modeled in Alloy [60].

A recent technique that is gaining a lot of popularity in testing literature is white-box dynamic test generation [14, 48, 106, 120, 121]. Dynamic test generation [72] consists of executing a program while gathering symbolic constraints on inputs from predicates encountered in branch statements, and of using a constraint solver to infer new program inputs from previous constraints in order to steer next executions towards *some* new program paths. This technique is now the foundation of several bug detection tools [2, 47, 49, 120]. These tools vary by the type of programs they can analyze, the type of constraints their symbolic execution can generate and by the constraint solver they use. Dynamic test generation techniques are powerful and efficient for handling primitive data yet require special handling for complex data structures.

Assertion-based repair combines the advantages of the two techniques to efficiently repair corrupt data structures. It employs a systematic search similar to the one used in Korat which is powerful for solving structural constraints, in conjunction with a symbolic execution similar to the one used for dynamic test generation which is effective for generating and solving data constraints. Repair optimizes the efficiency of the testing techniques by introducing the recurrent analysis and the checkpoint-based backtracking.

8.4 Invariant Detection

The repair algorithm expects the user to provide the integrity constraints by writing the `repOk` predicate. For complex constraints, writing a precise predicate is error-prone. Existing constraint-synthesis tools can be used to help users formulate the predicates correctly. Several static and dynamic techniques exist for synthesizing various forms of specifications, such as loop invariants [28, 105], heap abstractions [91, 103], and API level specifications [111, 118]. Daikon [39] is a popular framework for dynamically discovering likely program invariants, and has been used for data structure repair but not for linked-data structures, which our approach handles readily. Dwyer *et al.* [25] identified specification templates for temporal logic properties to assist users formulate their specifications but they do not consider error recovery.

A recently developed tool, Deryaft [83], specializes in generating constraints of complex data structures. Deryaft takes as input a handful of concrete data structures of small sizes and generates a `repOk` predicate that represents their structural integrity constraints. The constraints generated by Deryaft can be either directly used for repair, or used as a skeleton to help the users correctly formulate the `repOk` methods.

8.5 Model Checking

Structural constraint solving, state space exploration, and backtracking are commonly used techniques by software model checkers [46, 59, 100, 114].

Java PathFinder (JPF) [114] is a general purpose model checker that has also been used a solver for imperative predicates [72]. JPF performs stateful model checking of (multi-threaded) Java programs. It implements a custom Java Virtual Machine (JVM) that, unlike the standard JVM, enables non-deterministic re-executions of Java programs to, theoretically, cover all the possible executions of a program. JPF has been applied for testing data structure implementations both at concrete and abstract levels [115–117].

The implementation techniques we use in Juzi are inspired by our experience in optimizing JPF. Juzi specializes some of the concepts used by JPF to enable more efficient checking of properties. Juzi implements a lightweight backtracking mechanism by performing code instrumentation rather than implementing a custom JVM, which is required by JPF. It performs efficient incremental state saving. Rather than hashing the entire program state, and comparing it with the next state, it incrementally saves state changes and their corresponding *undo commands* as the changes occur in the program. While storing states incrementally (as “deltas”) is a known technique in explicit-state model checking [23], repair performs it at an abstract level.

Chapter 9

Future Work: Specification-based Error Recovery

We designed assertion-based repair to perform repair based on a single program state. This limited the approach to repairing data structures that can be characterized by their class invariant. This approach weakens the ability to reason about the repaired structures. The efficiency of repair using a single program state, however, suggests a potential use of assertion-based repair for repairing richer program properties. We envision an extension of assertion-based repair to program specifications. This enables reasoning on multiple program states and allows more precise analysis of the output of repair.

We propose *specification-based error recovery*, a comprehensive framework for repairing erroneous program states, which addresses the limitations of assertion-based repair and is based on a radically new role for rich behavioral specifications: to repair erroneous executions.

9.1 A New Definition of Repair

Specification-based repair requires a new definition of repair. In Chapter 3 we defined repair in terms of a `repOk` method that describes the structural integrity constraints of a data structure. The definition restricted the correctness specification to state properties of only one program state. We would like to enable repair for more general specifications, e.g., specifications that relate the pre-state of a method to a post-state, to allow recovery even after an erroneous method execution. The earlier definition also leaves the notion of similarity between the original and the repaired structure undefined. While the repair algorithm tries to minimize the perturbation to the original structure, it does not guarantee a "closest neighbor" would be generated. We would like to define the properties of the repaired structure precisely.

