DYNAMIC SYMBOLIC DATA STRUCTURE REPAIR AND EVALUATION OF PROGRAM ANALYSIS TOOLS WITH THE RUGRAT RANDOM PROGRAM GENERATOR

by

ISHTIAQUE HUSSAIN

Presented to the Faculty of the Graduate School of
The University of Texas at Arlington in Partial Fulfillment
of the Requirements
for the Degree of

DOCTOR OF PHILOSOPHY

THE UNIVERSITY OF TEXAS AT ARLINGTON

May 2014

To all my teachers, starting from my parents until now, who set the example and made me who I am.

Acknowledgements

I am grateful to so many people. Without their help, motivation, continuous support I could not have managed to do this dissertation.

I am simply blessed to have Professor Christoph Csallner as my supervisor. I came to the University of Texas at Arlington (UTA) as a BS–to-Ph.D. student in 2008. As a fresh college graduate I had almost zero knowledge in research, as well as program analysis and software engineering. Everything was new to me. Christoph literally taught me things in these areas bit by bit. He patiently held long office hours just to explain things, answer my naive questions and clarify concepts that I have learned in my undergraduate studies. He even taught me etiquettes in official communications, reviewed my resume and corrected it for my internship and job applications. Overall, he helped me in all aspects. I could never repay or thank him enough for what he did for me. I will try to emulate his work ethics and sincerity to wherever I go and serve.

My committee members, Dr. David Kung, Dr. Jeff (Yu) Lei and Dr. Bahram Khalili were very generous with their helpful comments and suggestions. I was Dr. Kung's teaching assistant for several of his undergraduate and graduate software engineering courses and I learned a lot from them. Earlier, I had Dr. Donggang Liu and Dr. Nate Nystrom in my committee and they were equally helpful with their guidance. Thank you!

I took several graduate level courses as part of my course requirements in UTA. They were very helpful and I enjoyed them. But I would like to especially thank Professor Gautam Das for his algorithms course. It was one of the most interesting, elaborate course that I took in UTA and it helped me overcome my fear of algorithms. I learned a lot from his course and later it helped me get the summer internship at Microsoft.

During my internship at the Accenture Technology Labs in Chicago, I met Dr. Mark Grechanik, Dr. Chen Fu and Dr. Qing Xie. Mark was my manager and he was very friendly. He has a very good sense of humor and with his never ending jokes, he made the work environment lively. I learned a lot from Mark, Chen and Qing during the internship. I also made good friends with Kunal Taneja and Sangmin Park, who were Ph.D. students, and were also doing internships at Accenture at that time. They are really smart and helped me a lot to overcome my limitations. This internship was very successful as we published few research papers together afterwords. Thank you guys!

My internship at Microsoft, on the other hand, didn't really help in this dissertation. But I met many smart people and learned a lot from them. I would like to take this chance to thank them, especially Soham Mehta, Bowen Zhang, Carter Vogds, Rob Cooper and Brent Jensen for their generous support. Soham and Bowen would come to help me out even when they were under tremendous work pressure. I also greatly appreciate the help and suggestions of Mohammad Saiful Islam and Protik Mohammad Hossain at Microsoft. The internship at Microsoft was my first real industry experience. I will treasure this experience.

Pam McBride, Sherri Gotcher, Camille Costabile, Bito Irie, Skipeer Harris and others from the department's administration, office and technical staff were always generous and very helpful to me. My sincere gratitude to these 'behind the scene' people who make things happen!

Luke Tesdal from the Writing Center at UTA reviewed my dissertation and helped me improve it with his numerous writing suggestions. I also attended his presentation on writing style, citation rules, etc. which was really helpful. Thank you!

My friends Mirza and Mishuk, my cousin Ashique and my uncle Maruf helped me in many ways from the very beginning of my journey. Since online payment was not easy from Bangladesh back then, they helped me pay for the GRE and TOEFL exams and application fee for UTA. Once I came to the U.S., they even lend me money to pay the tuition fees and other expenses for first few months when my first paycheck was delayed. I will always remember these helps and be grateful.

I am also grateful to my lab mates: Steven, Mainul, Sarker, Mehra, Shabnam, Derek, Laleh, Tony, Jhing and Luke for their support and help. The group studies, practice talks and presentations and finally preparing for the job interviews together were very helpful. Thank you guys!

During my studies I lost few family members and a good school friend in Bangladesh who were always a great source of motivation for me. Among them my dadi (grandmother), shejo-mama (uncle Sadiqur Rahman), cousin Sayeem-bhai and my disabled cousins Moni, Reema Apa and my friend Raza would have been very happy for me today. I miss them a lot and always pray for their departed souls.

My parents, Dr. Altaf Hussain and Shamim Ara Hussain and my siblings, Nazia (and her husband Mizan) and Sakhawat were always there to help me. Watching my nephew Nazeeb grow with his daily new activities were a good source of entertainment and recreation. Reem, my wife supported me all the time. Her patience was amazing! Thank you!

Finally, I would like to thank the government of People's Republic of Bangladesh and its people for supporting my studies up to the undergraduate level, almost for free.

April 21, 2014

Abstract

DYNAMIC SYMBOLIC DATA STRUCTURE REPAIR AND EVALUATION OF PROGRAM ANALYSIS TOOLS WITH THE RUGRAT RANDOM PROGRAM GENERATOR

ISHTIAQUE HUSSAIN, Ph.D.

The University of Texas at Arlington, 2014

Supervising Professor: Christoph Csallner

Generic automatic repair of complex data structures is a new and exciting area of research. Existing approaches can integrate with good software engineering practices such as program assertions. But in practice there is a wide variety of assertions and not all of them satisfy the style rules imposed by existing repair techniques. That is, a badly written assertion may render generic repair inefficient or ineffective. Moreover, the performance of existing approaches may depend on the location of an error in a corrupted data structure. This dissertation shows that generic automatic data structure repair can be implemented with full dynamic symbolic execution. Such an implementation can solve some of the problems of the existing generic repair approaches.

The dissertation also evaluates the usefulness of a novel random program generator, *RUGRAT-Random Utility Generator for Program Analysis and Testing*, for the evaluation and benchmarking of different Java source–to–bytecode compilers and *pRogram Analysis and Testing (RAT)* tools. It generates several programs in different size categories, ranging up to 5MLOC and uses them to compare and find bugs in the various RAT tools.

Table of Contents

List of Illustrations

List of Tables

Chapter 1

Introduction

1.1 My Thesis

Generic automatic data structure repair can be implemented with full dynamic symbolic execution. Such an implementation can solve some of the problems of the existing generic repair approaches.

Random program generators can create benchmarks with different sizes, properties and complexities. These benchmarks can be used to evaluate performance and find bugs in the different program analysis and testing tools.

1.2 Importance of Software Reliability

In todays modern world, software is everywhere. From as little significant devices as digital wristwatches to those more important as home appliances, hand held devices, telecommunications, automobiles, industries, office and businesses, to mission critical applications such as space crafts, flights and life saving health care systems - software running on top of computer hardware play vital roles. And because of the proliferation of such technologies in our daily lives, our reliance on these systems has never been greater.

Software reliability is defined as the probability of failure-free software operation for a specified period of time in a specified environment [134, 105]. Lack of software reliability due to software failures cost the US economy billions of dollars annually and with the rising dependency of businesses on software, it is expected to further increase [122, 86, 170, 54].

While many software failures may cause inconvenience, some software reliability issues can have severe consequences. Reliability of mission critical applications are crucial. For example, the European Space Agency's Ariane 5 Flight 501, a 1 billion US dollar prototype was destroyed just after launch because of a failure of its on-board guidance software [52]. These systems do not allow user intervention when a failure occurs and it is often hard (or impossible) to update them once they are deployed or in use. Similarly, reliability of health care systems is very important. An unreliable system can even cost human lives. For example, the Therac-25 radiation therapy machine was directly responsible for at least five patient deaths in the 1980s when it administered excessive quantities of X-rays due to an error in the code [99].

1.3 Threats to Software Reliability

Software failures that reduce software reliability can occur for various reasons. They may be due to errors, ambiguities, oversights or misinterpretations of the specification that the software is supposed to satisfy, carelessness or incompetence in writing code, inadequate testing, incorrect or unexpected usage of the software, or other unforeseen problems [125, 92]. Some software errors can be found quickly but others may remain undetected and occur only rarely after years of successful operation [83]. Similarly, while some hardware failures can crash a system, others may be almost benign and just flip a few bits in the memory.

Every software manipulates some sort of data structures and while a software executes, it often assumes the data structure to be in the correct states. However, at runtime the underlying data structures can dynamically get into inconsistent states and if not handled properly, can have dire consequences such as data loss, program halt, unexpected behavior, security breaches and even software crash. A rare undetected bug or a hardware failure that caused a few bits to flip in the memory can cause such inconsistencies.

Even without the mentioned errors, the data structure state can get corrupted. For example, a single bit flip due to a cosmic ray is considered to be the reason behind an inconsistent state in an aircraft software that resulted in the plane plunging few hundred feet several times during flight injuring its crew and passengers [2]. Though bit flips in [2] are caused by random environmental events, an attacker who has access to the physical device (e.g., smart card, palmtop or desktop) can also provoke such a bit flip by heating up the memory chip, compromise the safety of a Java Virtual Machine, and run arbitrary code [70]. It is reported [4] that microchip manufacturers have taken this seriously and are planning to build in-chip detectors for such flip bits. However, this might not be enough as the communication busses between processors and memory units would be still exposed to errors or attacks [70].

1.4 Overview of Existing Approaches to Detect and Repair of Corruptions

There are many approaches for detecting and repairing software corruptions. However, existing approaches are partial solutions and have their strengths and limitations. In this section we briefly discuss these approaches and their strengths and limitations.

1.4.1 Overview on Corruption Detection Approaches

To detect bugs, software developers can use assertions (e.g., user input verification and validation, pre and post–condition of a method, etc.), correctness condition (which is popularly known as *repOk* method [103]) that checks program consistency, or simply wait for the system to produce an incorrect, unexpected result or behave in an unintended way.

Software testing is also conducted primarily to detect bugs [120]. It is part of the software development process that provides the stakeholders with the information about the quality of the product or service under test [89]. However, software testing may not identify all the defects within a software [125, 129] and hence cannot guarantee a software will be bug free [8]. For example, software testing cannot detect the bit flips due to cosmic rays.

There are numerous software testing approaches including static and dynamic testing, manual and automatic testing, automated static analysis and dynamic-symbolic testing, etc. A detailed discussion of these approaches is out of the scope of this dissertation and can be found elsewhere: [16, 174, 120].

1.4.2 Overview on Repair Approaches

Depending in which phase of the software development process a bug is detected, the cost to fix it varies and it is perceived that the earlier the bug is detected, the cheaper it is to fix [113]. During the development phase, once a bug is detected it is usually the programmer who manually fixes it [8].

To lower the probability of developers making identical software faults in developing a software, researchers have proposed an approach called N-version programming [33]. The main idea of N-version programming is that from the specifications, independent teams will develop different versions of the same functionality of the software, probably using different algorithms and programming languages, and at the various program points the execution environment would choose between the outputs of the different versions [33, 11]. There are mainly two components of N-version programming approach: first, detecting that different versions returned different results and second, deciding on the final output (e.g., this could be taking the result of the majority versions or using some other complex decision algorithms [101]).

Although there are criticisms of the N-version programming approach; for instance, that it does not lower the probability of developers making identical software faults in developing a software, i.e., different teams can make similar mistakes [121, 95], it has been used in developing different software, e.g., software performing flight control computations, switching trains and electronic voting systems, etc. [101, 11, 121].

Researchers have proposed automatic repair approaches for buggy software [167, 143, 152, 151]. Weimer et al. [167] proposed a technique that utilizes a set of successful test cases and a single failing test case to locate the possible bug and then tries to evolve the program by source code modification until it passes all the test cases (including the failing test case). Schulte et al. [143] extended the approach to mutate the assembly code [143]. The main limitation of such a technique is that it depends on the availability of successful and failing test cases. Moreover, such techniques can fix only very specific bugs [8]. For example, Staber et al.'s approach [151] can only repair expressions and left-hand side of assignment statements in the program [8].

To correct faults, repair can be done in a software after delivery. But such a repair is costly and it is estimated that around 20 billion US dollars could be saved in the USA alone if better testing was done before a software release [8, 156]. Any modification of a software product after deployment or delivery is identified as software maintenance [5]. E.B Swanson et al. [154, 102] identified three categories of software maintenance: corrective, adaptive and perfective. They defined corrective maintenance as the modification of a deployed software product to correct discovered problems; adaptive maintenance as modifications that ensure a software's usability in a changed or changing environment; and perfective maintenance as the modification that improve performance or maintainability of a deployed software [154, 102]. It is estimated that the corrective maintenance is roughly 17% of the total maintenance cost [67].

Researchers have also proposed techniques to repair software faults at runtime without modifying the source code [128, 100, 32]. However, these techniques are not generic and can repair only very specific faults at runtime. For example, Perkins et al.'s technique [128] can fix out-of-bound memory write and illegal control flow transfer attacks on Windows X86 programs. Chang et al.'s technique [32] can handle specific integration issues with third party library invocation provided that the library writers provide the repair module.

1.5 Non-generic Repair of Complex Data Structures

Non-generic data structure repair is not new; traditionally, repair routines have been written manually for the specific data structure at hand [157, 117]. Classic examples include the IBM MVS/XA operating system [117] and the Lucent 5ESS telephone switch [73, 77]. In both the cases, developers used non-generic manual error detection and repair procedures. Other examples include the file system repair tools: fsck [3] and chkdsk [1]. These tools scan the file system at boot time and repair any inconsistencies.

1.6 Generic Repair of Complex Data Structures

Generic repair of complex data structures is a new approach to software robustness [50, 49, 55, 56, 107]. It promises to mutate the state of a running program in such a way that the resulting state satisfies a given assertion or correctness condition. It is generic in the sense that a single repair algorithm can repair many kinds of data structures. It thereby differs from traditional repair, wherein each kind of data structure needs a separate repair algorithm. This makes generic repair potentially very powerful. Indeed, initial generic repair techniques [50, 55] and implementations such as Juzi [56] are very promising.

Based on how the correctness condition is defined, generic data structure repair techniques can be broadly divided into two categories: the first category consists of approaches [50, 48, 176, 177] that use additional specification languages or grammar rules in defining the correctness condition. The second category consists of approaches [56, 55, 93, 82, 81] that do not use any additional specification languages or grammar rules and utilize the same programming languages that the data structures are implemented in to define the correctness condition.

Based on whether the repair approaches use dynamic symbolic execution techniques [29], generic repair approaches can be divided into two categories: dynamic symbolic execution-based and non-dynamic symbolic execution-based repair. The following subsections discuss them in brief.

1.6.1 Specification-based Generic Data Structure Repair

Demsky and Rinard proposed specification based generic data structure repair techniques [48, 50]. Their approach uses specification languages to define rules to transform concrete data structures and their consistency constraints into models (consisting of sets and relations). When the data structure model is checked against the consistency model and violations are detected, they apply repair actions in the data structure model and modify it to conform to the constraints. Later concrete data structure updates are made to reflect the changes done in the model to repair the corruption.

There are other approaches that use additional behavioral specification languages (e.g., Alloy) in defining the repOk method: [176, 139, 166]. The main strength of such a technique is that it tries to leverage the power of the specification language (or grammar rules) and their constraint solver (or repair algorithms). But it comes with the cost that the programmer must learn these specification languages or grammar rules. If the constraints are not written in these rules in the first place, then the she must transform these consistency constraints into grammar rules – which may be hard and error-prone. Moreover, some programmers may not be good at learning specification languages, as they are different from mainstream programming languages such as Java. Since correctness of these rules are crucial, an incomplete or buggy rule set may result in an inefficient or ineffective repair and ultimately lead to the failure of the approach.

We believe that the data structure repair approaches that do not require any specification languages in defining the correctness condition are more intuitive to a programmer since she does not need to learn a new language.

1.6.2 Dynamic Symbolic Execution-based Data Structure Repair

With recent advancement in dynamic symbolic execution [29] techniques, these techniques have been applied to wide range of software engineering fields [131] including automatic test input generation and testing [27, 28, 69, 68, 158, 141], automatic exploit generation [23, 94], program invariant detection [61, 42, 75] and finding implementation deviations [22, 133], etc.

Elkarablieh et al. in Juzi [56] combined dynamic symbolic execution techniques with exhaustive search strategies in repairing the runtime corrupted state of a data structure. Specifically, they used dynamic symbolic execution techniques to repair corruptions in the integer type data fields but for corruptions in the reference type data fields, they applied exhaustive search strategies. Later they [57, 175] applied heuristics on the exhaustive search strategies and improved Juzi's performance.

Our approach [82, 81] uses dynamic symbolic execution techniques to repair corruptions in both the primitive and reference type data fields.

We have analyzed existing repair approaches in our category (i.e., generic dynamic symbolic execution techniques that do not require other specification languages in defining the correctness condition) and found that their efficiency, and in some cases, overall success depend on: (i) the style of the correctness condition and on (ii) the initial state of the runtime corrupted data structure. That is, the correctness condition has to be written in a repair-specific style and for errors in different locations of a data structures, existing approaches may perform unnecessary repairs that change other parts of the data structure resulting in increased execution time and reduced scalability (details in Chapter 3).

1.7 Evaluation of Compilers, Program Analysis and Testing Tools

In this section we use materials from Hussain et al. [83] to introduce benchmarks, their importance and usage in the evaluation of the performance of a software, as well as program analysis and testing tools. The paper [83] proposes a novel approach to generate random benchmark applications with different properties.

A benchmark is a point of reference from which measurements can be made in order to evaluate the performance of hardware or software or both [114, 83]. Evaluation is important to organizations and companies that choose mission-critical software for the business operations [91]. Organizations and companies (e.g., U.S. Department of Defense) use application benchmarks in their evaluation and acquisition process [169]. Moreover, benchmarks are used for evaluating *pRogram Analysis and Testing (RAT)* algorithms and tools [137, 53, 17, 18, 142, 126, 83].

Different benchmarks exist to evaluate different RAT aspects. RAT aspects can be: how scalable RAT tools are, how fast they can achieve high test coverage, how through they are in handling different language features, how well they are in refactoring code, and how efficient they are in profiling, etc [83, 126].

Not all benchmarks can be readily used to evaluate a RAT aspect. For example, consider a situation where different test input generators are evaluated to determine which one achieves higher test coverage faster for a benchmark application [126]. Typically, test input generators use different algorithms to generate input data for each application run, and the cumulative statement coverage is reported for all runs as well as the elapsed time for these runs [126]. A "real-world" application of low complexity is a poor benchmark in this case as most test input generation approaches will perform very well by achieving close to 100% coverage in few runs. On the other hand, a real-world high complexity benchmark may have dependencies on components written in different languages and run on different platforms, making it hard to adjust to use in the experiments. Moreover, implementation of test generation approaches may have limitations and not support all the language features of such a complex program. Ideally, a large number of different benchmark applications are required with different levels of code complexity to appropriately evaluate test input data generation tools [126].

To overcome the problems of such real-world programs, one may think to write benchmark applications from scratch. But writing benchmark application from scratch requires a lot of manual effort, it can also introduce significant bias and human errors [88]. Moreover, more than one benchmark is often required to determine the sensitivity of the RAT approaches based on the variability of results for applications that have different properties [83].

Ideally, users should be able to easily generate benchmark applications with desired properties. This idea has already been used successfully in testing relational database engines, where complex *Structured Query Language (SQL)* statements are generated using a random SQL statement generator [149]. In [83] researchers have proposed an approach *Random Utility Generator for pRogram Analysis and Testing (RUGRAT)* for generating application benchmarks within specified constraints and within the range of predefined properties. RUGRAT is implemented for Java and it is open source software and available for download from the RUGRAT tool web site[1]. We used RUGRAT generate several Java application and used them to evaluate Java compilers and different RAT tools.

1.8 Contributions

Given a corrupted data structure and a correctness condition, which is popularly known as the *repOk* method to check the consistency, we propose that our *Dynamic Symbolic Data Structure Repair Algorithm* can overcome some of the limitations of the existing approaches and improve in repair performance in the following ways:

- When existing approaches do not scale, are slow or inefficient (e.g., due to exhaustive search strategies), it can take less time and fewer repair attempts in repairing the data structure.
- Less dependent on the style of the correctness condition and location of an error in the data structure.

We realize that our approach cannot be applied for all the data structures and in special cases it may perform less efficiently than the existing current state of the art approaches. For example, consider the case when a well written correctness condition finds a corruption that has a trivial solution (e.g., a *null* to break a self loop in the binary tree) and the existing approach uses the trivial solution as its first attempt to repair the structure in its exhaustive search. In this case the existing approach will

[1]RUGRAT: `https://sites.google.com/site/rugratproject/`

repair the corruption very quickly with less overhead (e.g., without building the path condition, invoking the constraint solver etc.) to find the solution. Details on the limitations of our approach, when and why it may fail or perform less effectively will be discussed under the context of threats to validity.