The new definition must include:

- Support for general specifications that relate properties across different states, e.g., adding a new key to a binary search tree modifies the tree such that all the old keys are still there and only the new key is added. Given such a specification and an incorrect execution we would like repair to generate a correct result using the specification.

- A definition of a distance-metric between graphs, which forms the basis of the repair definition with respect to a bound on the distance between the initial structure and the new structure; the metric must define distances up to isomorphism: isomorphic structures have distance zero.

9.2 The Design of a Specification Language

Specification-based repair requires the design of a language for writing repair specifications. Two key requirements must be satisfied by the language design.

First, the language must be rich enough to express complex structural and data constraints. Second, the language design must enable algorithms to provide efficient repair. There is an inherent tradeoff between these two requirements. We therefore need to find a sweet spot between expressivity and feasibility.

We foresee two directions for the language design. The first is by using relational bases to express repair [60] and using a veneer on Alloy as the specification language. The second is by extending the notion of `repOk` to handle multiple program states. Both directions have their advantages and disadvantages. While Alloy provides a powerful language for expressing program properties, the scalability of repair using Alloy remains questionable. We expect a `repOk` based approach to be more feasible in terms of performance, yet, the implementation of `repOk` methods to express the relationship between multiple program states can get very complex and becomes error prone.

9.3 The Design of Repair Algorithms

New algorithms need to be developed for performing repair based on specifications. The heart of the repair algorithms will be a mechanical translation of the repair specifications as well as the program *pre* and *post* states to an input language of a target solver. For example, if Alloy is used for writing repair specifications, then the repair algorithm must generate an Alloy model from the repair specifications and the corrupt program state and use the Alloy analyzer to repair the state. Given `repOk` specifications, the structural constraint solver described in Chapter 3 must be extended to handle multiple program states. Unlike the case of repairing a single program state, where the objects in the heap are restructured to satisfy the given constraints, repairing multiple states requires additional features such as creating new objects and extending the heap according to the specification. For example, suppose that a buggy `add` method replaces an existing element in a binary search tree. The repair algorithm must automatically add the necessary objects to retain the elements of the tree.

9.4 Assisting the Users with Writing Repair Specifications

Our design for assertion-based repair considered usability as a key requirement. The repair framework required minimal effort from the user to integrate repair in a program and provided repair logs to help the user understand the repair actions. A similar, but more powerful approach, must be taken for specification-based repair as the problem of writing specifications can get very complicated.

To help the user write correct specifications, algorithms need to be developed for checking specifications and for synthesizing skeletal specifications.

Specification checking: The repair performed at runtime is guided by the repair specifications. An erroneously formulated specification can misguide the repair algorithm. Algorithms need to be developed to run consistency analyses on repair specifications; for example, to determine whether the repair specifications are satisfiable at all. Additionally, algorithms must be developed to detect when a repair specification is too weak and inform the user to either re-write the specification to make it more precise or to inspect the repairs performed to notice any inconsistency if they choose to run repair using the original specification.

Specification synthesis: Writing detailed repair specifications can be tedious, especially for inexperienced users. Algorithms can be developed that synthesize skeletal repair specifications that the users can refine. Synthesis of skeletal specifications offers two key benefits: (1) it helps new users ease into the use of repair specifications; and (2) it reduces the burden on the user for writing specifications.

Chapter 10

Conclusions

We conclude this dissertation by providing a summary of our work on assertion-based error recovery and arguing its meaning and impact.

10.1 Summary

We started our work by designing a core algorithm for assertion-based repair (Chapter 3). The algorithm combines systematic search and symbolic execution to repair corrupt data structures. Our key target while designing the algorithm was the correctness of repair. The core repair algorithm is sound and complete with respect to the given structural constraints. While the algorithm effectively repairs data structures, the size of the structures is limited to those with up to few thousands of nodes.

The next steps were to develop a framework for assertion-based repair that could apply repair to general purpose Java programs and to devise optimizations to improve the efficiency of the repair algorithm.

We developed Juzi (Chapter 4), a framework for repairing Java programs. Juzi implements the core repair algorithm described in Chapter 3 and uses byte-code instrumentation to integrate it into existing Java code. Juzi also provides programmers with the necessary API for writing repairable classes and asserting their properties at runtime.