To explore the potential of automatic program generation for program analysis tool evaluation, we picked the recent automatic program generator tool RUGRAT [83] to generate dozens of Java applications, ranging from 300 LOC to 5 MLOC, to benchmark several versions of a popular Java source to bytecode compiler as well as popular program analysis and testing tools. That is, we conducted several experiments to explore the usefulness of RUGRAT for the evaluation and benchmarking of Java RAT tools.

Chapter 2

Background and Terminology

In this chapter we discuss the necessary background on program assertions and correctness, categories of data structure constraints, dynamic symbolic execution engine, one of the state-of-the-art data structure repair tool called Juzi, and constraint solvers – which we will need for a motivating example. We also discuss in brief which programming paradigm our approach applies to and how we evaluate it.

2.1 Assertions and Correctness

Software customers and developers likely have an informal notion of the conditions under which the state and behavior of their programs are correct. Such informal notions are the notions of program correctness that typically matter most in real-world software applications. Large parts of software engineering are therefore concerned with capturing informal correctness notions and transforming them into more formal ones, culminating in the fully formal notion of source code. This formal notion of correctness influences the terminology used in this dissertation. By *correct* we mean correct in the informal sense of the user. A *correctness condition* tries to capture this informal correctness in a more formal notion.

An easy way to write down a correctness condition is to add to the program text a simple if-condition or program assertion, commonly known as *repOk* [103]. Empirical evidence suggests that programmers write assertions into their code and the code that contains more assertions tends to contain fewer bugs [97].

Correctness conditions or the repOk methods check partial program correctness. Programs typically do not have a repOk method that returns true if and only if the entire program is in a correct state. Instead, a repOk method checks if some (small) part of the program is in a correct state. Moreover, the sum of all repOk methods in a program typically does not amount to a check of the full program state correctness.

Correctness conditions may be expressed in different styles and languages, ranging from formal modeling languages to program assertions. For this work, we concentrate on program assertions. Assertions are attractive as programmers do not have to learn a separate language in order to write down correctness conditions.

The main assumption we use is that correctness conditions have been engineered to be correct. This assumption is also used in all the previous works that we are aware of. However, we do not require the related assumption that is often made in this area, namely that correctness conditions also satisfy style rules that are specific to a certain repair technique. Our goal is therefore to provide an approach that does not depend on such repair-specific style rules.

2.2 Data Structure Constraints

Based on the related work on data structure repair of which we are aware, data structure constraints can be classified into two categories: (1) structural constraints and (2) data constraints. Data structures can be complex and these two categories can overlap to create constraints that have a combination of both the structural and data constraints. Structural constraints specify the structural correctness conditions that involve pointers and references of a data structure, whereas the data constraints specify correctness conditions over the non-pointer data that a data structure holds.

For example, the standard singly linked-list data structure only has structural constraints that specify that there cannot be any cycle or loop through the node's *next* pointer. The binary search tree data structure has both structural and data constraints (i.e., it has to maintain the tree structure and any node's left (right) subtree should contain nodes with keys less (greater) than the node's key, etc.) but the binary tree has only structural constraints. Similarly a sorted array only has data constraints over its elements' values.

2.3 Dynamic Symbolic Execution

Dynamic analysis is the ability to monitor code as it executes [144]. On the other hand, symbolic execution is a program analysis technique that executes programs with symbolic rather than concrete inputs [29]. As Cadar et al. describes [29], symbolic execution maintains a symbolic state of the program by mapping program variables to symbolic variables and a path condition of symbolic expressions that accumulates constrains on inputs that trigger the execution to follow the associated path. At every conditional statement, the path condition is updated with conditions on the input to choose between alternative paths (i.e., the *then* and *else* branch), resulting in two path conditions for the two execution paths. As a result, the program output is represented as a function of symbolic inputs. The the satisfiability of these path conditions are checked with a constraint solver. The main limitation of the symbolic analysis is that there are large numbers of possible paths. Moreover, one constraint solving attempt for a single path condition can take a long time (even infinite time) and many path conditions are simply infeasible and cannot be solved. For these limitations symbolic analysis suffers from scalability issues. Combining the two, dynamic symbolic execution or DSE tries to leverage benefits of both the approaches and tackle the scalability issues. DSE marks the program input as symbolic and then monitors concrete execution while performs symbolic execution of the program along the concrete path [131].

During a concrete execution a program can hit different branch conditions. Decisions of many of these branch conditions depend on the input values. In DSE, the path condition is formed by conjunction of all such branch conditions (involving some symbolic inputs) seen in a concrete execution. Any actual input that satisfies this path condition will steer the program execution along the same path as the original execution. A new path condition can be found by negating one of the branch conditions and removing the subsequent ones. A constraint solver can then check the satisfiability of the new path condition. If satisfiable, the solution from the constraint solver can provide values for the inputs (represented as the symbolic variables in the path condition) that will execute the program up until the modified branch and then take the other path. Thus DSE engines can analyze a program's execution and determine what input would cause which part of the program to execute.

Researchers have utilized this property in many ways: detecting program invariants [61, 42, 75], detecting software security issues and implementation anomalies [150, 23, 94, 22, 133], input generation and software testing [27, 28, 69, 68, 158, 141], etc.

There are two ways of performing DSE: through code instrumentation or trace-based analysis [131, 144]. In the code instrumentation approach, while the DSE engine executes the program concretely, it simultaneously runs the instrumented code using a symbolic interpreter [68, 158, 145]. This interpreter keeps track of the symbolic path condition along with all the symbolic expressions of all registers and memory locations. In the trace-based approach, DSE is performed in two steps [131]. First, for each value the program computes, it records all the input values that influence the computed value [65]. Then it finds the presence of these inputs in the execution trace and marks them as symbolic. In the second step it analyzes and extracts the path condition from the trace involving symbolic inputs [150].

2.4 Dynamic Symbolic Execution Engine: Dsc

We used Dsc [84, 126] [1] which is a dynamic symbolic execution engine for Java bytecode that uses code instrumentation approach. It uses the instrumentation facilities provided by the JVM of Java 5 to instrument the user program at load-time [34], using the open source bytecode instrumentation framework ASM [24]. Dsc instruments the bytecode of the target application automatically by inserting method calls (i.e., callbacks) after each instruction in the code. During the application's execution, the callbacks enable Dsc to maintain the symbolic state by mirroring the effects of each user program instruction. By manipulating programs at the bytecode level, Dsc extends its analysis from the user code into all libraries called by these programs. In addition, Dsc allows users to selectively exclude classes from instrumentation.

We used the the data structure consistency method, (i.e., the *repOk* method) as the target of Dsc. Since Dsc, unlike Juzi, can reason about the primitive and reference types and corresponding field accesses, we were able to apply the full DSE for our data structure repair approach.

There are other dynamic symbolic execution engines, e.g., Dart, jCute, and Pex [68, 145, 158]. These tools are used in automatic random testing [68] or automatic unit test case generation [145, 158]. We chose Dsc over these tools as they did not have support for Java programs (e.g., Dart and Pex) and because not all tools had source code available. Moreover, we had the expertise of the Dsc author [2] available to help us implement and integrate the data structure repair module on top of Dsc.

[1]Dsc: `http://ranger.uta.edu/~csallner/dsc/`

[2]Christoph Csallner: `http://ranger.uta.edu/~csallner/`

2.5 Most Closely Related Prior Work: Juzi

Juzi is one of the state of the art approaches to generic repair using assertions [55, 56]. It builds on the Korat framework [21] and monitors the execution of the assertion to determine the order in which the assertion accesses data structure fields. When the assertion returns false, Juzi mutates the value of the field that was accessed just before the assertion failed. If this repair attempt does not result in a satisfactory state, Juzi backtracks in the list of field accesses and continues with the field that the assertion had accessed earlier. Each repair attempt mutates the original state in one field.

Depending on the field type, Juzi uses different techniques to determine candidate values to be used in a mutation. For corruptions in integer fields, it applies dynamic symbolic execution to build the path condition and then employs a mix of integer constraint solvers [57, 56], Dicos [59] and CVCLite [12]. If the path condition is satisfiable it assigns the solution from the constraint solver to the symbolic variables. For reference fields, Juzi uses a systematic search algorithm along side the Korat technique [21] of skipping isomorphic structures in an exhaustive trial and error approach. Therefore, Juzi cannot be considered as a full dynamic symbolic execution technique as it does not apply the technique in repairing all data type corruptions.

2.6 Constraint Solver

Constraint solver is a basic component to most dynamic symbolic execution approaches as they heavily rely on it for checking the feasibility of a modified path condition and for solutions for the involved symbolic inputs in the expression [60]. Recently, researchers have renewed their focus on dynamic symbolic execution approaches and applications due to unprecedented improvements in constraint solver capabilities [13, 15] and growth in the computational power of average computers [60].

Our dynamic symbolic execution engine, Dsc, uses a powerful automated constraint solver from Microsoft Research, called *Z3* [116]. Dsc can handle both the reference and primitive type data fields and invokes Z3 for solving data structure constraints involving both the data types. Juzi, on the other hand, uses a mix of constraint solvers, Dicos [59] and CVCLite [12] for solving corruptions in the primitive integer type data fields, but for reference type fields, it applies a systematic search algorithm along with Korat technique [21] to reduce the solution search space [57, 56].

To promote improvements and compare the performance of different existing constraint solvers over benchmarks, starting from 2005, organizers have regularly arranged competitions [14, 15]. Since Z3's first appearance in 2007, with some minor exceptions, it has always shown better performance over CVC and other constraint solvers in all the benchmark categories [13]. In our research we did not compare the performance differences between Z3 and CVC in solving integer type constraints for the data structure repair and intend to pursue it as a future study. Nevertheless, we believe Z3 will outperform CVC as it has done so for almost all the benchmark categories in the competitions [13].

2.7 Application Domain and Evaluation of Our Approach

We describe our implementation in terms of object-oriented software and especially Java programs, but the algorithm equally applies to related languages (C++, C#, etc.) and related programming paradigms (i.e., procedural languages).

We evaluate our approach by applying it to a few text book data structures with induced errors. We utilize the consistency checker (or *repOk*) methods of these data structures to detect the errors. Both the data structures and the repOk methods are written in Java. We compare the performance of one of the state-of-the-art tools Juzi and the prototype implementation of our approach. We also discuss our experience in solving corruptions in a simplified example of a file allocation table data structure.

Chapter 3

The Problem

In this chapter we discuss the limitations of existing Java-based generic data structure repair techniques. These approaches include: [93, 57, 55, 56]. As a representative of these approaches, we picked Juzi [56] since it works for data structures and repOk methods written in Java and was available for experimentations. Other approaches are close variants of Juzi but were not available for experimentation. For example, Starc [57] extends Juzi by using static analysis to find the fields used in the repOk method to traverse the data structure. For corruptions in such a reference field, it prunes the search space by looking for *forward* or not yet traversed pointers as solutions.

In the following sections we discuss the limitations of Juzi. Juzi has style dependency on the repOk method. The repOk method must be written in such a way that all the fields in the data structure are directly accessed and it should return immediately after a corruption is detected. Moreover, Juzi performance varies for errors at different locations in a data structure and sometimes in repairing a corruption, it changes other data values instead of the original corrupted one resulting in data loss.

3.1 Overview of Limitations of Juzi

We have found that the performance of Juzi depends on the style of the repOk method. For example, for the two functionally same repOk methods, depending on the style of when it returns after detecting an error (either immediately or late), Juzi can take different execution time and number of repair attempts. For example, in some of our experiments we had to stop Juzi as it was taking too long (e.g., as in Figure 4.8, more than few hours for a random binary tree of size 12, as compared to a different repOk method, for which for the same binary tree it took only less than 100 milliseconds) to repair a corrupted data structure. This varying performance due to style dependency may cause problem where there is an upper bound on allowed repair execution time and number of repair attempts.

Moreover, Juzi requires that a data structure field must be directly accessed in the repOk method for it to be considered as a corrupted field. For example, if a local variable is used and a field is accessed indirectly through the local variable, Juzi cannot detect the field's involvement in a corruption. In such cases Juzi might miss potential corrupted fields and try to repair other fields resulting in futile attempts.

Authors of Juzi acknowledged that the performance of Juzi varies with different error locations in the data structure [54]. That is, for the same error in different locations of a data structure, it can take different execution time and number of attempts in repairing the structure. This varying performance can be an issue where we have an upper bound on execution time and number of allowed repair attempts in repairing a corrupted data structure.

We have also found that in repairing errors in different locations of a data structure, Juzi sometimes changes other data values instead of the original corrupted one resulting in data loss.

In practice the above problems may appear together (e.g., a late-returning repOk method may only use indirect field accesses). But we analyze the problems individually. In the following sections we describe these limitations with examples.

3.2 Correctness Condition Style Dependency

In this section we discuss Juzi's correctness condition (i.e., *repOk* method) style dependency. This is a problem since it affects Juzi's performance in repairing a corrupted data structure efficiently.

To illustrate the problem, we consider the binary tree data structure given in Figure 3.1. We took the correctness condition from [56]. The constraints for the binary trees are: (1) acyclic along the left and right pointers and (2) the number of nodes reachable from the root node along the left and right fields is stored in the *size* field. To emphasize the fact that Juzi's repair actions heavily depend on the writing style of the correctness condition, we slightly modified the correctness condition of Figure 3.1 creating another version in Figure 3.2. This modified version of repOk produces the same result as the original. It differs only in the style that whenever it discovers a corruption, it stores the result in a temporary boolean variable named *result* and returns the desired answer at the end of the method.

Figure 3.3 (a) shows an example binary tree that consists of five nodes. The first node has a corrupt value in its left field, namely it points to the root node creating a cycle. Note that, in the modified repOk of Figure 3.2, the *size* field of the binary tree is always the last accessed field. To repair the corruption, as shown in Figure 3.3 (b), Juzi first tries to mutate this last accessed field of the repOk method. After each repair attempt Juzi executes the repOk method to check if the resulting data structure satisfies the repOk method. Since *size* is an integer field, Juzi performs dynamic symbolic execution and tries to solve for the field with a constraint solver. But because *size* is not the corrupted field, mutation of this field does not repair the data structure. It then tries to mutate the second-to-last accessed field, *right* of node 5. Since this is a reference type field, Juzi applies its systematic search algorithm to find a solution. For each possible value for this field, it also tries to mutate the

```java
public class Node {
  Node left;
  Node right;
  // ..
}

public class BinaryTree {
  Node root;
  int size;
  // ..
  public boolean repOk() {

    // An empty tree must have zero in size
    if (root == null)
      return (size == 0);

    Set<Node> visited = new HashSet<Node>();
    visited.add(root);
    LinkedList<Node> workList = new LinkedList<Node>();
    workList.add(root);

    while (!workList.isEmpty()) {

      Node current = workList.removeFirst();

      // The tree must have no cycles along left
      if (current.left != null) {
        if (!visited.add(current.left))
          return false;
        else
          workList.add(current.left);
      }

      // The tree must have no cycles along right
      if (current.right != null) {
        if (!visited.add(current.right))
          return false;
        else
          workList.add(current.right);
      }
    }

    // Size must be equal to #visited nodes
    return (visited.size() == size);
  }
}
```

Figure 3.1: Example binary tree data structure, abbreviated, consisting of a Node class and a BinaryTree class. Method repOk is a correctness condition for the binary tree, which may be invoked by assertions throughout the program.

```
public boolean repOk() {

  // An empty tree must have zero in size
  if (root == null)
    return (size == 0);

  boolean result = true;

  Set<Node> visited = new HashSet<Node>();
  visited.add(root);
  LinkedList<Node> workList = new LinkedList<Node>();
  workList.add(root);

  while (!workList.isEmpty()) {

    Node current = workList.removeFirst();

    // The tree must have no cycles along left
    if (current.left != null) {
      if (!visited.add(current.left))
        result = false;
      else
        workList.add(current.left);
    }

    // The tree must have no cycles along right
    if (current.right != null) {
      if (!visited.add(current.right))
        result = false;
      else
        workList.add(current.right);
    }
  }

  // Size must be equal to #visited nodes
  if (visited.size() != size)
    result = false;

  return result;
}
```

Figure 3.2: Modified correctness condition. It produces the same output as the corresponding correctness condition of Figure 3.1. However, it stores the status of the check temporarily in a local variable and returns it at the end.

previously attempted *size* field. Failing to repair the data structure, subsequently (as shown in c - h), Juzi backtracks in the list of fields accessed in the repOk method and continues its repair actions in an exhaustive fashion trying all possible mutations for each reference fields and invoking the constraint solver for the integer type data field (along with mutating the previously attempted fields).

Using Juzi's [56] repair visualization option, we monitored Juzi's repair attempts for the binary tree example in Figure 3.3 (a). Table 3.1 shows first 17 attempts of Juzi's approach.

Finally, Juzi reaches a field that the repOk method had accessed very early, the corrupt *left* field of the first node, and now Juzi repairs the binary tree successfully in Figure 3.3 (z).

An exhaustive approach such as the one of Juzi may work for repairing small data structure instances, containing few nodes. But when repairing larger structures, at some point exhaustive search becomes inefficient. The number of possible mutations grows exponentially and most mutations do not result in a correct state.

Our key insight is that we can guide the repair by mutating the data structure in such a way that the repaired data structure takes a pre-determined execution path. In our example, we want to invert the outcome of the if-condition just after which the temporary variable *result* got assigned such that, instead of returning false, repOk returns true.

Chapter 4 discusses how we implement this insight and presents evaluations that show the promise of our proposed repair approach.

3.3 Error Location Dependency

We have found that depending on error locations in a data structure, Juzi's performance can significantly vary. In repairing errors in different locations, Juzi

Table 3.1: Juzi's repair attempts to repair the binary tree example in Figure 3.3 (a). The binary tree object is represented by *bt*. Node objects are represented by numbers that correspond to the numbers in Figure 3.3. For each repair attempt the table shows: attempt count, target repair object and field, attempted repair and result of the *repOk* invocation after the repair. Juzi starts with the *last* accessed field in the *repOk* method, *size*. Since mutation of this field (Juzi shows it as X) does not repair the structure, Juzi backtracks to the second-to-last accessed field, i.e., *right* of node 5 (attempt 2). For each possible value for this field, it also tries to mutate the previously attempted *size* field. After trying all possible values in an exhaustive fashion for the *right* field, it then tries to repair *left* of node 5 in combination with *size* and *right* of node 5 (attempts 12, 13, ..., 24, ...). Finally, Juzi backtracks to the actual corrupted field, *left* of node 1 and repairs the data structure in 1,345,333-th attempt.

Atmpt. count	Target object	Target field	Atmpt. repair	*repOk* returns
1 (Figure 3.3 (b))	bt	size	X	false
2	5	right	1	false
3	bt	size	X	false
4	5	right	2	false
5	bt	size	X	false
6 (Figure 3.3 (g))	5	right	3	false
7	bt	size	X	false
8	5	right	4	false
9	bt	size	X	false
10	5	right	5	false
11	bt	size	X	false
12	5	left	1	false
13	bt	size	X	false
14	5	right	1	false
15	bt	size	X	false
16	5	right	2	false
17	bt	size	X	false
...	...	...	...	...
...	...	...	...	...
24	5	left	2	false
25	bt	size	X	false
26	5	right	1	false
27	bt	size	X	false
28	5	right	2	false
...	...	...	...	...
...	...	...	...	...
1,345,333 (Figure 3.3 (z))	1	left	null	true

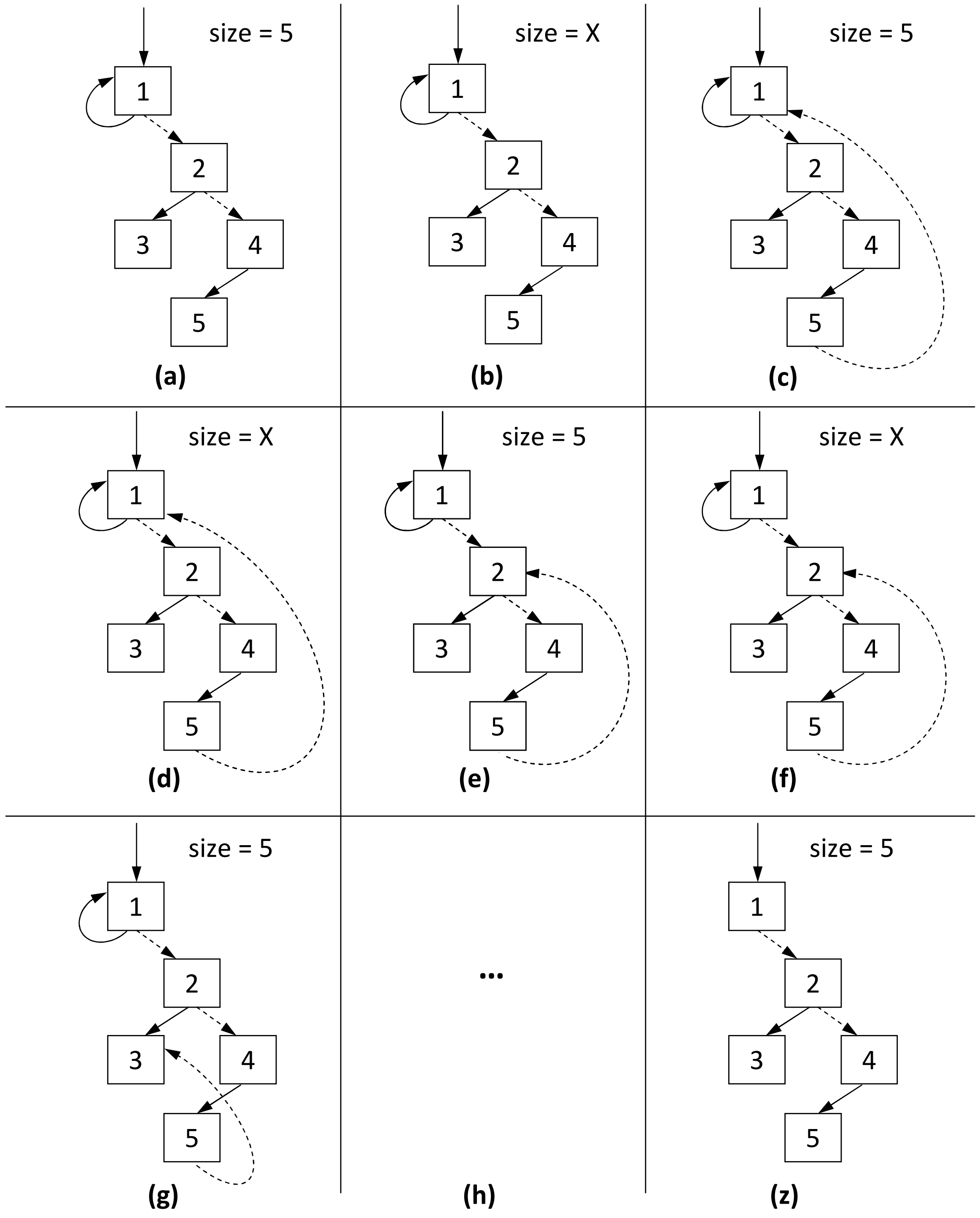


Figure 3.3: Exhaustive approach in Juzi. Initially (a), the binary tree is corrupt. Dotted lines and X in *size* field show repair attempts (b - g). Omitted are several subsequent repair attempts (h). Ultimately repair culminates in the binary tree (z). Note that there are no fresh objects in Juzi.

sometimes change other data values instead of the original corrupted one. These result in unnecessary repair attempts, increased execution time and data loss.

To illustrate the problem of error location dependency, we consider the binary search tree data structure and its correctness condition (Figure 3.4 and Figure 3.5) that we have also taken from the authors of Juzi [56] and added conditions to control the return.

The constraints for a traditional binary search tree are: (1) no-cycle: acyclic along the left and the right pointers, (2) The left (right) subtree of a node contains only nodes with keys less (greater) than the node's key; the subtrees must be binary search trees. The repOk method in Figure 3.5 has another constraint, (3) the number of nodes reachable from the root node along the left and the right fields is stored in the *size* field. The purpose of the third constraint probably is to prevent trivial repairs (e.g., making the root *null*) and ensure that the initial corrupted binary search tree and the repaired one are equal in size.

We introduce corruptions that break the second constraint. Specifically, we induce corruptions that change the data values of different nodes without introducing any cycle or changing the total number of nodes in the binary search tree. In Figure 3.4 the *repOk* method checks the tree property of the binary search tree. In the continued code in Figure 3.5, *repOk* method checks the second constraint. Note that, instead of using recursion, which is commonly used in checking the constraint, authors of Juzi in the repOk method in Figure 3.5 use a stack and iteratively check the property in a bottom-up fashion.

Figure 3.6(a) shows an example of a corrupted binary search tree of nine nodes: the left child of the root has a key which is greater than the root's key. For simplicity, we will refer to a node by its key from now on. Since the repOk method in Figure 3.5 uses a stack to check the invariant iteratively in a bottom-up fashion, the first detected error (at line 69) involves nodes 176 and 79. The error specifically is that the node 79, being in the right sub-tree of node 176, should have a greater value than 176 as its key, or the other way around.

When Juzi detects an error, it returns *false* immediately from the repOk method to initiate the repair process. To repair the corruption, Juzi tries to mutate the field that the repOk method accessed last – the *key* of node 79. Note that, Juzi fails to detect indirect field accesses and ignores 176 (it is accessed through *min* at line 69). As shown in Figure 3.6(b), Juzi replaces it with a symbolic variable, e.g., X and tries to solve for it. The first attempt solves X for 177, which further corrupts the data structure, resulting in a different last accessed field in the second attempt. Subsequently (omitted from the figure) Juzi backtracks in the list of fields that the repOk method accessed and continues its repair attempts in an exhaustive fashion. Finally, Juzi repairs the data structure in (z). Note that the corruption is in a location which affects other parts of the data structure, and since Juzi tries to mutate the last accessed field in a greedy fashion, its repair validates the original corrupted node by changing almost all the other nodes' key values (except three, including the original corrupted node).

In situations like this, our key insight is that we can guide the data structure repair by considering all fields involved in any corruption rather than just the last accessed field as the target; and when a corruption is detected, instead of returning from the repOk method immediately and attempt a repair in a greedy fashion, we can let it run and gather more constraints in solving for the corrupted field. Moreover, running the repOk until it returns at the end has its added benefit that we can count how many times any field appears in a corruption and take the most occurring (i.e., the most influential corrupted) field first as the repair target.

Chapter 5 discusses the updated algorithm that incorporates this insight and presents experiment data to show the promise of the approach.

```
public class Node {
  Node left;
  Node right;
  int key;
}

public class BinarySearchTree {
  Node root;
  int size;
  boolean return_immediate = false;
  // ..
  public boolean repOk() {
    // An empty tree must have zero in size
    if (root == null)
      return (size == 0);

    boolean result = true;
    Set<Node> visited = new HashSet<Node>();
    visited.add(root);
    LinkedList<Bst_Node> workList = new LinkedList<Bst_Node>();
    workList.add(root);

    // Checks for cylcles along left or right
    while (!workList.isEmpty()) {
      Node current = workList.removeFirst();
      if (current.left != null) {
       // The tree must have no cycles along left
       if (!visited.add(current.left)) {
         if (return_immediate)
           return false;
         result = false;
       }
       else
         workList.add(current.left);
      }

      if (current.right != null) {
       // The tree must have no cycles along right
       if (!visited.add(current.right)) {
         if (return_immediate)
           return false;
         result = false;
       }
       else
         workList.add(current.right);
      }
    }
```

Figure 3.4: Example binary search tree data structure [56], consisting of a Node class and a BinarySearchTree class.

```
    // Size must be equal to #visited nodes
    if (visited.size() != size){
      if (return_immediate)
         return false;
      result = false;

    }

    // Check for BST property
    int min = -1;
    Stack<Node> stack = new Stack<Node>();
    Node p = root;

    while (p != null) {
      stack.push(p);
      p = p.left;
    }

    while (stack.size() > 0) {
      Node q = stack.pop();
      if (q.key < min) {
        if (return_immediate)
          return false;
        result = false;
      }

      min = q.key;
      Node r = q.right;

      while (r != null) {
        stack.push(r);
        r = r.left;
      }
    }

    if (return_immediate)
      return true;

    return result;
  }
}
```

Figure 3.5: Example binary search tree data structure continued from Figure 3.4.

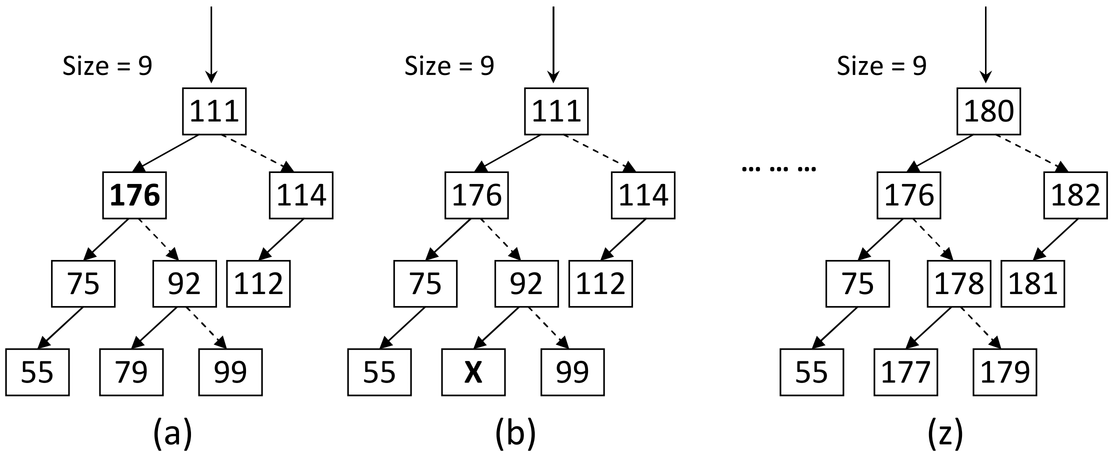


Figure 3.6: Initially (a), the binary search tree is corrupt in root's left child (176). Solid arrows represent left child to a node, dotted as right child. In (b), X in the *key* field represents symbolic variable in repair attempts. Omitted are several subsequent repair attempts in Juzi's exhaustive approach. Ultimately repair culminates in the correct binary search tree (z) which is different in all but three nodes from the initial structure.

Chapter 4

Solution: Dynamic Symbolic Data Structure Repair

Our approach has two parts: at the lower level is a dynamic symbolic execution engine that has a broad interface to allow modification of path conditions, etc. At the top layer, using the dynamic symbolic engine, sits our generic repair algorithm. In Section 2.4 we briefly discussed Dsc, the dynamic symbolic execution engine that we used in our approach. In this section we briefly describe the algorithm, implementation and present the evaluation of our approach. The material presented in this section is essentially from our two short papers on automatic data structure repair [82, 81]

4.1 Repair Algorithm

Figure 4.1 gives a high-level overview of our algorithm, Dynamic Symbolic Data Structure Repair or DSDSR. As part of its normal execution, a program invokes assertions or other methods that implement a correctness condition. In our description we follow previous work and name such a method repOk [56, 55]. When the correctness condition fails, i.e., repOk returns false, execution is handed over to our extended dynamic symbolic engine, which in turn invokes the instrumented version of repOk (I-repOk). Executing the instrumented repOk builds the path condition of the execution path that leads to the point at which repOk failed.

With the full symbolic path condition in hand, we can now modify the path condition to obtain a different path. I.e., if we invert the last if-condition (or, when we allow repOk to run and return late, the if-condition after which the error was

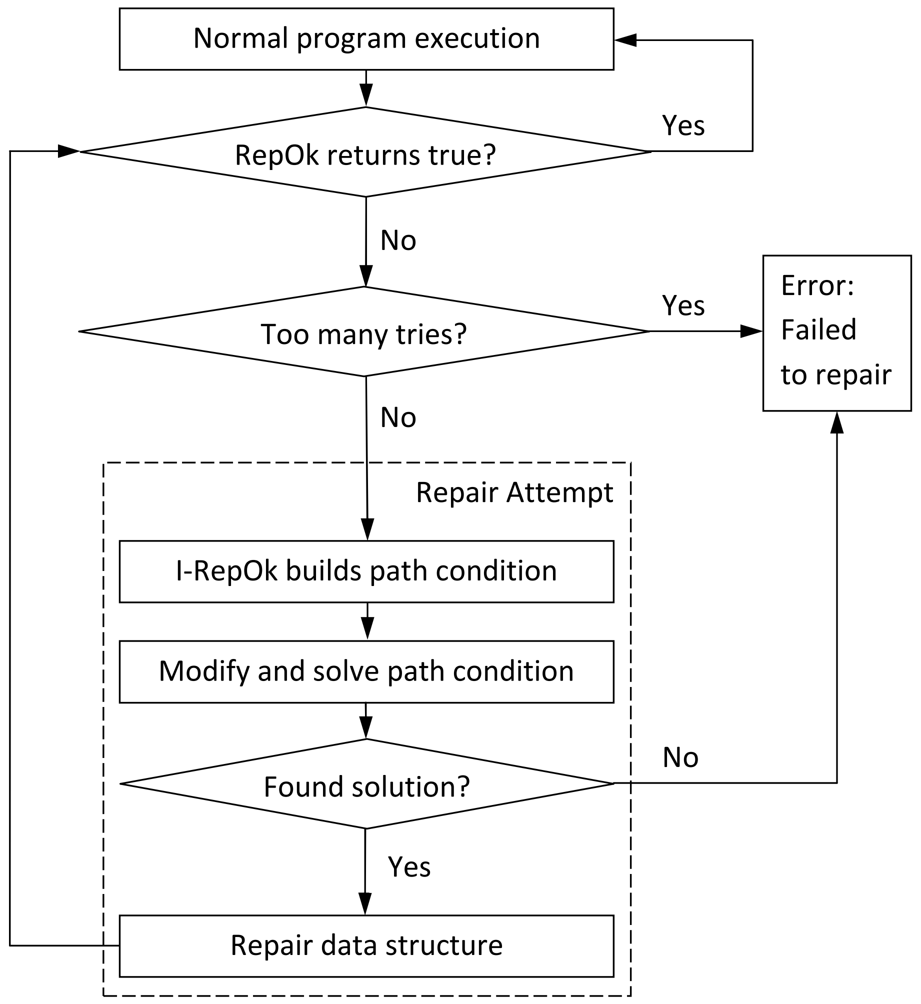


Figure 4.1: Overview of our dynamic symbolic data structure repair algorithm (DS-DSR). RepOk is a method that implements a given correctness condition. I-RepOk is the instrumented version of repOk.

detected) we obtain a path that does not return false at the point at which the original execution failed. At the same time, solving such a new path condition can give us an input state that will trigger the new path. If the new state satisfies the repOk correctness condition, we can mutate the existing state to resemble the new one, which completes the repair.

The algorithm relies on a faithful encoding of the path condition and other program constraints in a format suitable for automated reasoning. It further relies on a powerful automated constraint solver that can simplify such constraints and, if a solution exists, can produce a concrete solution. Finally, the solution of the constraint solver needs to be mapped back into the program state, to repair the existing data structures.

We repair the data structure according to the solution of the constraint solver and invoke repOk to check if the resulting structure satisfies the repOk correctness condition. If repOk again returns false, indicating there are more errors, we may make another iteration and attempt another repair.

To prevent an infinite loop of repair attempts, the algorithm terminates after reaching a user-defined number of futile attempts. If the repOk method returns true, we consider the repair attempt to be successful and resume normal program execution.

The main advantage of our approach is that, unlike Juzi, in the search for a data structure that satisfies a repOk correctness condition, we do not need to exhaustively generate many possible candidate data structures. Instead, DSDSR derives conditions directly from the repOk implementation to generate a single data structure that satisfies the correctness condition.

4.2 Design

We explain the design of our algorithms with the help of our motivating example the binary tree data structure and its correctness condition given in Figures 4.2 and 3.2, respectively. Recall that when the correctness condition fails, i.e., repOk returns false, execution is handed over to the extended dynamic symbolic engine, which in turn invokes the instrumented version of the repOk method. It calls the instrumented version of the repOk method twice, as shown in Algorithm 1, and the purpose of these two executions are twofold:

- Explore the state space to collect meta information of the data structure and identify the corrupt instance and field.
- Build an appropriate path condition, by negating the constraint that represents the corruption in the data structure.

```
$fieldAccessList \leftarrow emptyList$;
$instanceAccessList \leftarrow emptyList$;
$constraintsList \leftarrow emptyList$;
$lastFieldAccessed \leftarrow -1$;
$lastInstanceAccessed \leftarrow -1$;
$corruptConstraint \leftarrow -1$;
$maps \leftarrow createMaps(root)$;
$repOkExec1(maps)$;
$updateMap(maps)$;
$repOkExec2()$;
$negate\ constraintsList[corruptConstraint]\ and\ update\ the\ path\ condition$;
$solvethemodifiedpathcondition$;
$updatethedatastructurewiththesolution$;
```

Algorithm 1: Main Algorithm

Table 4.1 summarizes the purpose of the different items used in the Algorithm 1. The algorithm maintains three lists: lists for the instance and field accesses, and the observed constraints from the regular DSE during the repOk method execution. It also keeps three indices to the corresponding lists that points to the instance, field

Table 4.1: Items in the Algorithm 1 and their purposes

Item	Purpose
fieldAccessList	Field accesses in repOk
instanceAccessList	Instance accesses in the repOk
constraintsList	Constraints, both meta and branch constraints, observed by the repOk
lastFieldAccessed	An integer index to the last accessed field before the corruption
lastInstanceAccessed	An integer index to the last accessed instance before the corruption
corruptConstraint	An integer index to the last observed constraint before the corruption
root	Root or access point to the data structure
createMaps	Traverses the data structure to create map for each field type
repOkExec1	Finds target: the corrupted instance and the field; updates the indices.
updateMap	Replaces value of the corrupted field with symbol.
repOkExec2	Builds the path condition from *constraintsList* and *corruptConstraint*

and constraint that is accessed or observed just before a corruption is detected. After initialization of these lists and indices, the algorithm at first calls the *createMaps* method, which using Java's reflection mechanism (as shown in Algorithm 2), traverses the data structure to build maps for each field type.

Then it calls the instrumented repOk method, *repOkExec1* the first time which primarily finds the target corrupted instance and the field. Furthermore, the method gathers meta data regarding the state space. These meta information include the subtype and supertype relations, the dynamic type of objects, types of referenced objects, visibility and others that Dsc collects as part of normal DSE. These relationships are asserted and later utilized in solving for the modified path condition.

```
$classSet \leftarrow emptySet$;
$objectSet \leftarrow emptySet$;
$map \leftarrow emptyMap$;
$objectSet.add(root)$;
$worklist \leftarrow emptyQueue$;
$worklist.enqueue(root)$;
**while** *worklist not empty* **do**
    $obj \leftarrow$ *worklist.dequeue()*;
    $classSet.add(class(obj))$;
    **foreach** *field f in* $refFields(obj)$ **do**
        $refObj \leftarrow$ get referenced object in $f$ of $obj$;
        update map for $f$ at key $obj$ with $refObj$;
        **if** *objectSet.add(refObj)* **then**
            *worklist.enqueue(refObj)*;
**foreach** *class in classSet* **do**
    **foreach** $field$ *in* $refFields(class)$ **do**
        assert map for $field$;
**return** map
```

Algorithm 2: createMaps(Object root): Traverses the data structure to create map for each field type

Algorithm 3 describes *repOkExec1* – the first call of the instrumented repOk method. It basically monitors different instance and field accesses during the repOk method execution. In cases when there is a direct instance and field access, it simply records the event by appending to the corresponding lists. But there could be indirect accesses through local variables. Such indirect accesses usually occur first by assigning a field to a local variable and then using the local variable at the later part of the code.

Ideally, there should be no side effects of such assignment statements in the correctness conditions. So, as a heuristic, for any such assignment statement at a program point P, the algorithm wraps the value with two indices: the last accessed field and the last accessed instance prior to program point P. As a result, whenever the value is used later in the code, the corresponding instance and the field that it represents are easily traced back.

For example, the modified repOk method for the binary tree in Figure 3.2 at line 21, assigns *false* to the local boolean variable when it detects an error. At the end of the execution, when the repOk method finally returns false, the algorithm deduces the target corrupted instance and field by utilizing the instance and field access lists and the indices. It finds the values for the indices by either unwrapping the operand stack top return value (when it is wrapped) or simply by the maximum index of the lists. In our binary tree example in Figure 3.2, when the repOk method returns false using temporary variable *result* at line 39, it unwraps the *result* to get the indices and deduces that the first node (node 1) is the corrupt instance and its *left* field is the corrupted field.

Once the target corrupted instance and field is detected, the *updateMap* method as described in Algorithm 4, replaces the concrete value (in the field) with a symbolic variable. Specifically, it updates the map for the corrupted field by replacing the value with a symbolic variable at the corrupted instance key location. This update helps the second instrumented repOk method, *repOkExec2* to build the path conditions with symbolic variable.

Algorithm 5 describes *repOkExec2*. Its responsibility is to build the path condition. Whenever the method execution hits a condition and takes a branch, it creates

```
while more bytecode statements do
    match statement do
        case field read:
            resolve accessed instance and field;
            add to $fieldAccesses$;
            add to $instanceAccesses$;
        case local variable read:
            if field access through local variable then
                resolve accessed instance and field;
                add to $fieldAccesses$;
                add to $instanceAccesses$;
        case (x ← e):
            wrap e with pointers to maximum index of $fieldAccesses$,
            $instanceAccesses$;
        case return e:
            if operand stack top e is wrapped then
                unwrap the e and get pointers: $fieldAccessedPointer$
                $instanceAccessedPointer$;
                $lastFieldAccessed \leftarrow fieldAccessedPointer$;
                $lastInstanceAccessed \leftarrow instanceAccessedPointer$;
            else
                $lastFieldAccessed \leftarrow$
                maximum index of $fieldAccesses$;
                $lastInstanceAccessed \leftarrow$
                maximum index of $instanceAccesses$;
```

Algorithm 3: repOkExec1(maps): Finds target: the corrupted instance and the field; updates the indices.

```
    $corruptField \leftarrow fieldAccessList[lastFieldAccessed]$;
    $corruptInstance \leftarrow instanceAccessList[lastInstanceAccessed]$;
    $symbolicVar \leftarrow getSymbol(corruptInstance)$;
    update map for $corruptField$ at key $corruptInstance$ with $symbolicVar$;
    assert map for $corruptField$;
```

Algorithm 4: updateMap(maps): Replaces value of the corrupted field with symbol.

the corresponding constraint and adds it to the constraint list. For same reasons as in Algorithm 3, for any assignment statement, it tracks the index to the last built constraint in the list. At the end, before returning from the method, it updates the index, *corruptConstraint* that points to the target corrupted constraint in the list.