A powerful feature of Juzi is that users control the repair algorithm. Using Juzi, the user can control the fields that the repair algorithm mutates, the data it introduces, the order of the field mutations, as well as the repair-logs which can range from a log file that contains a summary of the repair actions to a visualization that allows the user to interact with the repair algorithm. The binaries of the Juzi framework along with a usage tutorial can be found at `http:\\www.ece.utexas.edu\~elkarabl`.

To enhance the efficiency of the core repair algorithm we introduced a set of key optimizations (Chapter 5) for scaling the performance of the original repair algorithm to handle large data structures with hundreds of faults. The first optimization was through a static analysis that identifies recurrent fields of the target data structure and uses the information of the static analysis to guide the search to good repair candidates. The second optimization was through an efficient backtracking engine for repair. In contrast to re-execution-based approaches for backtracking, it performs checkpoint-based backtracking by storing partial program states and performing abstract undo operations. The heart of our approach is a light-weight search that is performed purely through code instrumentation and that does not require special JVM support.

We evaluated the efficiency of repair by using it to repair corruptions in a diverse set of library data structures (Chapter 6). The experimental results demonstrated the scalability of the approach when repairing large data structures with randomly injected faults in a feasible amount of time. We also evaluated the acceptability of the repaired results by using repair on three stand-alone applications. The results showed that for applications with redundancy in their data structures, the structures were fully repaired. For structures with corruptions in the data fields, the structures were mutated but the program execution proceeded safely.

The efficiency of repair directed our attention to alternative uses of repair in testing. We leverage the efficiency of repair for constraint based generation of large data structures (Chapter 7). The work is inspired from the experiments on repairing randomly injected faults. We introduce repair-based generation which uses our approach to repair randomly generated graphs. By separating the generation tasks, repair based generation enables combining random graph generation with data structure repair and constraint solving, to efficiently generate large data structures based on specifications.

Experiments on generating large data structures using subjects with complex structural and data constraints show that repair-based generation can efficiently generate structures with up to a hundred thousand nodes. In comparison with two existing constraint-based generation frameworks, repair-based generation is able to generate structures that are up to 100 times larger.

The evaluation of assertion-based repair provides solid evidence of the feasibility of repair. Our experience with repair suggested an extension of the repair approach to program specifications. While prior uses of specifications have been numerous, ranging from documentation, to testing, to runtime checking, rich behavioral specifications have not previously been used for error recovery.

10.2 Meaning

Methodologies that improve software reliability not only provide substantial economic benefits but also improve our quality of life. To bring about a real change in the current state of unreliable software, we must equip developers with state-of-the-art tools as well as sound foundations in reasoning and logic.

Assertion-based repair leads to a substantial advance in our ability to develop correct programs. For programs that already have assertions, error recovery using the proposed approach can come for free. The same assertion can be used for checking code before deployment using existing techniques as well as ensuring the program executions do not go awry after deployment. Thus, this approach enables a novel unification of software verification and error recovery. Such a unification has not been possible before and is likely to substantially improve the quality of software.

The benefits of assertions are widely recognized, but for the most part they have not been realized, and programmers still view them as more trouble than they are worth. Much progress has been made. Assertions are now better integrated with programming languages; and they can handle the complexities of object-oriented code. But to make them attractive to practitioners, we believe it is necessary to squeeze more value from them, by providing new analyses for the same assertions. The ability to recover from errors on-the-fly can make assertions significantly more attractive and beneficial, and a fundamental element of developing dependable software.

Bibliography

[1] William Adjie-Winoto, Elliot Schwartz, Hari Balakrishnan, and Jeremy Lilley. The design and implementation of an intentional naming system. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles (SOSP)*, Kiawah Island, SC, December 1999.

[2] Saswat Ananad, Patrice Godefroid, and Nikolai Tillmann. Demand-driven compositional symbolic execution. In *Proceedings of the 14th Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS)*, Budapest, Hungary, March 2008.

[3] Tony Andrews, Shaz Qadeer, Sriram Rajamani, Jakob Rehof, and Yichen Xie. Zing: A model checker for concurrent software. In *Proceedings of the 16th International Conference on Computer Aided Verification (CAV)*, Boston, MA, July 2004.

[4] Cyrille Artho, Howard Barringer, Allen Goldberg, Klaus Havelund, Sarfraz Khurshid, Mike Lowry, Corina Pasareanu, Grigore Rosu, Koushik Sen, Willem Visser, and Rich Washington. Combining test case generation and runtime verification. *Theoretical Computer Science*, 2005.