The main algorithm, Algorithm 1, then negates the target corrupt constraint in the constraints list. Conjunction of the constraints up to the negated constraint in the list yields the modified path condition. It then invokes the constraint solver to solve for the modified path condition. The solution (if exists) is then reflected back to update the data structure state in an attempt to repair the corruption.

Figure 4.2 shows the repair attempts of our approach for the example binary tree. Note that the solution is a new node for the corrupted field (the *left* child of the root node). But this repair attempt changes the size of the data structure which mismatches with the *size* field – resulting in another corruption. As a result, we need a second repair attempt and that updates the *size* field to finally repair the data structure. We acknowledge that creation of a new node might arise complexities on what the initial values of the new node should be. In such a case, an alternate

```
while more statements do
    match statement do
        case branch:
            create constraint of outcome;
            add constraint to constraintsList;
        case (x ← e):
            wrap e with pointer to maximum index of constraintsList;
        case return e:
            if operand stack top e is wrapped then
                unwrap the e and get pointer: ptConstraint;
                corruptConstraint ← ptConstraint;
            else
                corruptConstraint ←
                maximum index of constraintsList;
```

Algorithm 5: repOkExec2(): Builds the path condition from *constraintsList* and *corruptConstraint*

solution could be to use *null* for the corrupted field in the first attempt. We plan to investigate this issue in the future.

4.3 Implementation

We implement our generic repair algorithm on top of the new dynamic symbolic execution engine for Java, called Dsc. Dsc works on top of any standard Java virtual machine. It does not require modifications of the virtual machine or the user code. This means we can repair existing Java code when it is executed on a standard JVM.

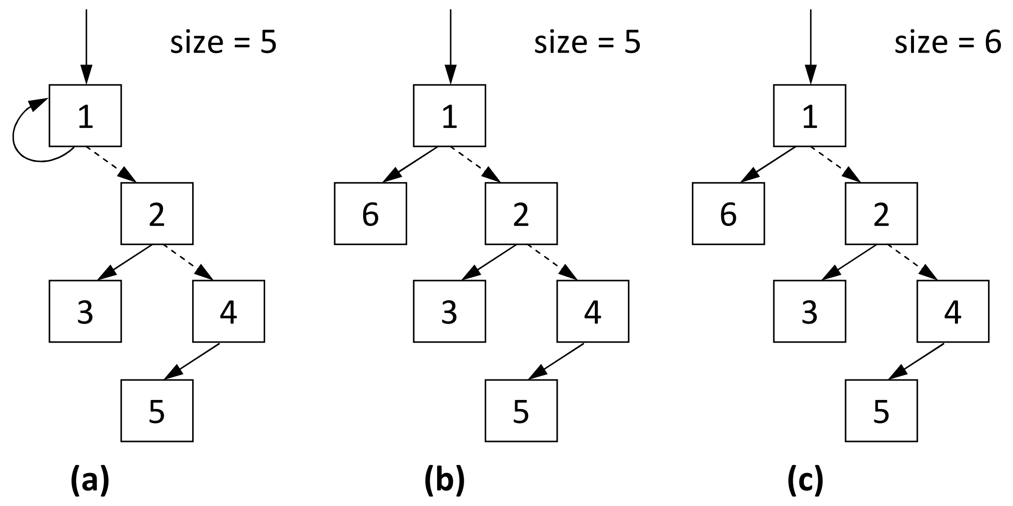


Figure 4.2: Directed approach in DSDSR. Solid lines represent *left* field and dashed lines represent *right* field. Initially (a), the binary tree is corrupt, as the first node's left field points to the root, creating a cycle, which is incorrect according to repOk correctness condition. In first attempt (b), DSDSR creates a new node for left field. Finally, in (c), updates the *size* field to reflect the repair.

Dsc analyzes user code at the bytecode level. It uses the instrumentation facilities provided by Java 5 to instrument user code at load-time, using the ASM bytecode instrumentation framework [34, 24].

4.4 Evaluation

In this section we present our experiments and the result that shows how existing repair algorithms depend on the style of the correctness condition. We evaluate our approach and Juzi [56] and show that ours is less dependent on the style of the correctness condition and takes less repair attempts in repairing a corrupted data structure.

4.4.1 Subject Data Structures

Singly-linked list: We considered the singly linked list data structure given in Figure 4.3. Apart from the usual reachability constraint of a singly linked list, the correctness condition for this structure is that the first node has to have a value that is equal to the number of nodes in the list.

```
public class Node {
  int value;
  Node next;
  // ..
}

public class LinkedList {
  Node header;
  // ..
  public boolean repOk() {
    Node n = header;
    if (n==null)
      return true;

    int length = n.value;
    int count = 1;

    while(n.next != null) {
      count += 1;
      n = n.next;
      if (count > length)
        return false;
    }

    if(count != length)
      return false;

    return true;
  }
}
```

Figure 4.3: Example singly linked list data structure, abbreviated, consisting of a Node class and a LinkedList class. Method repOk is a contrived correctness condition for the linked list, which may be invoked by assertions throughout the program.

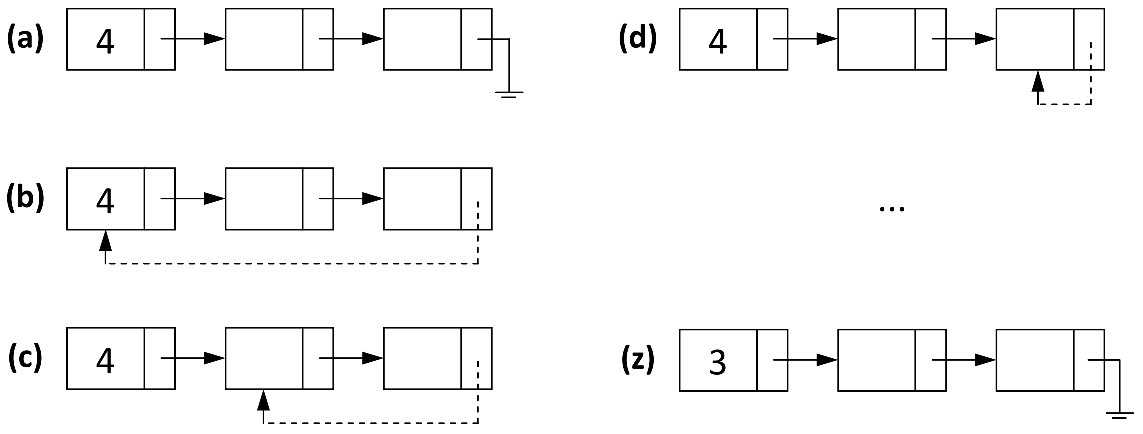


Figure 4.4: Exhaustive approach in Juzi. Initially (a), the LinkedList is corrupt, as the first node contains value 4, which is incorrect according to the Figure 4.3 repOk correctness condition. Dotted lines show the first three repair attempts (b, c, d). Omitted are several subsequent repair attempts. Ultimately repair culminates in the correct list (z).

The repOk method in Figure 4.3 checks the correctness as follows: it first stores the value field of the first node in a temporary variable named length. Then the method iterates over the list nodes to count them. This loop terminates prematurely once the node count exceeds the value of the length variable. This also prevents lists that are circular from forcing repOk into an infinite loop. Finally, the length variable is compared with the node count, to produce the desired answer.

Binary Tree: We took the binary tree data structure example as described in Figure 3.1. As mentioned earlier we took the constraints from [56], specifically these are: (1) acyclic along the left and right pointers and (2) the number of nodes reachable from the root node along the left and right fields is stored in the *size* field.

4.4.2 Induced Errors

For the singly linked list, we constructed lists of different lengths. Specifically, each run constructed a correct singly linked list of a given length and then corrupted the value of the first node by increasing it by one.

Similarly, we conducted experiments with binary trees of different sizes. Each run constructed a correct binary tree of a given size and then corrupted one of the leaf node's left or right field by pointing the root as its child. To emphasize the fact that writing good correctness condition is very hard and repair actions heavily depend on the programmer defined correctness condition, we considered slightly different versions of the same correctness condition and applied both the repair tools. We considered three versions of the same correctness condition. They vary in places of return statements, e.g., the first one returns false immediately when it detects a corruption (Figure 3.1), the second one waits until the end of the method (Figure 3.2) and the last one returns false immediately if the corruption is in the right pointer but returns late if it is in the left pointer and vice versa.

4.4.3 Experiment Setup

We conducted the experiment with the latest version of Juzi (0.0.0.1) which we obtained from the Juzi website[1] and took all measurements on a Sun HotSpot JVM 1.6.0_17 running on Windows on an intel laptop 2.26GHz Core2 Duo processor.

At each experiment run, after inducing an error, we invoked one of the repair tools, and measured the time the tool takes to repair [2].

[1]http://users.ece.utexas.edu/~elkarabl/Juzi/index.html

[2]The current version of our prototype makes only one repair attempt. For cases where we needed multiple repair actions to finally correct the data structure, we ran the tool multiple times—adding the time taken each time to simulate the final repair.

4.4.4 Results

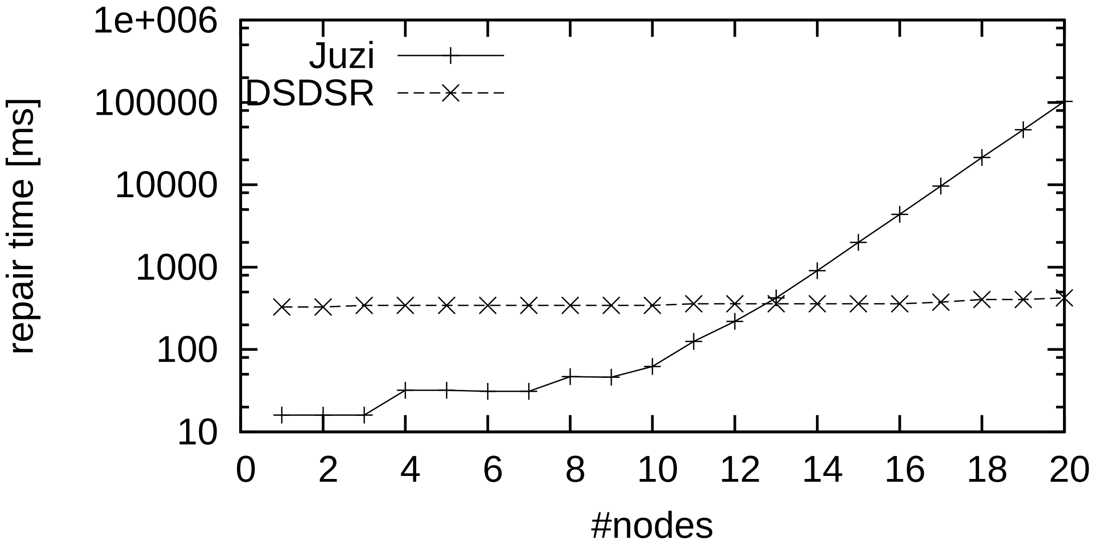


Figure 4.5: Result of applying the linked list example of Figure 4.4 to lists of different lengths. #nodes is the number of nodes in the list. Repair time is the time a tool took to produce the correct repair action. Smaller repair times are better.

Figure 4.4 shows an example of how Juzi exhaustively repairs the corrupted singly linked list. Figure 4.5 shows the result of our experiment with the singly linked list [3]. In the experiment both tools succeed in that they terminate with producing the correct repair action in all cases. Juzi repairs small lists more efficiently than our prototype implementation. But starting with 13 nodes, our approach is more efficient. This makes sense intuitively, as an exhaustive approach such as Juzi is bound to be inefficient for larger data structures, motivating more directed approaches such as ours.

[3]At the time of this experiment, our prototype suggested but did not perform a repair action. The implementation of performing a suggested repair was later straightforward and added only a negligible overhead.

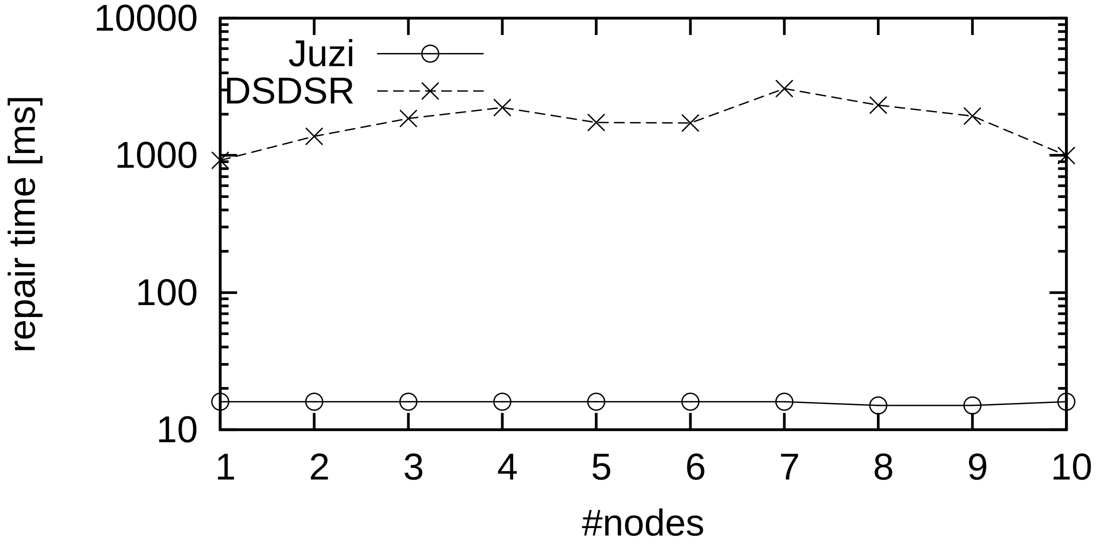


Figure 4.6: Repair time for binary trees of different sizes and immediately returning correctness condition (Figure 3.1). #nodes is the number of nodes in the binary tree. Repair time is the time a tool took to produce the correct repair action. Smaller repair times are better.

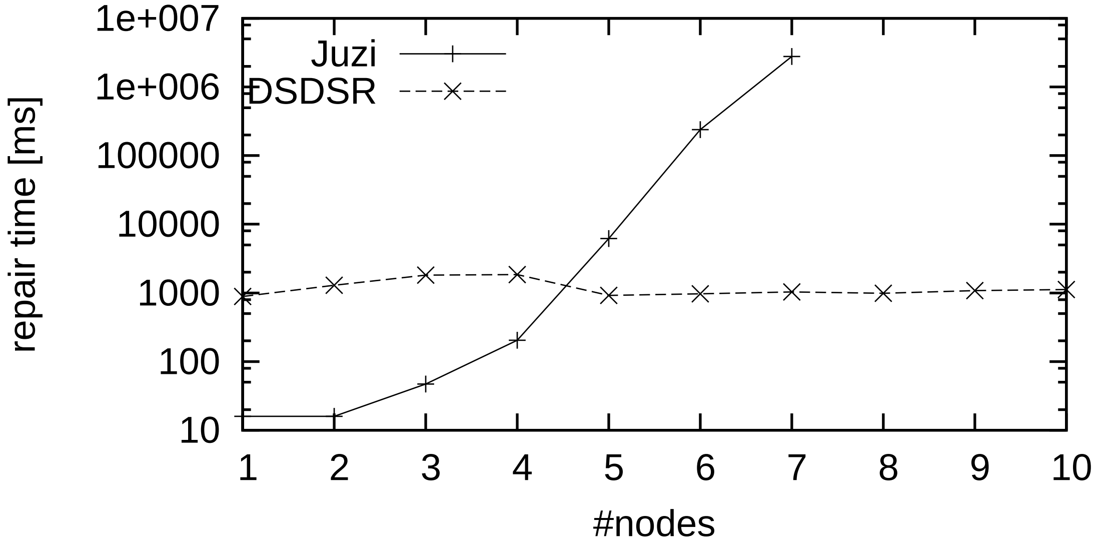


Figure 4.7: Repair time for binary trees of different sizes and late returning correctness condition (Figure 3.2).

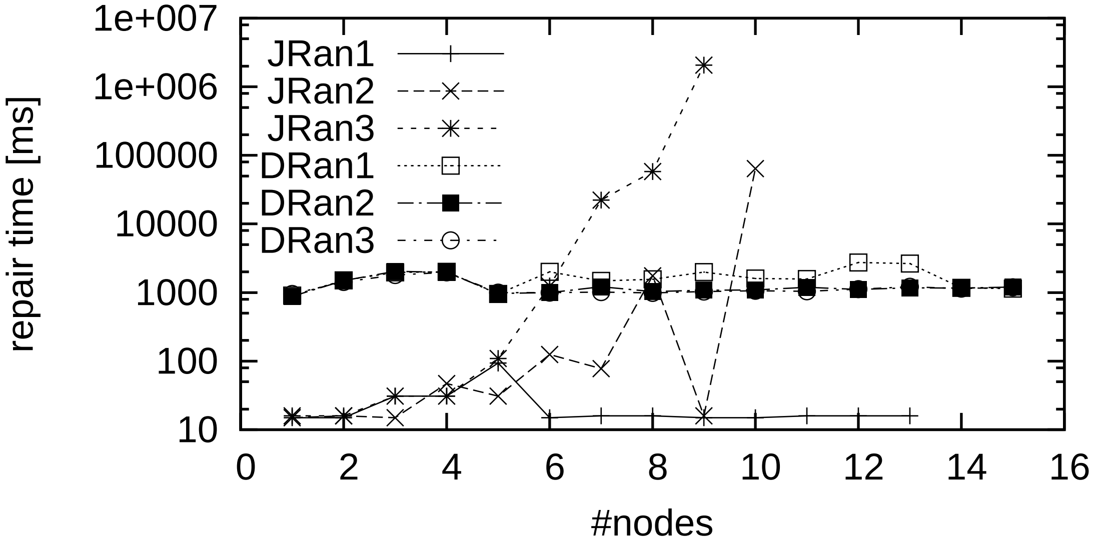


Figure 4.8: Repair time for three sets of random binary trees and yet another variation of the correctness condition. We had to terminate Juzi for 10, 11, or 14 nodes.

Figures 4.6, 4.7, and 4.8 show the result of our experiments with the binary tree. For a carefully designed correctness condition, Juzi repairs more efficiently than our prototype implementation, as shown in Figure 4.6. But with a modified correctness condition, as shown in Figure 4.7, starting with 5 nodes, our approach is more efficient.

Figure 4.8 shows the result of our experiment with third version of the repOk method that returns false immediately if the corruption is in the *right* field but returns late if the corruption is in the *left* field. We found similar results for the opposite case of this repOk. We applied both the tools for three sets of binary trees, where each set had randomly built binary trees containing from 1 to 15 nodes. In each case, we had to terminate Juzi prematurely because it was taking too long to repair. This is intuitively expected as an exhaustive approach such as Juzi is bound to be inefficient for larger data structures. This motivates our more directed approach, which takes approximately the same amount of time to repair—irrespective of the size of the data structure and variation in the correctness conditions.

4.5 Limitations

The main concerning issue with a data structure repair approach is that it does not guarantee producing the same data structure that a normal execution of the program would have produced [49, 54]. It also incurs the possibility of data loss. Therefore, the program might behave differently and even produce unexpected results after a successful repair. We acknowledge that, such limitations prevent applying our approach to programs where absolute correctness is more desirable than continued execution. Examples of such programs can be cryptographic algorithms and applications that have strict numerical calculations [49].

However, in case of a corruption, similar to other approaches, the main goal of our repair approach is to enable the program for continued execution rather than having it halted or in the worst case, crashing it and thereby loosing all it's volatile state.

Our approach that takes the correctness condition or the repOk method as input, assumes the successful and correct implementation of the repOk method. As a result, a faulty repOk method can jeopardise the whole repair effort. But we believe that a programmer who uses a data structure also knows the details of it and since she uses the same programming language that she uses to implement the data structure to write the repOk method, she could implement it correctly.

From the implementation perspective, since our approach is built on top of the dynamic symbolic execution engine Dsc and uses Z3 as the constraint solver, it carries all the limitations of these frameworks. For example, current prototype implementation has limitations on solving floating point arithmetic, infinite loop, etc. Moreover, since our evaluation does not contain all possible data structures and corruption scenarios, it is neither sound nor complete. It may contain bugs and fail to repair many corrupted data structures.

Chapter 5

Priority Based Corruption Repair

In this chapter we propose an updated approach that tries to address the second problem mentioned in Section 3.3, namely, the error location dependency problem of the existing repair approaches. That is, for the same error in different positions in the same data structure, existing approaches can take different execution time and number of attempts in repairing the structure. Moreover, while trying to fix the problem, these repair attempts sometimes change other data values instead of the original corrupted one and results in data loss.

To illustrate the problem, we consider the binary search tree data structure introduced in Section 3.3 that we took from the authors of Juzi [56]. For the ease of explanation we present the same example again in Figures 5.1 and 5.2.

As mentioned earlier, the repOk method in Figures 5.1, 5.2 check the following constraints of the binary search tree: (1) no-cycle: acyclic along the left and right pointers, (2) The left (right) subtree of a node contains only nodes with keys less (greater) than the node's key; the subtrees must be binary search trees, and (3) the number of nodes reachable from the root node along the left and right fields is stored in the *size* field.

We introduce corruptions that break the second constraint. Specifically, we induce corruptions that change the data values of different nodes without introducing any cycle or changing the total number of nodes in the binary search tree. Figure 5.2 shows the code that checks the second constraint. As before, note that, instead of using recursion, which is commonly used in checking the constraint, authors of Juzi

```
public class Node {
  Node left;
  Node right;
  int key;
}

public class BinarySearchTree {
  Node root;
  int size;
  boolean return_immediate = false;
  // ..
  public boolean repOk() {
    // An empty tree must have zero in size
    if (root == null)
      return (size == 0);

    boolean result = true;
    Set<Node> visited = new HashSet<Node>();
    visited.add(root);
    LinkedList<Bst_Node> workList = new LinkedList<Bst_Node>();
    workList.add(root);

    // Checks for cylcles along left or right
    while (!workList.isEmpty()) {
      Node current = workList.removeFirst();
      if (current.left != null) {
       // The tree must have no cycles along left
       if (!visited.add(current.left)) {
         if (return_immediate)
           return false;
         result = false;
       }
       else
         workList.add(current.left);
      }

      if (current.right != null) {
       // The tree must have no cycles along right
       if (!visited.add(current.right)) {
         if (return_immediate)
           return false;
         result = false;
       }
       else
         workList.add(current.right);
      }
    }
```

Figure 5.1: Same Figure as Figure 3.4 example binary search tree data structure, consisting of a Node class and a BinarySearchTree class. Partial *repOk* method that checks acyclicity.

```
    // Size must be equal to #visited nodes
    if (visited.size() != size){
      if (return_immediate)
         return false;
      result = false;

    }

    // Check for BST property
    int min = -1;
    Stack<Node> stack = new Stack<Node>();
    Node p = root;

    while (p != null) {
      stack.push(p);
      p = p.left;
    }

    while (stack.size() > 0) {
      Node q = stack.pop();
      if (q.key < min) {
        if (return_immediate)
          return false;
        result = false;
      }

      min = q.key;
      Node r = q.right;

      while (r != null) {
        stack.push(r);
        r = r.left;
      }
    }

    if (return_immediate)
      return true;

    return result;
  }
}
```

Figure 5.2: Example binary search tree data structure, *repOk* continues from Figure 5.1, checks correctness condition for the binary search tree.

made the repOk method use a stack and iteratively check the property in a bottom-up fashion.

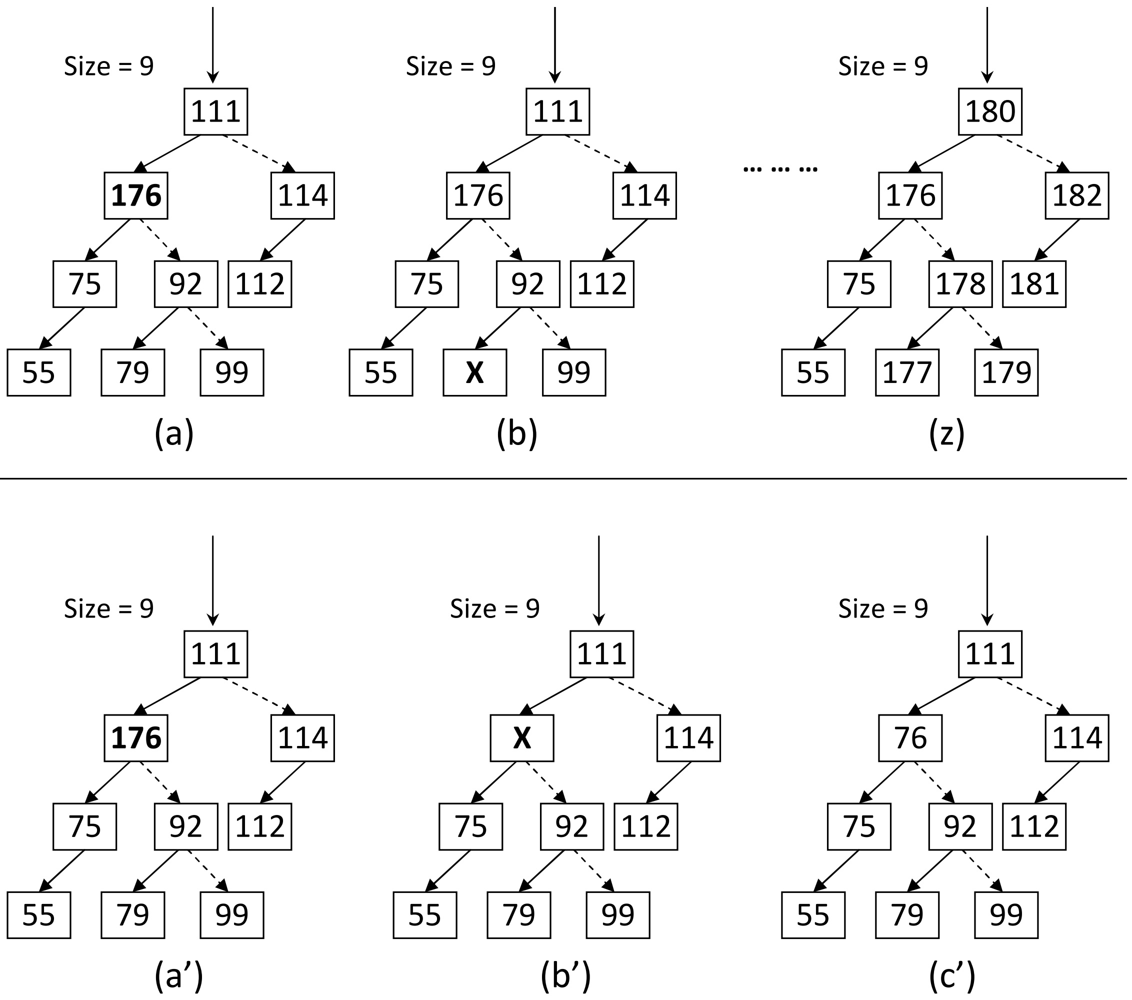


Figure 5.3: Initially (a), the binary search tree is corrupt in root's left child (176). Solid arrows represent left child to a node, dotted as right child. X in *key* field represents symbolic variable in repair attempts. Upper: exhaustive approach in Juzi, omitted are several subsequent repair attempts. Ultimately repair culminates in the binary search tree (z) which is different in all but three nodes. Bottom: starting with the same corrupted structure in (a'), our approach that allows *repOk* to return late, detects the actual corrupted field in (b'), and, repairs the structure in first attempt retaining key values in (c').

Figure 5.3(a) shows an example of a corrupted binary search tree of nine nodes. The corruption is with the left child of the root node: namely, it has a key value that is greater than the root's key value. For simplicity, we will refer to a node by its key value from now on.

The repOk method in Figure 5.2 uses a stack to check the second invariant iteratively in a bottom-up fashion. The code works as follows: it initializes the local variable *min* to -1 and starting from the root node, pushes all the non-null *left* nodes into the stack (lines 62–65). Figure 5.4(a) shows the stack state for the example at this point. Next, while the stack is not empty (lines 67–82), it pops a node q from the stack and performs basically two things: first, it compares q's key value with the *min* (at line 69); if the *min* is greater than the key value then the repOk detects an error. Otherwise, it updates the *min* with the key value. Figure 5.4 (b) – (d) shows the updates of *min* for the nodes in the left branch of the example binary search tree. Second, it pushes the non-null right node of q (if it exists) and all its non-null *left* nodes into the stack. Figure 5.4 (e) shows the stack state at this point for the example binary search tree. The *while* loop continues performing these operations until the stack is empty or *repOk* detects a violation and returns false.

For our example, in the second iteration of the loop, while the stack is in the state of Figure 5.4 (f), it detects an error. Specifically, the current *min*, 176 is greater than the popped value 79. Intuitively, this is an error as the node 79, being in the right sub-tree of node 176, should have a greater value than 176 as its key, or the other way around.

Note that the last directly accessed field before the corruption is detected is the *key* of node 79. The check that detects this error (at line 69) involves the local variable *min* through which another data field (*key* of node 176) is indirectly accessed.

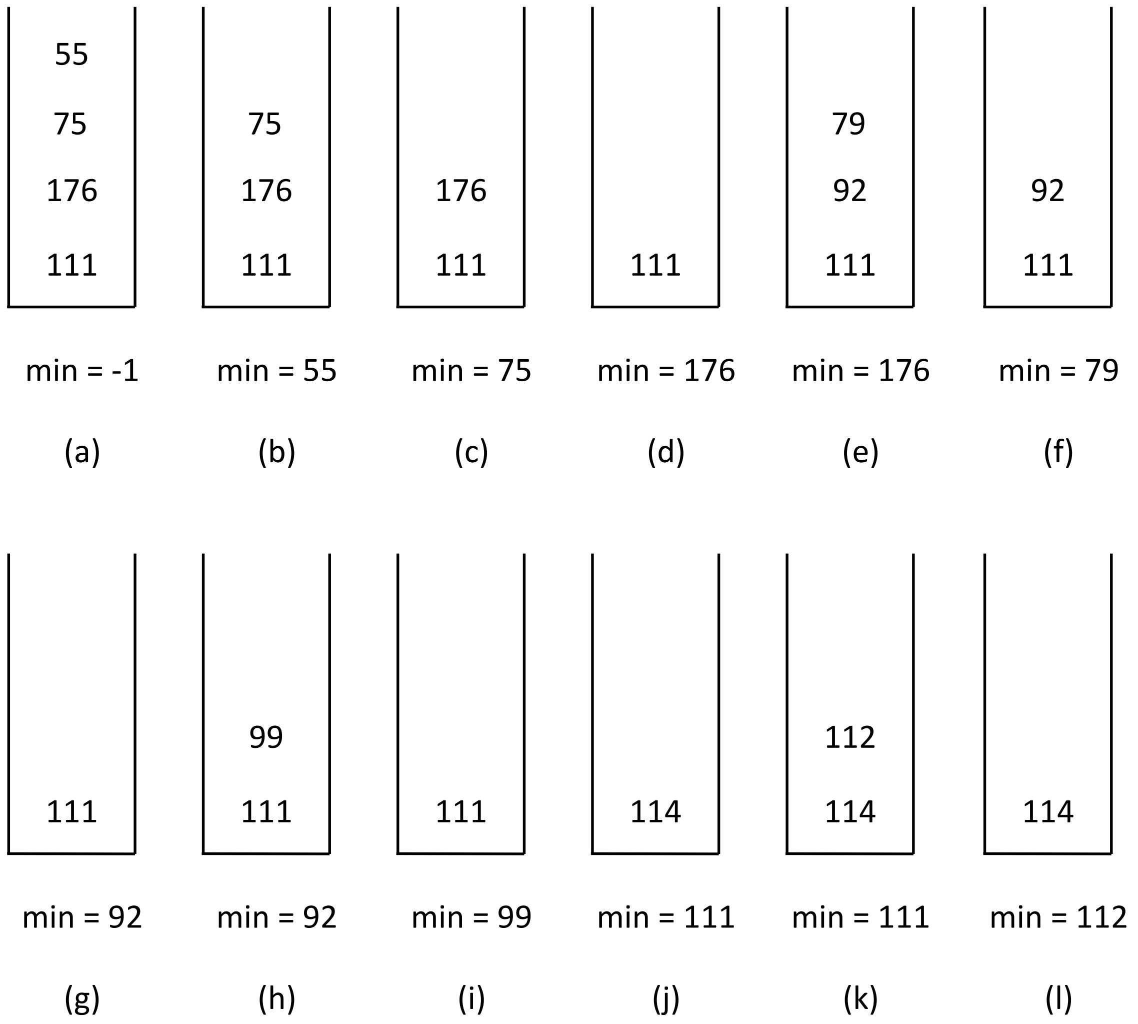


Figure 5.4: Different stack states while the repOk runs on the binary search tree in Figure 5.3. With the *min* set to -1, (a) shows the stack state after all the nodes in the left branch of the tree are pushed in; (b)–(d) shows stack pops; *min* is updated with the popped value if the popped value is greater than the *min*. Stack states after the pushes at lines 38–43 in the repOk are shown in (e)–(f), which keeps the *min* unchanged. (f) also shows the stack state before the error is detected at line 69 of the repOk method.

Recall that Juzi always considers the last accessed field as corrupted and cannot detect indirect field accesses. As a result, in Juzi's repair attempt for the example binary search tree, it tries to mutate the value in the *key* field of node 79 but ignores the value in the *key* field of node 176. Juzi replaces it with a symbolic variable as shown in Figure 5.3(b), and tries to solve for it. The first attempt yields the value 177 – which further corrupts the data structure. Subsequently (omitted from the figure) Juzi backtracks in the list of field accesses and continues its repair actions in an exhaustive fashion. Finally, Juzi repairs the data structure in (z). Note that it validates the original corrupted node by changing almost all the nodes' key values (except three, including the original corrupted node).

Our key insight is that we should not consider only the last accessed field before the corruption as the target corrupted field. Rather we should consider all the involved fields in any corruption, both directly and indirectly accessed fields as targets. For our example binary search tree this would make both the *key* fields of nodes 79 and 176 (accessed indirectly through the local variable *min*) as targets. Moreover, when a corruption is detected, instead of returning from the repOk method immediately, we should rather let it run to discover more corruptions (if exists) and gather more constraints on the overall data structure state. This way we can prioritize among the target corrupted fields, reduce the search space with the gathered constraints and guide the repair approach to save futile attempts and improve repair efficiency.

5.1 Updated Algorithm

Figure 5.5 gives a high-level overview of our updated algorithm. As before, as part of its normal execution, a program invokes its correctness specification method, repOk. When repOk fails, i.e., returns false, execution is handed over to the extended dynamic symbolic engine, which in turn invokes the instrumented version of the repOk, shown as I-repOk.

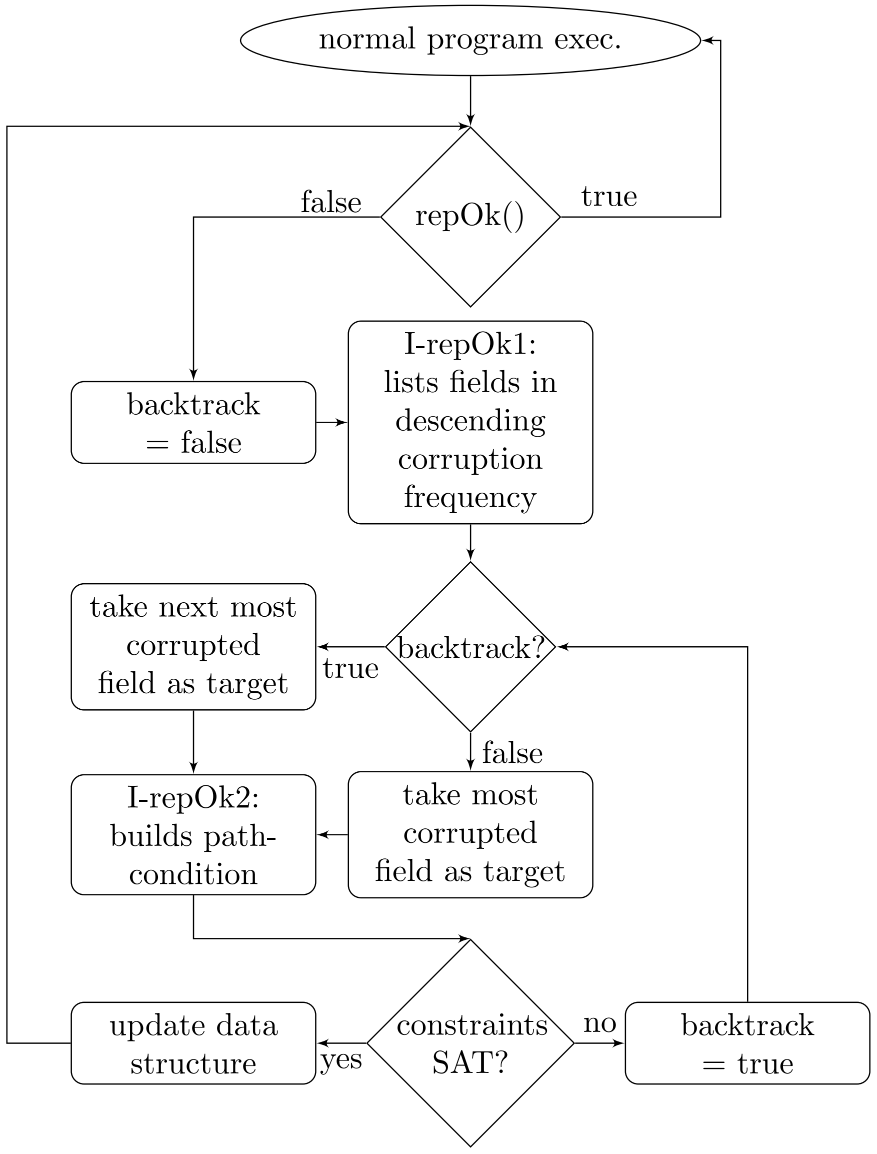

normal program exec.
repOk()
false
true
backtrack = false
I-repOk1: lists fields in descending corruption frequency
backtrack?
true
false
take next most corrupted field as target
I-repOk2: builds path-condition
take most corrupted field as target
constraints SAT?
yes
no
update data structure
backtrack = true

Figure 5.5: High-level overview of our approach. I-repOk1 and I-repOk2 are the instrumented versions of repOk. I-repOk1 returns late: for each detected corruption, it increments the counter for the involved fields and at the end, based on the occurrence count, prioritizes all the corrupted fields. I-repOk2, with the target field replaced by a symbolic variable, executes and returns late: it builds the path-condition by negating all the constraints involved in any corruption involving the symbolic variable.

The instrumented repOk or I-repOk is called twice. In both the cases, when an error is detected, it does not return immediately, rather it executes and returns late. The first call to I-repOk, represented as I-repOk1, monitors the program execution and logs all the involved fields in any corruption.

Whenever I-repOk1 detects an error, rather than returning immediately making the last accessed field (before the detected corruption) as the target, it logs all the accessed fields involved in the corruption, increments the counters for each of the involved fields (maintained in a map) and continues the execution. At the end of its execution, I-repOk1 lists all the corrupted fields in the descending order of occurrence, i.e, the field that appeared the most in the detected corruption(s) appears at the top of the list. We call the field at the top of this list as the *most corrupted* field. If there are multiple candidate fields for the top position (i.e., with equal maximum values in their counters), we apply our heuristic that considers the access sequence of these fields and prefers the field that was accessed relatively later in the repOk method over other fields to be in the top position. For example, in the example binary search tree, both the *key* fields of the nodes 79 and 176 were involved in the only detected corruption (hence both have a counter value of 1) and both are the targets. But since the *key* field of 176 was accessed later, it is selected as the target corrupted field.

Once the most corrupted field is selected as the target, the algorithm replaces it with a symbolic variable and calls the instrumented repOk for the second time, represented as I-repOk2. I-repOk2 monitors the program execution and builds the path condition. Note that, I-repOk2 also does not return immediately as it detects any corruption, rather completes its execution and returns false late. At the end, with the full path condition at hand, it modifies the path condition by inverting the constraint(s) (involving the symbolic variable) that correspond to the detected corruption(s). In our example, replacing the *key* of the node 176 with a symbolic variable X, the observed constraints in the path condition involving X are : $(X \geq 75)$ and $(X > 79)$ (from Figures 5.4 (c) and (e)). Since the later constraint represents the detected corruption, the algorithm inverts it to get the modified path condition as: $... \wedge (X \geq 75) \wedge (X \leq 79) \wedge ....$

Next the algorithm calls the constraint solver with the modified path condition as input. If the modified path condition is solvable, the constraint solver gives back a solution for the symbolic variable that satisfies the path condition. Mutating the target corrupted field with the solution yields a new data structure state which, if passes the repOk correctness condition, completes the repair process. If it does not pass the repOk method, the repair process is repeated. But if the modified path condition is not solvable, the algorithm backtracks and chooses the next top corrupted field as the target and tries to repair. To prevent an infinite loop of repair attempts, the algorithm terminates (not shown in Figure 5.5) after reaching a user-defined number of futile attempts. In our example, the modified path condition is solvable and the constraint solver gives 76 as the solution for the symbolic variable X. Mutating the corrupted *key* with 76, as shown in Figure 5.3 (c‘), gives a new data structure state that passes the repOk method and completes the repair.