[5] Thomas Ball and Sriram Rajamani. Automatically validating temporal safety properties of interfaces. In *Proceedings of the 8th International Workshop*

on Model Checking of Software (SPIN), Toronto, Canada, May 2001.

[6] 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 (CAV)*, Boston, MA, July 2004.

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

[8] Mike Berger. Borg calendar. `http://borg-calendar.sourceforge.net/`.

[9] Stephen M. Blackburn, Robin Garner, Chris Hoffmann, Asjad M. Khang, 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, B. Moss, Aashish Phansalkar, Darko Stefanović, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiedermann. The DaCapo benchmarks: Java benchmarking development and analysis. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2006.

[10] Chandrasekhar Boyapati, Sarfraz Khurshid, and Darko Marinov. Korat: Automated testing based on Java predicates. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, Rome, Italy, July 2002.

[11] Olga Brukman, Shlomi Dolev, and Marcelo Sihman. Recovery oriented programming. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP)*, Brighton, United Kingdom, October 2005.

[12] Eric Bruneton, Romain Lenglet, and Thierry Coupaye. ASM: a code manipulation tool to implement adaptable systems. In *Adaptable and extensible component systems*, Grenoble, France, November 2002.

[13] Lilian Burdy, Yoonsik Cheon, David Cok, Michael Ernst, Joe Kiniry, Gary Leavens, K. Rustan M. Leino, and Erik Poll. An overview of JML tools and applications. *International Journal on Software Tools for Technology Transfer (STTT)*, 2005.

[14] Cristian Cadar, Vijay Ganesh, Peter Pawlowski, David Dill, and Dawson Engler. Exe: Automatically generating inputs of death. *ACM Transactions on Information and System Security*, 2008.

[15] Brendon Cahoon. *Effective Compile-Time Analysis for Data Prefetching in Java*. PhD thesis, University of Massachusetts, Amherst, MA, 2002.

[16] Brendon Cahoon and Kathryn S. McKinley. Recurrence analysis for effective array prefetching in Java. *Concurrency and Computation Practice and Experience*, 17, February 2005.

[17] Feng Chen and Grigore Roşu. Java-MOP: A monitoring oriented programming environment for Java. In *Proceedings of the 11th Conference on Tools*

and Algorithms for Construction and Analysis of Systems (TACAS), Edinburgh, Scotland, April 2005.

[18] Yoonsik Cheon and Gary Leavens. A simple and practical approach to unit testing: The JML and JUnit way. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, Malaga, Spain, June 2002.

[19] Shigeru Chiba. Javassist—a reflection-based programming wizard for Java. In *Proceedings of the ACM OOPSLA'98 Workshop on Reflective Programming in C++ and Java*, October 1998.

[20] Microsoft chkdsk manual page. Chkdsk.

[21] Edmund Clarke, Orna Brumberg, and Doron Peled. *Model Checking*. MIT Press, 1999.

[22] Edmund Clarke, Daniel Kroening, and Flavio Lerda. A tool for checking ANSI-C programs. In *Proceedings of the 10th Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS)*, Barcelona, Spain, March 2004.

[23] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. The MIT Press, Cambridge, MA, 1999.

[24] Lee Copeland. *A Practitioner's Guide to Software Test Design*. Artech House, 2003.

[25] James Corbett, Matthew Dwyer, John Hatcliff, Corina Pasareanu, Robby, Shawn Laubach, and Hongjun Zheng. Bandera: Extracting finite-state models from Java source code. In *Proceedings of the 22nd International Conference on Software Engineering (ICSE)*, Limerick, Ireland, June 2000.

[26] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. The MIT Press, Cambridge, MA, 1990.

[27] Standard Performance Evaluation Corporation. *SPECjvm98 Documentation*. Release 1.03 edition, March 1999.

[28] Patrick Cousot and Nicolas Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Proceedings of the 5th Annual ACM Symposium on the Principles of Programming Languages (POPL)*, Tucson, Arizona, 1978.

[29] Christoph Csallner and Yannis Smaragdakis. Dynamically discovering likely interface specifications. In *Proceedings of the 28th International Conference on Software Engineering (ICSE)*, Shanghai, China, 2006. Emerging results track.

[30] Paul Darga and Chandrasekhar Boyapati. Efficient software model checking of data structure properties. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, Portland, OR, 2006.

[31] Brian Demsky. *Data structure repair using goal-directed reasoning*. PhD thesis, Massachusetts Institute of Technology, January 2006.

[32] Brian Demsky, Michael Ernst, Philip Guo, Stephen McCamant, Jeff Perkins, and Martin Rinard. Inference and enforcement of data structure consistency specifications. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, July 2006.

[33] Brian Demsky and Martin Rinard. Automatic detection and repair of errors in data structures. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, Anaheim, CA, 2003.

[34] Michael Donat. Automating formal specification based testing. In *Proceedings of the 7th International Conference on Theory and Practice of Software Development (TAPSOFT)*, Lille, France, 1997.

[35] Bassem Elkarablieh, Iván García, Yuk Lai Suen, and Sarfraz Khurshid. Assertion-based repair of structurally complex data. In *Proceedings of the 22th Conference on Automated Software Engineering (ASE)*, Atalanta, Georgia, November 2007.

[36] Bassem Elkarablieh, Sarfraz Khurshid, Duy Vu, and Kathryn S. McKinley. Starc: Static analysis for efficient repair of complex data. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, Montereal, Canada, October 2007.

[37] Myrvin Ellestad, editor. *Stress Testing: Principles and Practice*. Oxford University Press, USA; 5th edition, 2003.

[38] Michael Ernst, Jeff Perkins, Philip Guo, Stephen McCamant, Carlos Pacheco, Matthew Tschantz, and Chen Xiao. The Daikon system for dynamic detection of likely invariants. *Science of Computer Programming*, 2007.

[39] Michael D. Ernst, Jake Cockrell, William G. Griswold, and David Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE Transactions on Software Engineering*, 2001.

[40] The Eclipse Foundation. The Eclipse development platform. `http://www.eclipse.org/`.

[41] Foundations of Software Engineering, Microsoft Research. The AsmL test generator tool. `http://research.microsoft.com/fse/asml/doc/AsmLTester.html`.

[42] Stefan Freudenberger, Jacob Schwartz, and Micha Sharir. Experience with the SETL optimizer. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 1983.

[43] Ext2 fsck manual page. `http://e2fsprogs.sourceforge.net/`.

[44] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional Computing Series. Addison-Wesley Publishing Company, New York, NY, 1995.

[45] Iván García. Enabling symbolic execution of Java programs using bytecode instrumentation. Master's thesis, The University of Texas at Austin, May 2005.

[46] Patrice Godefroid. Model checking for programming languages using VeriSoft. In *Proceedings of the 24th Annual ACM Symposium on the Principles of Programming Languages (POPL)*, Paris, France, January 1997.

[47] Patrice Godefroid, Adam Kieżun, and Michael Y. Levin. Grammar-based whitebox fuzzing. In *Proceedings of the ACM SIGPLAN'08 Conference on Programming Language Design and Implementation (PLDI)*, Tucson, AZ, USA, 2008.

[48] Patrice Godefroid, Nils Klarlund, and Koushik Sen. Dart: directed automated random testing. In *Proceedings of the ACM SIGPLAN'05 Conference on Programming Language Design and Implementation (PLDI)*, Chicago, IL, 2005.

[49] Patrice Godefroid, Michael Levin, and David Molnar. Automated whitebox fuzz testing. In *Proceedings of the 15th Annual Symposium on Network and Distributed System Security (NDSS)*, 2008.

[50] John Goodenough and Susan Gerhart. Toward a theory of test data selection. *IEEE Transactions on Software Engineering*, June 1975.

[51] Pete Goodliffe. *Code Craft: The Practice of Writing Excellent Code*. No Starch Press, San Francisco, CA, USA, 2006.

[52] Sudhakar Govindavajhala and Andrew Appel. Using memory errors to attack a virtual machine. In *Proceedings of the Symposium on Security and Privacy (SSP)*, 2003.

[53] Andreas Griesmayer, Roderick Bloem, and Byron Cook. Repair of boolean programs with an application to C. In *Proceedings of the 18th International Conference on Computer Aided Verification (CAV)*, 2006.

[54] Yuri Gurevich. Evolving algebras 1993: Lipari guide. In *Specification and Validation Methods*. Oxford University Press, 1995.

[55] G. Haugk, F. Lax, R. Royer, and J. Williams. The 5ESS(TM) switching system: Maintenance capabilities. *AT&T Technical Journal*, 64(6 part 2), 1985.

[56] Klaus Havelund and Thomas Pressburger. Model checking Java programs using Java PathFinder. *International Journal on Software Tools for Technology Transfer*, 1999.