5.2 Evaluation

5.2.1 Experiment on Text Book Data Structures

We conducted our experiments with four types of data structures: singly-linked list, doubly-linked list, binary search tree and binary heap. We selected these basic data structures as they are easy to implement and their correctness either depends on the structural constraints or data constraints or both. We describe their structural and data constraints, size and the induced errors for the experiments in the following sections.

5.2.1.1 Subject Data Structures

Specialized singly-linked list or SSLL: we reused our previous example from [82]. The integrity constraints are : (1) no-cycle: acyclic along the *next* pointer; (2) the first node's *value* should be equal to the length of the list. The length of the list is the number of nodes reachable from the *header* following the *next* pointer. An empty list has *null* in the *header*.

Doubly-linked list or DLL: we took the algorithm from the authors of Juzi [56]. The integrity constraints of that doubly-linked list are: (1) circular structure along *next* (and also *prev*); (2) transpose relation between the *next* and *prev* fields; and (3) number of nodes reachable from the *header* following *next* is stored in *size*. An empty list has *null* in the *header* and its *size* is 0.

Binary search tree or BST: we also took the binary search tree algorithm from the authors of Juzi. The constraints of the binary search tree here: (1) no-cycle: acyclic along the *left* and *right* pointers; (2) The *left (right)* subtree of a node contains only nodes with keys less (greater) than the node's key; the subtrees must be binary search trees; and (3) the number of nodes reachable from the *root* node along the *left* and *right* fields is stored in the *size* field.

Binary heap or BH: we took the algorithm from the popular algorithm text book by Cormen et al. [39]. It uses an integer array based implementation of the max-heap property of binary heap: for every entry at i, other than the root (0-th position), $A[Parent(i)] \geq A[i]$ – where Parent(i) is $\lfloor i/2 \rfloor$. Because of the array based implementation, it did not have any structural constraints.

Table 5.1: Structural and data constraints of the data structures used in the experiments.

Data Structure	**Structural Constraints**	**Primitive Type Constraints**
Special singly-linked list (SSLL)	acyclicity, reachability	first node's value should be equal to the length of the list
Doubly-linked list (DLL)	reachability, transpose, circularity	n/a
Binary search tree (BST)	acyclicity, reachability, one parent	natural order on elements
Binary heap (BH)	n/a	max heap: for any index i of array A– $A[Parent(i)] \geq A[i]$; where Parent(i) is $\lfloor i/2 \rfloor$

Table 5.1 follows the format from [57] to summarize the structures and integrity constraints of these data structures.

5.2.2 Induced Errors

Table 5.2 describes induced errors for each of the data structures. These errors corrupt different locations of the data structures. For example, for the binary search tree, the first error that swaps the root's key with its left child's key (BST-E1), corrupts the data structure next to its root, at the top. It violates the data constraint that the root's left subtree should contain nodes with keys less than the root's key. Another such data constraint violation occurs when the key of the first leaf node, discovered by the breadth-first-search approach, is swapped with its parent node's key (BST-E2). This error corrupts the bottom part, the edge of the binary search tree. Similarly, errors BST-E3 and BST-E4 corrupts different parts of the binary search tree.

As another example, for the specialized singly-linked list, the first error (SSLL-E1) corrupts the data structure at the end (i.e., far from the node pointed by the *header*) by making the last node's *next* point to the first node. This corruption violates the structural acyclicity constraint of the singly-linked list. On the other

Table 5.2: Description of different induced errors in the data structures used in the experiments. First column shows abbreviated names of the data structures used, namely, special Singly-linked list, Doubly-linked list, Binary Search Tree and Binary Heap. Second and third column, respectively, show their annotations (used in Table 5.3) and describe how each of them corrupt a corresponding data structure. Note that, for Binary Search tree and Binary heap we used the same kinds of errors.

Data Structure	**Error Type**	**Error Description**
SSLL	E-1	Self loop by *next* pointer in third node
	E-2	First node's *value* is not equal to the length of the list
	E-3	Cycle as the last node's *next* points to the first node
	E-4	Second node's *next* points to the last node
DLL	E-1	Self loop by *next* pointer in third node
	E-2	First node's *next* points to the last node
	E-3	Third nodes' *prev* points to the last node
	E-4	Fourth node's *prev* points to the second node
BST / BH	E-1	Swaps root's *value* with its *left* child's *value*
	E-2	First leaf node's *value*, found by BFS with it's parent's *value*
	E-3	Swaps first node's *value* in level-1 with it's *right* child's *value*
	E-4	Increments fifth node's *value*, found by breadth-first-search, by 100

hand, the second error (SSLL-E2) changes the *value* of the first node (pointed by the *header*) and violates the data constraint that it should be equal to the length of the list.

Similar errors were introduced for other subject data structures that corrupt different locations of the data structure. Table 5.2 summarizes these induced errors for different data structures. Though the binary heap is an array-based data structure it can be visualized as a binary tree [39]. We induced errors by modifying the values at different indices of the data structure. Since these errors resemble the errors induced in the binary search tree, Table 5.2 describes the induced errors for the binary search tree and the binary heap together.

5.2.3 Experiment Setup and Time-out

We applied three repair tools: the state-of-the-art repair tool Juzi [56], our previous prototype DSDSR [81] that implements the algorithm in Section 4.1, and our latest prototype DSLTR that implements the updated algorithm in Section 5.1 on the subject data structures with the induced errors and measured their repair performance. Specifically, we measured their execution time and counted the number of repair attempts. We also monitored if the final repair changed the size of the initial corrupted data structure and counted the affected nodes in the process.

For all the subject data structures, we allowed the repOk method to return late. For our prototypes, DSDSR and DSLTR we allowed a maximum of 100 repair attempts before we terminated and declared it a *time-out*. This was done to avoid infinite loop in repair attempts. For Juzi we did not have any such restrictions and allowed it to run till the end.

We ran all the experiments on a HotSpot 1.6.0_32 JVM on Windows 7 on a 2.26GHz 32-bit Intel Core2 Duo processor laptop with 3GB RAM.

5.2.4 Results

We start the discussion by describing the issues that we faced choosing the sizes of the different subject data structures in our experiments with Juzi. Note that when we mention a data structure, we mean the corrupted version of the data structure in general, not with any specific induced errors as mentioned in the Section 5.2.2.

We observed that for the larger data structures, Juzi took too long to complete its repair executions. For example, for the specialized singly-linked list with 15 nodes, it was more than 2 hours but Juzi was still running its repair actions. When we reduced the size to 10 nodes, Juzi still did not complete its execution after 2 hours. We observed similar behaviors with the doubly linked-lists. As a result, we chose 5 nodes-sized data structures for both the specialized singly and doubly-linked lists in our experiments.

For the binary search tree examples, Juzi could handle relatively larger data structures than the singly or doubly-linked lists. It could repair up to 15 nodes sized binary search trees in reasonable time (e.g., less than 30 minutes). But for more than 15 nodes binary search trees, Juzi was again taking too long to terminate. Therefore, in our experiments, we chose to build 15 nodes-sized binary search trees.

Juzi cannot handle array based data structures. As a result Juzi could not repair our array based binary heap data structures. For comparisons between our prototype tools DSDSR and DSLTR for the binary heap, we simply took the size that we used for the binary search trees and used an integer array of 15 elements for the experiments.

Table 5.3 shows the experimental results. For the subject data structures with induced errors, it shows the performance of the repair tools in terms of the following metrics: execution time (in milliseconds), number of repair attempts, whether the size of the original structure is altered in the final repair and the number of affected nodes.

A highlevel observation from the table is that, for a fixed sized data structure with errors at different locations, the repair performance of the tools can vary. For example, for the binary search tree, as mentioned earlier, both the errors E1 and E2 swap a node's key with its parent's key but at different locations. To repair these errors in the two versions of the binary search tree (V1 and V2), the tools show varying metrics values. This is intuitive as errors at different locations may have different influences over the rest of the structure. Detecting and repairing errors on such *depended-on* locations may have differing affects in the execution time, repair attempts, etc.

With respect to execution time and number of repair attempts, Juzi's performance varies the most. That is, Juzi shows more error location dependency for its repair performance, but our prototype tools are less dependent on the error locations. For example, for the first version of the binary search tree (i.e. V1), Juzi took more than 22,000 attempts in repairing the error E1 but only 3 attempts for E2; whereas our prototype DSLTR took only 3 and 1 attempts, respectively.

We acknowledge that our earlier prototype DSDSR timed-out in repairing the E1 error for the binary search tree. DSDSR showed such time-outs for few other errors for other data structures as well. We investigated and found that since DSDSR did not backtrack in finding a target corrupted field when the modified path-condition was unsatisfiable, it resulted in futile attempts.

Regarding the number of affected nodes in the final repair of the data structures, the results (i.e., the *Aff.* columns in the table) show that Juzi modifies more nodes than our prototype tools. That is, Juzi may incur more data loss in the final repair. Moreover, for the doubly-linked list we saw Juzi changed the size of original data structure in its final repair.

Among the three tools, DSLTR showed steady performance and in general is less dependent on the error locations. It can repair a corrupted data structure with smaller number of repair attempts and results in less affected nodes.

5.2.5 File System Example

We next present a simple file system example that we took from Demsky's paper [50]. The file system consists of three parts: the directory, the file allocation table (FAT), and an array of file blocks. Each file is considered to have a chain of file blocks. The directory has fixed size of entries. Each directory entry represents a file

Table 5.3: Experiment results for different data structures ($D.S$): the first column shows abbreviated name of a data structure with its size in parenthesis. Second column shows the induced error type. For the binary search tree, we used two randomly built trees with different node values, they are represented by *V1, V2*. For each of the repair tools, i.e., Juzi, DSDSR and DSLTR, following columns show experiment results for different metrics: required repair time in milliseconds *(T(ms))*, number of repair attempts made *(Att.)*, if the final repair has altered the size of the initial corrupt data structure *(Alt?)*, and finally, how many node(s) were affected by the repair, *(Aff.)*. Juzi could not handle array based binary heap data structure, we also had time-outs ($T.O$, allowed 100 attempts) for DSDSR and DSLTR for some data structures. We could not measure performance of the tools for these cases and the metrics are represented as not-applicable *(n/a)*.

D.S.	**Error**	**Juzi**				**DSDSR**				**DSLTR**			
(size)	**Type**	**T(ms)**	**Att.**	**Alt?**	**Aff.**	**T(ms)**	**Att.**	**Alt?**	**Aff.**	**T(ms)**	**Att.**	**Alt?**	**Aff.**
SSLL (10)	E-2	T.O	n/a	n/a	n/a	503	1	no	1	510	1	no	1
	E-1	16	35	no	1	858	2	no	1	845	2	no	1
SSLL	E-2	26	194	no	1	412	1	no	1	425	1	no	1
(5)	E-3	14	2	no	0	305	1	no	0	311	1	no	0
	E-4	20	34	no	1	365	1	no	0	404	1	no	1
	E-1	42813	99769	yes	5	T.O	n/a	n/a	n/a	T.O	n/a	n/a	n/a
DLL	E-2	40619	93224	yes	5	792	2	yes	3	865	2	yes	3
(5)	E-3	50342	118162	yes	5	378	1	no	0	396	1	no	0
	E-4	53091	128323	yes	5	371	1	no	0	397	1	no	0
	V1-E1	733	22363	no	8	T.O	n/a	n/a	n/a	4943	3	no	2
	V1-E2	32	3	no	1	1637	1	no	1	2013	1	no	1
	V1-E3	749	22225	no	4	T.O	n/a	n/a	n/a	4695	3	no	2
BST	V1-E4	749	22363	no	8	1451	1	no	1	1774	1	no	1
(15)	V2-E1	129745	3596457	no	6	T.O	n/a	n/a	n/a	4415	3	no	2
	V2-E2	1707857	42043961	no	2	2075	1	no	1	1232	1	no	1
	V2-E3	131462	3575857	no	4	T.O	n/a	n/a	n/a	4620	3	no	2
	V2-E4	119371	3314553	no	2	1591	1	no	1	1373	1	no	1
	E-1	n/a	n/a	n/a	n/a	2823	7	no	7	432	1	no	1
BH	E-2	n/a	n/a	n/a	n/a	429	1	no	1	414	1	no	1
(15)	E-3	n/a	n/a	n/a	n/a	2486	6	no	6	487	1	no	1
	E-4	n/a	n/a	n/a	n/a	1167	3	no	3	419	1	no	1

and holds the following information: the file name, size of the file in terms of blocks and the index of the first file block in the FAT.

The FAT is an array that is used to quickly find which file blocks are in use, which are free to allocate, etc. The number of entries in the FAT are equal to the available number of the file blocks. It also helps to maintain the chain structure of the file blocks. Specifically, if a file has two consecutive file blocks i and j in the chain, then the FAT will indicate this by having the value j at index i, i.e., FAT[i] = j. A file block can also be in two other states: the last block in a chain or not in any chain at all and free for allocation. These states are indicated by special values, the last block is indicated by the value -1, and free block by -2.

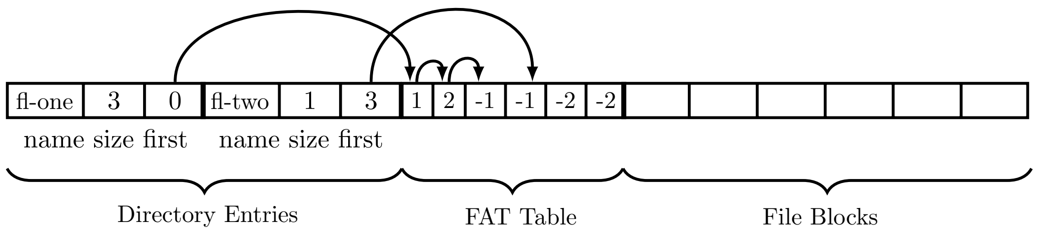


Figure 5.6: Correct file system example. Two entries in the directory for the two files. No file block is shared, no file block chain contains a free block.

Figure 5.6 shows an example of such a file system. Specifically, it has two entires in the directory for the two files, *fl-one* and *fl-two*. The first file contains three file blocks and starts from the first block (at index 0) in the FAT. The second file contains only one block, the fourth block (at index 3) in the FAT.

Similar to [50] we realize that even a file system this simple has many consistency constraints and we focus on the following two FAT constraints:

- Chain Disjointness: Each block should be in at most one chain.
- Free Block Consistency: No chain should contain a block marked as free in the FAT.

Note that, the file system example in Figure 5.6 is in a valid state since it complies to the two consistency constraints: the files do not share any common file block and no file block marked as free belongs to any of the file chains. Some of the other obvious constraints are: all the blocks except the last should be marked as used (state must be 1) and the last block should be marked as last (state must be -1). For simplicity, we ignore these constraints.

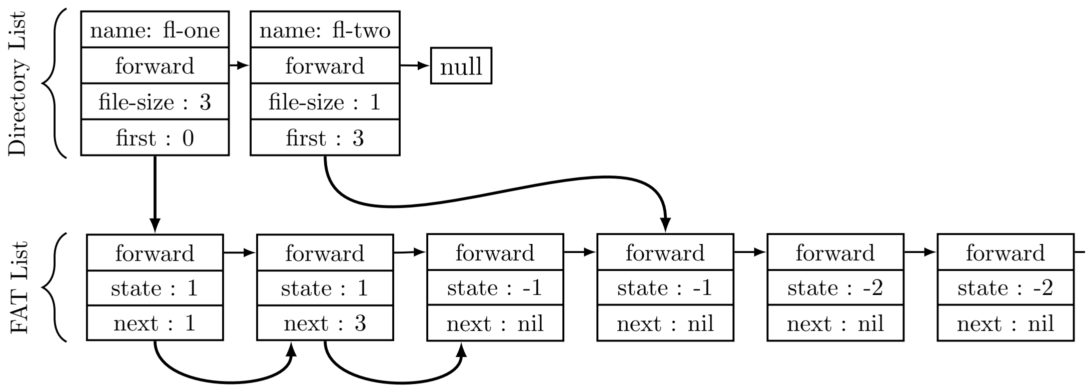


Figure 5.7: Equivalent file system example with modified structure as in Figure 5.6. Two entries in the directory for the two files. No file block is shared, no file block chain contains a free block.

Since, Juzi cannot handle arrays, for our experiments we modified the file system structure from arrays to linked-lists while keeping the basic concept. Figure 5.7 shows the equivalent file system example with the modified structure. Specifically, we introduced a reference, *forward* that maintains the list structure in both the directory and the FAT entries. Instead of the indices to the FAT array, the directory entries now hold a reference, *first* that points to the first FAT block. Similarly, each of the FAT entries hold a reference field, *next* that points to the next block in the chain. We introduced an integer field, *state* in each of the FAT entry nodes that represents the corresponding block's states. Namely, any free block has -2, any block that used in a file chain but is not the last block has 1 and the last block in the chain has -1 as their values in the *state* field.

```
public class DirectoryEntry {
  String name;
  DirectoryEntry forward;
  int size;
  FAT first;
}

public class FAT {
  FAT forward;
  int state;
  FAT next;
}

public class FileSystem {
   DirectoryEntry dirHeader;
   FAT fatHeader;
   // ...
   public boolean repOk() {

     HashSet<FAT> set = new HashSet<FAT>();
     boolean result = true;
     DirEntry dir = dirHeader;

     while (dir != null) {
       int count = 0;
       FAT node = dir.first;

       while (node != null) {

         // Chain disjointness error
         if(set.add(node) == false)
           result = false;

         // Free block consistency error
         if (node.next != null && node.state != 1)
           result = false;

         count++;
         node = node.next;
       }
```

Figure 5.8: Example of simple file system data structure, abbreviated, consisting of a file allocation table (FAT) class and a DirectoryEntry class. Method repOk is a correctness condition that checks chain disjointness and free block consistency.

```

        // Size mismatch
        if (count != dir.size)
          result = false;

        dir = dir.forward;
      }

      return result;
    }
}
```

Figure 5.9: Continued repOk method from Figure 5.8 that checks for directory size mismatch.

Figure 5.8 shows the repOk method for the modified file system example. It checks the two constraints as follows: for each of the directory entries, the repOk starts by following the *first* field to get the first block in the FAT. Then it traverses the chain through the *next* field until it hits the end, the *null*. For each of the visited block in the chain it checks two things. First, it utilizes a set to check the block's uniqueness. If an attempt to add the block to the set fails, it means that the block has been visited before indicating a violation to the chain disjointness constraint. Second, it checks if the visited block's state is -2 (i.e., the *free* block). In such a chase it detects a free block consistency error.

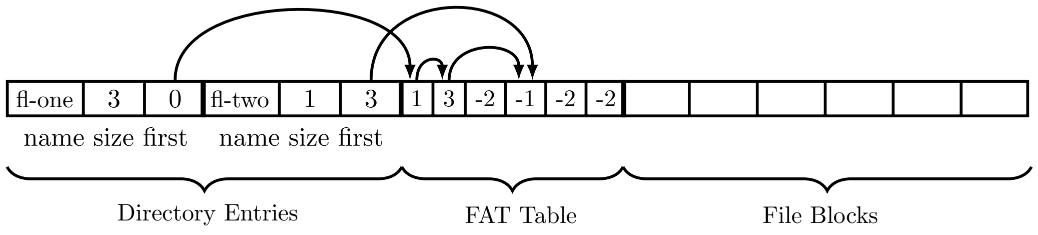


Figure 5.10: File system example with chain disjointness error. Files should not share blocks, but both the files share the fourth block.

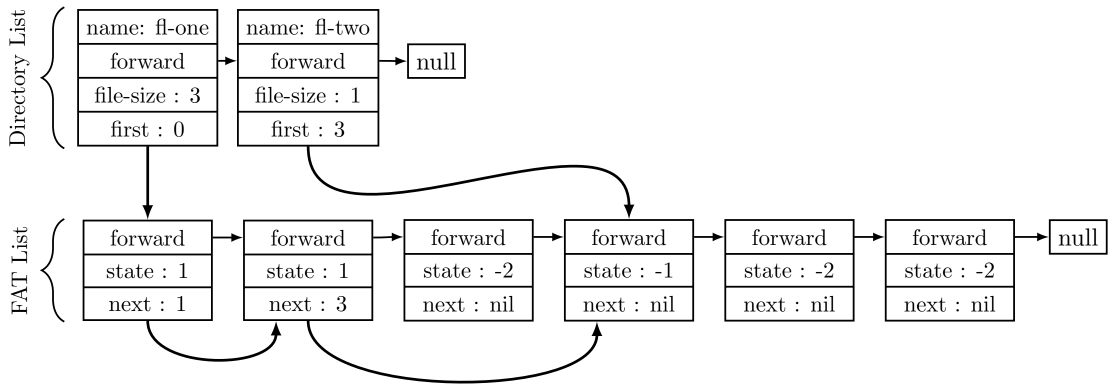


Figure 5.11: File system example with chain disjointness error. Both the files share the fourth block.

In our experiment, we manually induced errors to break the consistent state. For example, the equivalent Figures 5.10 and 5.11 shows an induced error that breaks the chain disjointness consistency. Files should not share blocks but both the files share the fourth block. In other examples we deliberately made the state of a used block to be free breaking the second consistency constraint.