[57] Klaus Havelund and Grigore Rosu. *Workshops on Runtime Verification (RV'01, RV'02, RV'04*. 2001, 2002, 2004.

[58] Mats Heimdahl, Sanjai Rayadurgam, Willem Visser, Devaraj George, and Jimin Gao. Auto-generating test sequences using model checkers: A case study. In *Proceedings of the 3rd International Workshop on Formal Approaches to Testing of Software (FATES)*, Montreal, Canada, October 2003.

[59] Gerald Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5), May 1997.

[60] Daniel Jackson. Alloy: A lightweight object modeling notation. *ACM Transactions on Software Engineering and Methodology*, April 2002.

[61] Daniel Jackson. *Software Abstractions: Logic, Language and Analysis*. The MIT Press, Cambridge, MA, 2006.

[62] Daniel Jackson, Martyn Thomas, and Lynette I. Millett, editors. *Software for Dependable Systems: Sufficient Evidence?* Committee on Certifiably Dependable Software Systems, National Research Council, 2007.

[63] Barbara Jobstmann, Andreas Griesmayer, and Roderick Bloem. Program repair as a game. In *Proceedings of the 17th International Conference on Computer Aided Verification (CAV)*, 2005.

[64] Cliff Jones. *Systematic Software Development using VDM*. Prentice-Hall, 1990.

[65] Maria Jump and Kathryn S. McKinley. Dynamic shape analysis via degree metrics. In *Proceedings of the International Symposium on Memory Management (ISMM)*, Dublin, Ireland, 2009.

[66] Yamini Kannan and Koushik Sen. Universal symbolic execution and its application to likely data structure invariant generation. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, Seattle, WA, 2008.

[67] Feras Karablieh, Rida A. Bazzi, and Margaret Hicks. Compiler-assisted heterogeneous checkpointing. In *Proceedings of the 20th Symposium on Reliable Distributed Systems (SRDS)*, October 2001.

[68] Shadi Abdul Khalek, Bassem Elkarablieh, Ola Laleye, and Sarfraz Khurshid. Query-aware test generation using a relational constraint solver. In *Proceedings of the 23th Conference on Automated Software Engineering (ASE)*, Sept 2008.

[69] Sarfraz Khurshid, Iván García, and Yuk Lai Suen. Repairing structurally complex data. In *Proceedings of the 12th International Workshop on Model Checking of Software (SPIN)*, San Francisco, CA, August 2005.

[70] Sarfraz Khurshid and Darko Marinov. Checking Java implementation of a naming architecture using TestEra. In *Electronic Notes in Theoretical Computer Science (ENTCS)*, volume 55. Elsevier Science Publishers, 2001.

[71] Sarfraz Khurshid and Darko Marinov. TestEra: Specification-based testing of Java programs using SAT. *Automated Software Engineering Journal*, 2004.

[72] Sarfraz Khurshid, Corina Pasareanu, and Willem Visser. Generalized symbolic execution for model checking and testing. In *Proceedings of the 9th Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS)*, Warsaw, Poland, April 2003.

[73] Sarfraz Khurshid and Yuk Lai Suen. Generalizing symbolic execution to library classes. In *Proceedings of the 6th ACM Workshop on Program Analysis for Software Tools and Engineering (PASTE)*, Lisbon, Portugal, September 2005.

[74] Junguk Kim and Taesoon Park. An efficient protocol for checkpointing recovery in distributed systems. *IEEE Transactions on Parallel and Distributed Systems*, Aug 1993.

[75] James C. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7), 1976.

[76] Eugene Kuleshov. Using ASM framework to implement common bytecode transformation patterns. In *Proceedings of the 6th International Conference on Aspect-Oriented Software Development (AOSD)*, Vancouver, Canada, 2007.

[77] Gary Leavens. An overview of Larch/C++: Behavioral specifications for C++ modules. In *Specification of Behavioral Semantics in Object-Oriented Information Modeling*. Kluwer Academic Publishers, 1996.

[78] Gary Leavens, Albert Baker, and Clyde Ruby. Preliminary design of JML: A behavioral interface specification language for Java. Technical Report TR 98-06i, Department of Computer Science, Iowa State University, June 1998.

[79] Flavio Lerda and Willem Visser. Addressing dynamic issues of program model checking. In *Proceedings of the 8th International Workshop on Model Checking of Software (SPIN)*, Toronto, Canada, May 2001.

[80] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Addison Wesley, second edition, 1999.

[81] Barbara Liskov and John Guttag. *Program Development in Java: Abstraction, Specification, and Object-Oriented Design*. Addison-Wesley, 2000.

[82] Joshua Madadhain, Danyel Fisher, and Tom Nelson. Java universal network/graph framework. `http://jung.sourceforge.net/index.html`.

[83] Muhammad Zubair Malik, Aman Pervaiz, and Sarfraz Khurshid. Generating representation invariants of structurally complex data. In *Proceedings of the 13th Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS)*, Braga, Portugal, March 2007.

[84] Muhammad Zubair Malik, Aman Pervaiz, Engin Uzuncaova, and Sarfraz Khurshid. Deryaft: a tool for generating representation invariants of structurally complex data. In *Proceedings of the 30th International Conference on Software Engineering (ICSE)*, Leipzig, Germany, 2008.

[85] Darko Marinov. *Automatic Testing of Software with Structurally Complex Inputs*. PhD thesis, Computer Science and Artificial Intelligence Laboratory, Massachusetts Institute of Technology, 2004.

[86] Darko Marinov, Alexandr Andoni, Dumitru Daniliuc, Sarfraz Khurshid, and Martin Rinard. An evaluation of exhaustive testing for data structures. Tech-

nical Report MIT-LCS-TR-921, MIT CSAIL, Cambridge, MA, September 2003.

[87] Darko Marinov and Sarfraz Khurshid. TestEra: A novel framework for automated testing of Java programs. In *Proceedings of the 16th Conference on Automated Software Engineering (ASE)*, San Diego, CA, November 2001.

[88] Bertrand Meyer. *Eiffel: The Language*. Prentice Hall, New York, N.Y., 1992.

[89] Aleksandar Milicevic, Sasa Misailovic, Darko Marinov, and Sarfraz Khurshid. Korat: A tool for generating structurally complex test inputs. In *Proceedings of the 29th International Conference on Software Engineering (ICSE)*, Washington, DC, 2007.

[90] Sasa Misailovic, Aleksandar Milicevic, Nemanja Petrovic, Sarfraz Khurshid, and Darko Marinov. Parallel test generation and execution with Korat. In *Proceedings of the 15th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE)*, September 2007.

[91] Anders Moeller and Michael I. Schwartzbach. The pointer assertion logic engine. In *Proceedings of the ACM SIGPLAN'01 Conference on Programming Language Design and Implementation (PLDI)*, Snowbird, UT, June 2001.

[92] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an efficient SAT solver. In *Proceedings*

of the 38th Conference on Design Automation (DAC), June 2001.

[93] Samiha Mourad and Dorothy Andrews. On the reliability of the IBM MVS/XA operating system. *IEEE Transactions on Software Engineering*, 13(10), 1987.

[94] Peter Müller, Arnd Poetzsch-Heffter, and Gary Leavens. Modular specification of frame properties in JML. Technical Report 02-02, Iowa State University, February 2002.

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

[96] Jeff Offutt and Aynur Abdurazik. Generating tests from UML specifications. In *Proceedings of the 2nd International Conference on the Unified Modeling Language*, October 1999.

[97] Corina Pasareanu and Willem Visser. Verification of Java programs using symbolic execution and invariant generation. In *Proceedings of the 11th International Workshop on Model Checking of Software (SPIN)*, Barcelona, Spain, April 2004.

[98] Apache J. Project. The byte code engineering library.

[99] The Jython Project. `http://www.jython.org/`.

[100] Robby, Matthew B. Dwyer, and John Hatcliff. Bogor: An extensible and highly-modular software model checking framework. In *Proceedings of the 11th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE)*, Helsinki, Finland, September 2003.

[101] Robby, Edwin Rodríguez, Matthew Dwyer, and John Hatcliff. Checking strong specifications using an extensible software model checking framework. In *Proceedings of the 10th Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS)*, Barcelona, Spain, March 2004.

[102] James Rumbaugh, Ivar Jacobson, and Grady Booch. *The Unified Modeling Language Reference Manual*. Addison-Wesley Object Technology Series, 1998.

[103] Mooly Sagiv, Thomas Reps, and Reinhard Wilhelm. Solving shape-analysis problems in languages with destructive updating. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, January 1998.

[104] Roopsha Samanta, Jyotirmoy Deshmukh, and Ellen Emerson. Automatic generation of local repairs for boolean programs. In *Proceedings of the 9th Formal Methods in Computer-Aided Design (FMCAD)*, Nov 2008.