In all our examples, Juzi suffered time-outs and failed to repair the corruptions. Whereas our approach took less than 5 attempts to repair. We further investigated Juzi's performance and found the reason behind it's time-outs. Because of its style dependency on the repOk method, Juzi was trying to fix the *next* field (which was not corrupted) in futile attempts. An updated repOk finally helped Juzi to repair the corruptions but still took more than 50 attempts.

Chapter 6

Evaluation of Tools and Compilers with Large Generated Random Benchmark Applications

This section and the following sections describe the experience with the *Random Utility Generator for pRogram Analysis and Testing (RUGRAT)* tool [83]. That is, we have conducted several experiments to explore the usefulness of RUGRAT for the evaluation and benchmarking of Java *pRogram Analysis and Testing (RAT)* algorithms and tools.

The RUGRAT project[1] is a joint work by several people from different institutions. The contributors are: Christoph Csallner and myself, Ishtiaque Hussain from the University of Texas at Arlington, Mark Grechanik and his students B.M. Mainul Hossain, Balamurugan Prabakaran, Nischit Rangapan and Arthi Vijayakumar at the University of Illinois at Chicago, Chen Fu (now at Microsoft) and Qing Xie from the Accenture Technology Labs, Sangmin Park from the Georgia Institute of Technology and Kunal Taneja (now at Accenture) from the North Carolina State University.

I collaborated with Mark's students in developing the algorithm and implementing the RUGRAT framework. However, since my major contribution was in the evaluation of the RUGRAT tool, this section focuses on the evaluation of the tool with Java source-to-bytecode compilers and RAT tools.

[1]RUGRAT: `https://sites.google.com/site/rugratproject/`

6.1 Background, Goal, Approach and Benefits of RUGRAT

In this section, at first we briefly describe how RUGRAT relates to the data structure repair. Then we discuss the background on the random program generation approach, how RUGRAT implements and leverages the approach by building on top of it with added goals. Finally we discuss the benefits of using RUGRAT in creating benchmark applications over hand-written ones.

6.1.1 Data Structure Repair and RUGRAT

Data structure repair approaches require subject data structures to work on. To evaluate performance and effectiveness of a repair approach and compare against other existing techniques, a developer needs to run her repair approach, along with other existing techniques, on different target data structures with varying complexities. Creating large number of different data structures with varying complexities and defining their correctness conditions can be quite challenging.

An automatic technique that can generate data structures and define their correctness conditions would be beneficial for a developer of a data structure repair approach. Ideally, random program generators should be able to automatically create applications that have desired data structure components. Based on the data structures used in the generated programs, these random program generators could also automatically define their correctness conditions. Although current implementation of RUGRAT does not have this feature, it can be considered as a first step in this research direction.

6.1.2 Stochastic Grammar Model

Consider that every program is an instance of the grammar of the language in which this program is written. Typically, grammars are used in compiler construction to write parsers that check the syntactic validity of a program and transform its source code into a parse tree [7]. An opposite use of the grammar is to generate branches of a parse tree for different production rules, where each rule is assigned the probability with which it is instantiated in a program. These grammars and parse trees are called *stochastic*, and they are widely used in natural language processing, speech recognition, information retrieval [35], and also in generating SQL statements for testing database engines [149]. RUGRAT uses a *stochastic grammar model* to generate large random object-oriented programs.

Random programs are constructed based on the stochastic grammar model, and the construction process can be described as follows. Starting with the top production rules of the grammar, each nonterminal is recursively replaced with its corresponding production rule. When more than one production rule is available to replace a non-terminal, a rule is randomly chosen based on the rules' probabilities. Terminals are replaced with randomly generated identifiers and values that preserve syntax rules of the given language. Termination conditions for this process of generating programs include the limit on the size of the program or selected complexity metrics.

In addition to the rules that are found in a typical context-free grammar of a programming language, RUGRAT takes into account additional rules and constraints that are imposed by the programming language specification. For example, a variable has to be defined before it can be used and a non-abstract class in an object-oriented program has to implement all abstract methods it inherits from its super-types. With such an enhanced stochastic grammar model it is ensured that the generated program is syntactically correct and compiles. The construction process can be fine-tuned by varying the ranges of different configuration parameter values and limiting the grammar to a subset of the production rules that are important for evaluating specific

RAT tools (e.g., recursion, use of arrays, or use of different data types can be turned off if a RAT approach does not address these).

6.1.3 RUGRAT's Configuration Options

To satisfy different requirements for generated programs, RUGRAT is highly configurable. Some of the important parameters include number of classes, number of methods per class, number of interfaces, number of methods per interface, maximum depth of the inheritance hierarchy, number of class fields, number of parameters per method, and recursion depth (if recursion is enabled). Most of the parameters have a lower and an upper limit. Moreover, many parameters are inter-dependent (e.g., there should be enough classes to populate an inheritance tree of a desired depth). Once these limits are defined, RUGRAT randomly chooses values from each range.

For each of these configuration parameters, we define a default range that seems reasonable based on empirical data [72, 178, 36]. For example, to determine the number of classes, we follow Zhang et al.'s observation that LOC is roughly 114 times the number of classes in a program [178] and set $classes = LOC/114$. To define the number of interfaces, we follow the observation of Collberg et al. [36], that each package in a program has roughly 12 classes and there is one interface per package. Hence we set $interfaces = LOC/(114*12) = LOC/1368$. Grechanik et al. [72] found that the average value for the 'maximum number of methods per interface' is 3.4, and thus we took ten times the average value and set the upper limit of the range to 34. Collberg et al. [36] found that 96% of the programs have less than 20 class fields, and 99% of the programs have less than 60 methods per class. We conformed to these observations and used these values as the upper bound for respective parameters. We used similar heuristics for other parameters, such as number of parameters per method and maximum inheritance depth. The RUGRAT tool website[2] has a complete list of the configuration parameters and their default values.

6.1.4 RUGRAT's Goal and Approach

RUGRAT addresses one main goal—to allow experimenters to automatically generate benchmark applications that have desired properties for evaluating RAT approaches and tools. RUGRAT is not a replacement of real-world applications for evaluating RAT approaches and tools. It is a tool that enables experimenters to quickly generate a large number of application benchmarks that have desired properties. The goal of RUGRAT is thus to supplement evaluations of RAT tools using real-world application benchmarks. In a way, RUGRAT can be viewed as a rapid prototyping tool for producing a set of benchmark applications for initial evaluation of RAT approaches and tools.

To achieve the goal, RUGRAT has to address several issues. First, generated programs must have a wide variety of language constructs that are important for evaluating RAT approaches and tools. Sample constructs include recursion, dynamic dispatch, and array manipulations using expressions that compute array indices that test the boundaries of RAT algorithms. Existing program generators often do not take into consideration such language constructs and do not add them to generated programs.

In addition, there is a requirement that generated programs should represent real-world programs using software metrics. This requirement is motivated by the needs of the potential RUGRAT users. That is, we expect that RAT tool developers and RAT tool users care most about the performance of RAT tools on real-world

[2]RUGRAT tool website: `https://sites.google.com/site/rugratproject/`

programs, since processing real-world programs is likely going to be the main use case of these RAT tools.

While stress-testing a RAT tool with unusual programs is also important, we expect RUGRAT users to care most about RAT tool performance on programs that are more mainstream. In this regard we expect RUGRAT users to be similar to RAT tool users. RAT tool users have frequently complained about RAT tools generating input values for the analyzed program that appear exotic and do not represent expected program behavior as well as about warnings on bugs that cannot occur or can only occur in very rare situations [119, 136, 180, 161].

As a consequence, RUGRAT allows to tune the default parameters such that generated programs are as similar as possible to what one would consider a normal hand-written program. RUGRAT implements this issue by varying the probabilities that are assigned to different production rules of the language grammar. RUGRAT users can diverge from the default parameters to produce more exotic kinds of programs. In our evaluation we explore an example use of RUGRAT with non-standard parameters.

6.1.5 Benefits of Using RUGRAT

RUGRAT scales to generating programs that at the same time are large, have complex properties, and are similar to hand-written programs [83]. While RUGRAT-generated programs are similar to hand-written programs, it provides multiple benefits over benchmarking with hand-written applications.

First, using RUGRAT one can easily generate a large variety of random programs. Such a large set of programs can complement existing suites of hand-written benchmark programs, which are often relatively small sets of programs. For example, the well-known Siemens suite [51] consists of a few small programs and could benefit from a large set of complimentary applications that cover a range of program configuration points.

Second, RUGRAT-generated programs have a designated entry point or main function. In contrast, hand-written programs such as libraries typically lack such a clear entry point, which forces many RAT tool developers to write test harnesses for RAT tool evaluation [90]. Each RUGRAT-generated program has a dedicated main method that can be used to start the program directly and does not require a test harness.

Third, RUGRAT can scale down from realistic applications to toy applications that only contain a specified set of language features. This down-scaling is useful during RAT tool development. That is, at an early RAT tool development stage, a RAT tool may only be able to handle a few programming language features. At this point a RAT tool developer may still want to test her RAT tool on large applications. However hand-written applications often use multiple language features and it may be hard to find hand-written applications that only use a given set of language features, especially when looking for a variety of larger applications. For example, randomly sampling current open-source Java programs will likely yield programs that are using newer language features like generics, etc., which makes it impossible to use it to compare all versions of the JDK compilers.

Fourth, as a special case of down-scaling, RUGRAT is useful for generating programs that have no external dependencies. In contrast, most realistic hand-written applications have external dependencies such as on external libraries. Such external dependencies often complicate RAT tool operation, even for industrial-strength RAT tools. For example, in a recent study of why the industrial-strength Pex dynamic symbolic execution (DSE) tool achieved less than perfect branch coverage, more than a quarter of the missed branches were due to calls to external libraries such as native code [171]. To analyze such applications, DSE tool developers sometimes have to resort to writing mock versions or models of the external dependencies, which is tedious and laborious [110]. In contrast, programs generated by RUGRAT do not have external dependencies, so DSE tool developers can easily compare the scalability of their tools on benchmark applications.

Fifth, since RUGRAT-generated applications do not have external dependencies it is very easy to compile, install, execute, and test RUGRAT-generated applications. In contrast, hand-written programs are often difficult to install and execute. Additional systems such as databases, servers, and communication infrastructure may need to be installed and configured for such programs. Moreover, before a realistic hand-written application can be tested, external dependencies have to be resolved. These can be tedious. For example, it took me more than two hours to figure out all the necessary external libraries, look up, download them and finally resolve the build path before I could compile the open-source Java project *fmj*[3] from SourceForge with different Java compilers.

A survey on evaluating static analysis tools and benchmarks showed that most user-reported failures in software repositories are false failures, i.e., failures that will not be fixed as they do not concern the code [165]. Indeed, the false failures are mostly installation failures, which may be caused by poor documentation and difficult deployment procedures. RUGRAT users avoid this potential pitfall as RUGRAT-generated applications do not require any installation or configuration and can be compiled and executed immediately.

Given the difficulties inherent in using hand-written programs for benchmarking, it is maybe not surprising that existing comparisons of RAT tools have mainly focused on using small to medium-sized subject programs for benchmarking. That is,

[3]fmj website: `http://sourceforge.net/projects/fmj/`

the empirical comparisons we are aware of are limited to subject sizes of small test cases [46, 164, 163], less than 150k LOC [136, 161, 165], or less than 500k LOC [10]. With RUGRAT it is easy to generate subject applications that contain several million lines of code (see Section 6.2.2).

Finally, RUGRAT enables more experiments at lower cost by providing, on demand, many high-quality programs in short time. When evaluating a RAT tool with hand-written programs, if the RAT developer wants to evaluate the tool with more programs, she needs to explore code repositories with specific requirements, which can be time-consuming. However, with RUGRAT she can generate such programs automatically in a short amount of time (see the RUGRAT resource consumption evaluation in Section 6.2.2), by specifying such requirements as parameters to RUGRAT.

6.2 Overview of Experiments with RUGRAT-generated Applications as RAT Tool Benchmarks

At a high level, a prospective Java RAT tool user is interested in comparing and benchmarking RAT tools and may be asking several questions. For example, how do Java RAT tools behave when running on applications of different sizes? This question is especially relevant if experiments published to date do not cover the kind of input application sizes that are relevant for the prospective RAT tool user [136, 161, 46, 165, 45, 164, 163, 10].

Specifically, before acquiring a particular Java RAT tool, a prospective RAT tool user may be wondering if a given Java RAT tool will break down for large input applications, while a competitor RAT tool may scale to such applications. Secondary questions are how for different input application size categories the time and memory requirements of various RAT tools compare against each other.

When using a random benchmark program generator such as RUGRAT to explore such questions, it is important to determine if the RUGRAT-generated applications resemble real-world applications. Only if generated applications are similar to real-world applications we can attempt to extrapolate results obtained on benchmark applications to how RAT tools will behave on actual user applications.[4]

Before using a new tool such as RUGRAT, a potential user may want to know how expensive RUGRAT is in terms of computational resources. Part of the appeal of RUGRAT is that RUGRAT can save the user time, as installing handwritten programs can be very time intensive. Then a natural question is how much time it takes RUGRAT to generate programs.

We thus explore the following concrete research questions.

- *RQ0.* How many computational resources (i.e., execution time, main memory and disk space) does the RUGRAT random benchmark program generator require?
- *RQ1.* Can RUGRAT-generated applications be used for focused benchmarking of existing Java RAT tools—i.e., compilers and static and dynamic program analysis tools?
  - *RQ1a.* Can RUGRAT-generated applications be used for benchmarking the execution time and memory requirements of existing Java source to bytecode compilers?
  - *RQ1b.* Can RUGRAT-generated applications be used for benchmarking the execution time and amount of output of existing static and dynamic Java program analysis tools?

[4]Another interesting question is whether RUGRAT-generated applications provides more benefits than real-world programs. We list such benefits in Section 6.1.5, and our research questions deal with comparisons and the effectiveness of RUGRAT on RAT tools.

- *RQ2.* Can RUGRAT-generated applications find defects in program analysis tools?

Recall that the goal of RUGRAT is not to supplant all other ways of RAT tool benchmarking (Section 6.1.4). Instead, RUGRAT aims at enabling a focused benchmarking of specific RAT tool features. A complete benchmarking of existing Java RAT tools is therefore outside the scope of this work. A complete benchmarking would also address important issues such as RAT tool installation and maintenance requirements, the precision and recall of the RAT tool outputs, and ease of use. We leave such issues for future work. Instead, in the following we focus on selected features of RAT tools that are easy to measure, such as the quantity of RAT tool outputs as well as RAT tool memory consumption and execution time.

6.2.1 RUGRAT Generated Applications

For each experiment we used RUGRAT to generate applications of various sizes, ranging from some 10kLOC to 5MLOC. Specifically, we picked 7 target application sizes given in non-comment, non-blank lines of code (LOC), i.e., 10k, 50k, 100k, 500k, 1M, 2.5M, and 5M and generated several applications for each target LOC size. (Due to implementation limitations the actual LOC of an AUT may deviate slightly from the target value.)

In the context of benchmarking RAT tools, we refer to RUGRAT-generated applications also as *applications under test* or just AUTs. For these experiments we generated only single-threaded applications. Extending RUGRAT to generate multi-threaded applications is a subject of future work.

Table 6.1: RUGRAT execution time, main memory, and disk space consumption when generating AUTs of various sizes on a standard desktop computer. Later experiments in Section 6.4 were performed on a more powerful machine. AUT sizes (10k, 50k, etc.) are given in LOC. All aborts are due to RUGRAT running out of main memory.

Resource		10k	50k	100k	500k	1M	2.5M	5M
Time [s]	Avg.	7	26	73	440	794	5,934	3,259
	Best	2	6	19	113	113	383	227
	Worst	12	49	230	798	1,867	20,696	6,256
Mem [MB]	Avg.	20	48	105	415	575	847	1,327
	Best	16	23	42	174	286	361	866
	Worst	25	81	341	826	997	1,060	1,498
Disk [MB]	Avg.	1	5	12	55	72	173	262
	Best	<.4	2	4	23	40	110	137
	Worst	2	12	47	109	130	315	315
Abort		0	0	0	0	0	1	5

6.2.2 RQ0: RUGRAT Resource Consumption

In this section we try to answer the research question, R0, namely, how many computational resources (i.e., execution time, main memory and disk space) does the RUGRAT random benchmark program generator require?

To capture RUGRAT's behavior on a standard desktop computer, we performed most experiments on a 32-bit Windows 7 OS running on a 2.5GHz AMD dual core processor with 4GB RAM. This machine is a few years old, but it is well suited to explore the scalability limitations of the current RUGRAT implementation. With these limitations established we switched to a more powerful machine for subsequent experiments that benchmark existing RAT tools on RUGRAT-generated applications. The more powerful machine has more main memory with 32GB RAM and is running a 64-bit Windows XP OS on a 2.33GHz Xeon processor (see Section 6.4).

Table 6.1 summarizes the resource consumption of our RUGRAT prototype. For each column or LOC category (10k, 50k, etc.), we used RUGRAT and its default parameter ranges to generate 10 random programs. For 2.5M LOC we generated 9 programs, as one attempt aborted with an out of memory exception. Similarly, for 5M LOC five attempts were aborted with out of memory exceptions. This data indicates that the current RUGRAT implementation scales to about 2.5MLOC on a standard desktop computer (a 2.5GHz machine with 4GB RAM).

For each column or group of 10 RUGRAT executions, the table shows the average, best, and worst consumption of time, main memory[5], and disk space of that group. Lower numbers are better as they indicate lower resource consumption. Disk space consumption is the space that is required to store a generated application.

In our experiments the size of the generated applications grew about linearly with the target LOC size. The data further suggests that the average RUGRAT execution time currently does no scale linearly with respect to LOC. In general, the RUGRAT tool implementation can be used on a standard desktop computer but it is currently not optimized for either speed or (main) memory consumption and we expect that these aspects can be improved with more engineering work.

6.3 Experience with RUGRAT-Generated Applications as Benchmarks for Java Source to Bytecode Compilers

In this section we describe our experiments with the RUGRAT-generated AUTs and several versions of a popular Java source to bytecode compiler (RQ1a).

6.3.1 Experimental Setup

We obtained 8 versions of the Java development kit (JDK) from the Oracle Java Archive[6], i.e., versions 1.2.1, 1.2.2, 1.3.0, 1.3.1, 1.4.0, 1.5.0, 1.6.0, and 1.7.0. Each

[5]We used the Windows 7 default performance monitor PerfMon to log memory usage.

[6]http://www.oracle.com/technetwork/java/archive-139210.html

downloaded development kit contains a default Java source to bytecode compiler. These 8 compilers are listed in Table 6.2 by their JDK version jY.Z, i.e., as j2.1, j2.2, etc., omitting the common top-level version 1 identifier.

Since some of the older Java development kits were only available in 32-bit versions, we conducted all experiments on our standard desktop computer, a 32-bit Windows 7 OS running on a 2.5GHz AMD dual core processor with 4GB RAM. As on this machine our current RUGRAT random benchmark program generation prototype does not scale to generating 5M LOC programs (Section 6.2.2) we used a more powerful machine to run RUGRAT and supply us with a total of 70 subject programs, 10 in each size category.

For the experiments we configured each compiler to use the maximum amount of memory (heap space) that was possible on our machine for that particular compiler. As a side note, the compiler options for setting this maximum amount of memory changed between compiler versions and the corresponding maximum amount that could be set on the machine also fluctuated between compiler versions, i.e., between 1.15 GB (for j3.0) and 2 GB (for j2.1 and j2.2). The remaining compilers accepted a maximum of either 1.6 GB (j4.0) or 1.5 GB.

6.3.2 RQ1a: Comparing Java Source to Bytecode Compilers on RUGRAT-Generated Benchmark Applications

Table 6.2 shows the absolute execution time and main memory consumption of the subject compilers when compiling our 70 RUGRAT-generated subject programs. For both execution time and memory consumption the table shows average ($\ominus$), maximum ($\top$), and minimum ($\perp$) measurements. Lower values are better, as they indicate lower resource consumption.

Figure 6.1 plots these values to show the trends in these data. Each plot in Figure 6.1 shows the compiler execution time and memory consumption for all compilers on the 10 subject programs of an AUT size category. The plot uses box and whiskers to show minimum, lower quartile, median, upper quartile, and maximum values.

Table 6.2: Benchmarking the standard JDK Java source to bytecode compilers. The table shows average (⊖), maximum (⊤), and minimum (⊥) absolute execution time (t) and main memory consumption (m) of the subject compilers. Lower values are better, as they indicate lower resource consumption. All compilers compiled all 10k, 50k, and 100k AUTs. For the other AUT size categories dnc (did not compile) is the number of subjects a compiler could not compile.