[105] Sriram Sankaranarayanan, Henny Sipma, and Zohar Manna. Non-linear loop invariant generation using grobner bases. In *Proceedings of the 31th Annual ACM Symposium on the Principles of Programming Languages (POPL)*, Venice, Italy, 2004.

[106] Koushik Sen, Darko Marinov, and Gul Agha. CUTE: a concolic unit testing engine for C. In *Proceedings of the 13th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE)*, Lisbon, Portugal, 2005.

[107] Alexey Smirnov and Tzi-cker Chiueh. DIRA: Automatic detection, identification, and repair of control-hijacking attacks. In *Proceedings of the 12th Annual Symposium on Network and Distributed System Security (NDSS)*, San Diego, CA, February 2005.

[108] Graeme Smith. *The Object-Z Specification Language*. Kluwer Academic Publishers, 2000.

[109] Mike Spivey. *The Z Notation: A Reference Manual*. Prentice Hall, second edition, 1992.

[110] Yuk Lai Suen. Automatically repairing structurally complex data. Master's thesis, Department of Electrical and Computer Engineering, The University of Texas at Austin, May 2005.

[111] Mana Taghdiri. Inferring specifications to detect errors in code. In *Proceedings of the 19th Conference on Automated Software Engineering (ASE)*, Washington, DC, 2004.

[112] The HSQLDB Development Group. HSQL database engine. `http://www.hsqldb.org/`.

[113] Emina Torlak and Daniel Jackson. Kodkod: A relational model finder. In *Proceedings of the 13th Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS)*, Braga, Portugal, March 2007.

[114] Willem Visser, Klaus Havelund, Guillaume Brat, and SeungJoon Park. Model checking programs. In *Proceedings of the 15th Conference on Automated Software Engineering (ASE)*, Grenoble, France, 2000.

[115] Willem Visser, Corina Pasareanu, and Sarfraz Khurshid. Test input generation with Java PathFinder. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, Boston, MA, 2004.

[116] Willem Visser, Corina Păsăreanu, and Radek Pelánek. Test input generation for red-black trees using abstraction. In *Proceedings of the 20th Conference on Automated Software Engineering (ASE)*, Long Beach, CA, USA, 2005.

[117] Willem Visser, Corina Păsăreanu, and Radek Pelánek. Test input generation for Java containers using state matching. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, Portland, Maine, 2006.

[118] John Whaley, Michael Martin, and Monica Lam. Automatic extraction of object-oriented component interfaces. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, July 2002.

[119] Joan Winston and Lynette Millett, editors. *Summary of a Workshop on Software-Intensive Systems and Uncertainty at Scale*. National Research

Council, National Research Council, 2007.

[120] Tao Xie, Darko Marinov, and David Notkin. Rostra: A framework for detecting redundant object-oriented unit tests. In *Proceedings of the 19th Conference on Automated Software Engineering (ASE)*, Linz, Austria, September 2004.

[121] Tao Xie, Darko Marinov, Wolfram Schulte, and David Notkin. Symstra: A framework for generating object-oriented unit tests using symbolic execution. In *Proceedings of the 11th Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS)*, Edinburgh, Scotland, April 2005.

[122] Guoqing Xu, Atanas Rountev, Yan Tang, and Feng Qin. Efficient checkpointing of Java software using context-sensitive capture and replay. In *Proceedings of the 15th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE)*, Dubrovnik, Croatia, 2007.

[123] Guoqing Xu and Zongyuan Yang. JMLAutoTest: A novel automated testing framework based on JML and JUnit. In *Proceedings of the 3rd International Workshop on Formal Approaches to Testing of Software (FATES)*, Montreal, Canada, October 2003.

[124] Fadi Zaraket, Adnan Aziz, and Sarfraz Khurshid. Sequential circuits for relational analysis. In *Proceedings of the 29th International Conference on Software Engineering (ICSE)*, May 2007.

Vita

Bassem was born in Beirut, Lebanon in August 1982. He received his Bachelor of Engineering in July 2004 from the Lebanese American University, and his Masters of Engineering degree in May 2006 from Syracuse University, New York. Bassem has interned with Intel Corp. in summer 2006, with Google Corp. in summer 2007, and with Microsoft Research in summer 2008. His research interests focus on discovering new techniques for increasing the reliability and degree of confidence in software systems. Bassem enjoys playing soccer, and he is an excellent chess player.

Permanent address: 3517 North Hills Dr.
Austin, Texas 78731

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

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