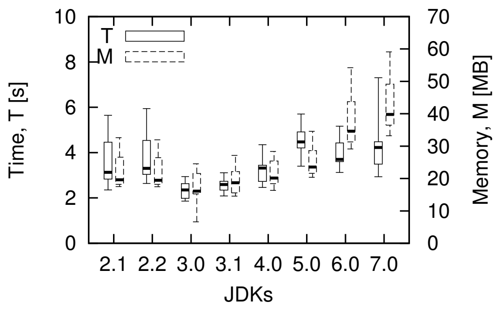
LOC			j2.1 [s,MB]	j2.2 [s,MB]	j3.0 [s,MB]	j3.1 [s,MB]	j4.0 [s,MB]	j5.0 [s,MB]	j6.0 [s,MB]	j7.0 [s,MB]
10k	t	⊤	6	6	3	3	4	6	5	7
		⊖	4	4	2	3	3	4	4	4
		⊥	2	3	2	2	2	3	3	3
	m	⊤	33	32	25	27	28	35	54	59
		⊖	22	22	17	19	21	25	38	43
		⊥	18	17	7	15	16	20	29	33
50k	t	⊤	64	64	14	14	18	21	19	26
		⊖	18	19	6	7	8	10	10	12
		⊥	7	8	4	4	5	6	5	8
	m	⊤	191	187	143	144	156	175	292	318
		⊖	76	76	61	62	68	72	114	121
		⊥	45	44	38	38	40	43	68	69
100k	t	⊤	110	112	31	33	37	36	41	45
		⊖	46	47	15	16	18	20	21	25
		⊥	12	12	5	6	7	9	9	11
	m	⊤	385	386	323	324	365	423	648	667
		⊖	166	166	131	131	147	152	247	258
		⊥	73	73	61	60	69	67	107	111
500k	t	⊤	125	350	166	197	194	205	247	365
		⊖	108	179	81	98	106	112	110	138
		⊥	76	82	31	31	35	35	34	43
	m	⊤	398	519	969	1,273	1,444	1,432	1,483	1,510
		⊖	333	406	510	593	665	676	948	987
		⊥	249	249	201	201	226	216	353	376
		dnc	7	5	1	0	0	0	0	0

Table 6.3: Continued from Table 6.2

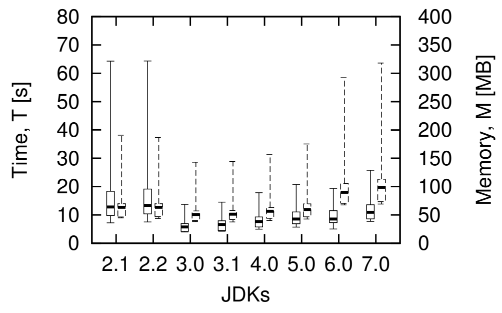
LOC			j2.1 [s,MB]	j2.2 [s,MB]	j3.0 [s,MB]	j3.1 [s,MB]	j4.0 [s,MB]	j5.0 [s,MB]	j6.0 [s,MB]	j7.0 [s,MB]
1M	t	$\top$	237	236	204	481	466	477	367	591
		$\ominus$	218	220	129	177	183	194	159	208
		$\bot$	188	195	80	81	90	104	62	102
	m	$\top$	519	519	953	1,430	1,591	1,450	1,493	1,504
		$\ominus$	519	518	668	761	856	813	1,182	1,212
		$\bot$	518	518	437	445	507	482	737	776
		dnc	7	7	2	1	0	0	0	0
2.5M	t	$\top$	n/a	n/a	n/a	615	729	802	583	1,021
		$\ominus$	n/a	n/a	587	461	566	567	302	562
		$\bot$	n/a	n/a	n/a	288	302	302	125	181
	m	$\top$	n/a	n/a	n/a	1,532	1,631	1,522	1,524	1,505
		$\ominus$	n/a	n/a	930	1,355	1,484	1,374	1,487	1,486
		$\bot$	n/a	n/a	n/a	961	1,077	1,027	1,364	1,392
		dnc	10	10	9	5	5	4	4	4
5M	t	$\top$	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a
		$\ominus$	n/a	n/a	n/a	919	1,064	1,031	214	462
		$\bot$	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a
	m	$\top$	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a
		$\ominus$	n/a	n/a	n/a	1,492	1,569	1,469	1,496	1,405
		$\bot$	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a
		dnc	10	10	10	9	9	9	9	9

From the results we can make several observations. First, maybe expected, the newer the compiler the more likely it can compile more of the generated applications, including the very large ones. Each case in which a compiler failed to compile a subject is noted in Table 6.2 as dnc ("did not compile"). In our experiments each dnc case was caused by the compiler running out of available (heap) memory. In other words, the older the compiler, the more likely it ran out of memory when attempting to compile some of the largest applications in our sample.

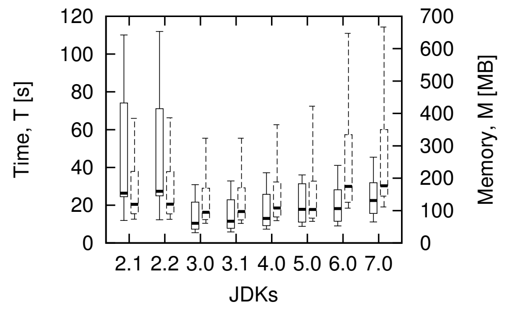
Specifically, the newest compilers, j5.0, j6.0, and j7.0, could compile the most applications and, on the sample size of 70 generated applications, failed to compile only 13 applications. The next older compiler, j4.0, failed to compile 14 applications;



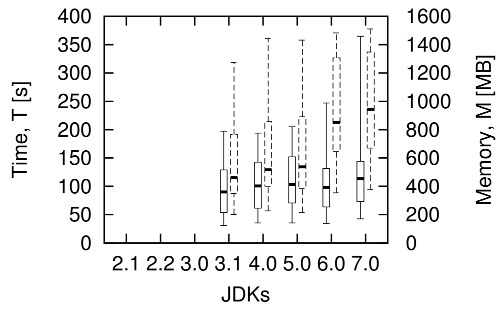
(a) 10k LOC.



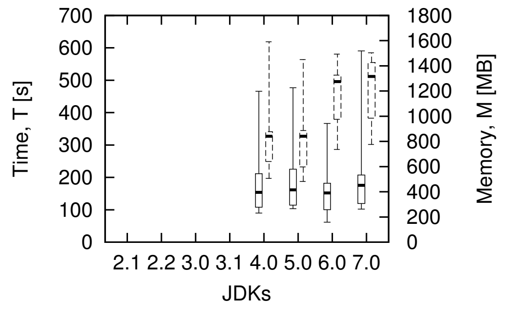
(b) 50k LOC.



(c) 100k LOC.



(d) 500k LOC.



(e) 1M LOC.

Figure 6.1: Comparing time and memory consumption of the subject compilers for different program size AUTs. The x-axis represents the compilers; the left y-axis represents compiler runtime in seconds and the right y-axis represents compiler main memory usage in megabytes. We did not plot a compiler in an AUT size category if the compiler could not compile all AUTs in that category. No compiler could compile all 2.5M or 5M LOC AUTs.

the next older compiler, j3.1, failed to compile 15; j3.0 failed to compile 22; j2.2 failed to compile 32; and j2.1 failed to compile 34. Table 6.2 shows that the applications that could not be compiled are mostly in the 2.5M and 5M LOC categories.

While for this experiment we ran RUGRAT with its default parameters, the results of the experiment, somewhat surprisingly, resemble the results of stress-testing. That is, although all generated programs use common combinations of language features, the tested compilers could not compile all programs. This effect increased with the size of the generated programs and was most pronounced for the largest generated programs, i.e., in the 5M LOC category.

Second, on the subjects up to and including 2.5M LOC that each compiler could compile, the newer compilers had the highest average memory consumption. Specifically, j7.0, had the highest average memory consumption, followed closely by the next older compiler, j6.0. On individual subjects the older compilers j4.0 and j3.1 had higher memory consumption than their newer peers, but on average these older compilers consumed less memory.

Third, j2.2 was the slowest compiler with the highest average compile time for most AUT size categories. The exceptions are the smallest and possibly the largest AUTs, since for the largest AUTs (2.5M) this compiler did not compile a single subject. A close second slowest was the predecessor compiler j2.1, with the same caveats for the smallest and largest AUT sizes.

Finally and maybe somewhat surprising, for the small and medium sized AUTs of up to 100k LOC, the fastest compiler was j3.0. For these AUT size categories, the average compile speed of j3.0 was between 50 and 60% of the newer j7.0 baseline compiler. This trend continues to larger AUTs of up to 1M LOC if we only consider the applications the respective compilers could compile. For these larger AUTs the average compile time of j3.0 fluctuated between 70 and 80% of the j7.0 baseline compiler.

However, for the largest AUTs of 2.5 MLOC, j3.0 failed to compile 9 of 10 AUTs and for the one AUT it did compile j3.0 needs more than three times the compile time than the j7.0 baseline compiler.

6.4 Experience with RUGRAT-Generated Applications as Benchmarks for Static and Dynamic Java Program Analysis Tools

We conducted two experiments to explore the research questions related to RAT tools (RQ2b and RQ3). In the first experiment, we used RUGRAT to generate applications under test (AUTs) using RUGRAT's default parameter ranges, which model the properties of typical third-party applications.

In the second experiment we widened the parameter ranges to also allow for values that are only found rarely in third-party applications. While rare, these values are still possible according to the empirical data described in Section 6.1.3. This second experiment simulates a stress-testing of RAT tools.

6.4.1 Experimental Setup

For the first experiment we used RUGRAT to generate 10 random AUTs per LOC value, yielding 70 AUTs. For the second experiment we just generated a single AUT per LOC value, yielding 7 AUTs. We ran all experiments on a HotSpot 1.6.0_24 JVM on Windows XP on a 2.33GHz 64-bit Xeon processor with 32GB RAM.

On each of the 77 generated AUTs, we applied five Java program analysis tools: four static analysis tools, Checkstyle, FindBugs, JLint, and PMD and one dynamic analysis tool, Randoop[7]. These tools apply different techniques in analyzing

[7]Checkstyle version 5.4: http://checkstyle.sourceforge.net/

FindBugs version 1.3.9: http://findbugs.sourceforge.net/

JLint version 2.3: http://artho.com/jlint programs and produce output in the form of various kinds of warnings. Such program analysis tools are typically highly configurable. To approximate the behavior of the tools under different configurations, for each tool we set a minimum and a maximum configuration. In the minimum configuration, we try to evoke a minimum amount of tool features; in the maximum configuration, we try to invoke all tool features.

In the following we briefly summarize the key features of the five Java RAT tools and describe how we configured them for our minimum and maximum configurations.

6.4.1.1 Checkstyle

Checkstyle [63] works on Java source code, is easy to expand, and supports custom bug patterns called ‘checks’. Checkstyle provides a standard ‘check’ that has 64 modules to check the Sun coding conventions which we used for the minimum-effort level experiments. For the maximum effort level experiments, we enabled 128 checking modules (which include the 64 in the standard check). Example modules are FileLength, MethodName, ConstantName, Indentation, and ParameterNumber.

6.4.1.2 FindBugs

FindBugs [80] applies syntactic bug patterns and dataflow-analysis on AUT bytecode to find bugs. It supports custom patterns and is easily expandable. For the configurations, we used two flags (‘effort’ and ‘reportLevel’). For the maximum configuration, we set ‘effort’ to maximum and ‘reportLevel’ to ‘low’, which yields all bugs found during analysis. For the minimum configuration, we set ‘effort’ to minimum and ‘reportLevel’ to ‘high’, to restrict reporting to high priority bugs.

PMD version 4.2.5: http://pmd.sourceforge.net

Randoop version 1.3.2: http://code.google.com/p/randoop

6.4.1.3 JLint

Like FindBugs, JLint [9] applies syntactic bug patterns and dataflow analysis on AUT bytecode, but it is not easy to expand [136]. JLint has patterns for detecting thread synchronization bugs, which we disabled in the minimum configuration. For the maximum configuration, we enable all patterns.

6.4.1.4 PMD

PMD [37] applies syntactic bug patterns on AUT source code. It supports custom bug patterns (called ruleset) and is easily expandable. For the minimum configuration, we enabled only ruleset 'basic'. For the maximum configuration, we also enabled rulesets braces, clone, codesize, controversial, coupling, design, imports, naming, strictexception, strings, typersolution, and unusedcode. Descriptions of these ruleset are in the PMD manual.

6.4.1.5 Randoop

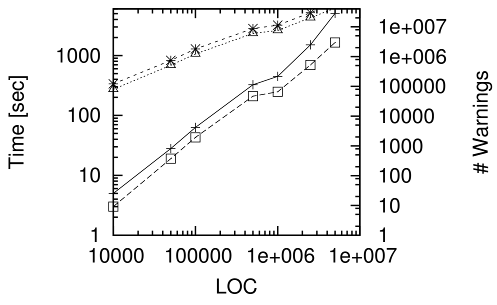
Randoop [123] applies feedback-directed random test case generation [124] to AUT bytecode to deduce program behavior and create assertions to detect bugs. Randoop does not have any flags or configuration options we could set for our configurations. By default, it runs either for 100 seconds or until 100,000,000 tests are generated. We limit the timing to 100 seconds for the minimum configuration and 2,400 seconds (40 minutes) for the maximum configuration.

6.4.2 RQ1b: Comparing RAT Tools on RUGRAT-Generated Benchmark AUTs

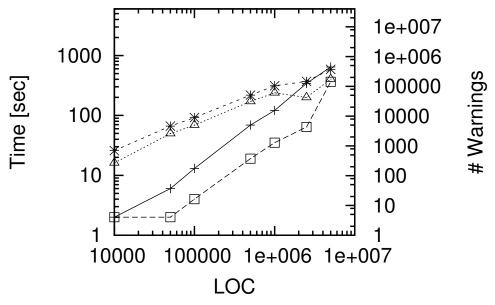
We performed 770 experiments by invoking 5 RAT tools in 2 configurations each on 77 generated AUTs. The two configurations are the minimum and the maximum

Table 6.4: Benchmarking popular static and dynamic Java RAT tools. Ef = effort level defined in Section 6.4.1; W = number of warnings (outputs) a tool generated, T = tool execution time.

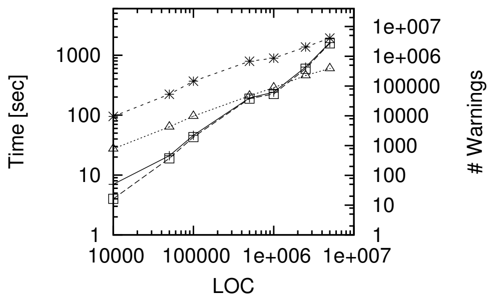
AUT	Ef	Tool	W_{min}	W_{max}	W_{mean}	T_{min}	T_{max}	T_{mean}
10k	Min	Checkstyle	37,506	144,799	80,008	2	6	3
		FindBugs	111	1,218	499	16	64	25
		JLint	76	651	269	1	4	2
		PMD	192	1,636	776	1	9	4
		Randoop	0	28	8	101	107	103
	Max	Checkstyle	58,354	216,221	120,970	3	9	5
		FindBugs	778	7,889	3,217	17	35	25
		JLint	366	1,101	694	1	5	2
		PMD	3,485	17,901	9,550	4	11	7
		Randoop	0	795	152	2,401	2,404	2,403
50k	Min	Checkstyle	276,984	1,351,638	513,644	10	47	19
		FindBugs	1,080	7,102	2,866	45	146	68
		JLint	211	11,241	2,589	1	5	2
		PMD	3,149	7,631	4,277	5	60	19
		Randoop	0	4	1	102	105	103
	Max	Checkstyle	417,065	1,946,076	749,994	14	71	28
		FindBugs	4,681	51,335	16,097	54	189	86
		JLint	2,468	14,047	4,606	2	11	6
		PMD	23,243	156,035	53,334	9	61	21
		Randoop	0	75	18	2,402	2,410	2,406
100k	Min	Checkstyle	507,352	3,317,995	1,229,880	21	117	43
		FindBugs	1,771	10,026	5,823	77	243	136
		JLint	310	13,737	5,023	1	11	4
		PMD	5,767	14,530	9,749	11	112	43
		Randoop	0	25	4	102	109	105
	Max	Checkstyle	757,294	4,637,270	1,809,312	30	158	63
		FindBugs	9,067	130,142	48,889	98	326	173
		JLint	881	17,486	8,936	3	19	13
		PMD	44,214	403,141	144,826	16	112	46
		Randoop	0	204	33	2,404	2,411	2,406
500k	Min	Checkstyle	1,798,495	13,252,753	5,998,586	62	486	209
		FindBugs	5,172	70,566	30,251	255	1,101	640
		JLint	2,522	123,133	30,967	4	59	19
		PMD	19,105	85,612	46,662	32	393	188
		Randoop	0	0	0	104	134	113
	Max	Checkstyle	2,797,354	19,093,653	8,740,053	97	779	325
		FindBugs	35,965	622,225	223,451	311	1,535	794
		JLint	13,467	145,244	50,548	12	131	69
		PMD	171,052	1,713,544	685,026	41	399	194
		Randoop	0	17	2	2,406	2,438	2,415



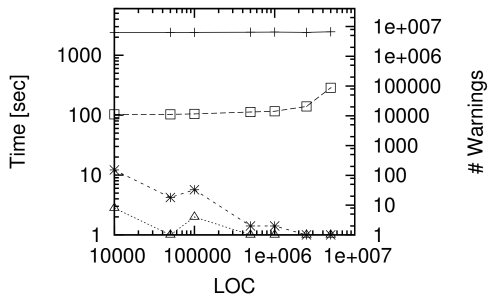
(a) Checkstyle.



(b) Jlint.



(c) PMD.



(d) Randoop.

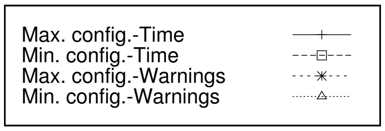
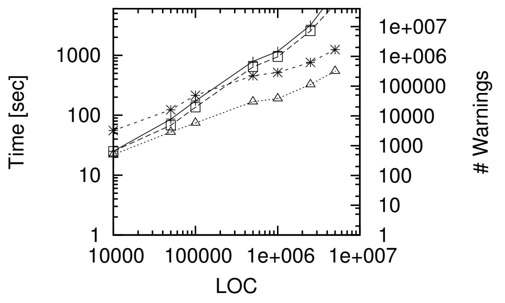
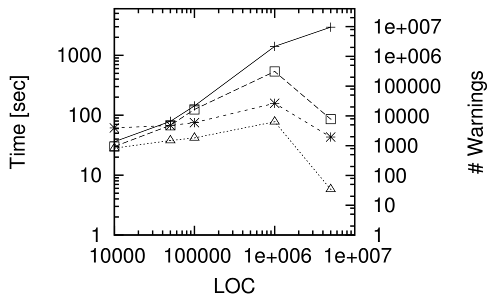


Figure 6.2: (Continued in Figure 6.3) Comparing static and dynamic Java program analysis tools on RUGRAT-generated programs. Each data point is the average of 10 AUTs from RUGRAT's default parameter range (experiment 1), except for Figure 6.3(b), which shows a single data point each from a wider range of configuration parameters (experiment 2); x-axis = LOC (log scale); left y-axis = RAT tool runtime; right y-axis = RAT tool warnings (log scale); MaxA/MinA = average A in maximum/minimum RAT tool configuration where A is either RAT tool time or number of RAT tool warnings. For static analysis tools both the average execution time and the average number of warnings mostly increased with program size (6.2(a)–6.3(a)). FindBugs was an exception when using a wider parameter range (6.3(b)). The dynamic analysis tool Randoop also behaved differently (6.2(d)).



(a) FindBugs.



(b) FindBugs skipping some classes.

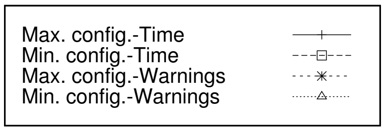


Figure 6.3: Continued from Figure 6.2

RAT tool effort configurations described in Section 6.4.1. For each experiment we captured each tool's execution time and the number of warnings generated by the tool.

Tables 6.4 and 6.5 summarize the experimental results for the bulk of the experiments, i.e., the 700 experiments on the default parameter ranges. For each RAT tool, program size category, and both the minimum and the maximum RAT tool effort configurations, these tables give the minimum, maximum, and average RAT tool runtime and number of warnings produced by a RAT tool.

For space reasons we omit the results of the remaining 70 experiments and instead plot highlights of both sets of experiments in Figure 6.2 and Figure 6.3. Figure 6.2 shows the average execution time and average number of warnings for each program size category for both the minimum and the maximum RAT tool effort configuration of the experiments on the default parameter ranges. Figure 6.3 shows these measurements for the relaxed parameter range for the FindBugs experiments.

From the results we can make several observations. First, as one would expect, for static analysis tools both the average execution time and the average number of warnings increased with the program size (LOC). This was true for both the minimum and the maximum effort category. The one exception to this observation is the data for JLint in the minimum effort configuration. There the average number of warnings decreases from 1M LOC to 2.5M LOC, while the average number of warnings for 5M LOC is again higher than for both 1 and 2.5M LOC.

Second, Checkstyle produced by far the most warnings among all the RAT tools (Figure 6.2(a)). This is true for both effort categories and across all AUT sizes. The difference to the second most producing RAT tool was one or two orders of magnitude, across both effort levels and all AUT sizes. The second highest average number of warnings was produced by the PMD tool, consistently in both effort categories and all AUT sizes (Figure 6.2(c)).

Third, the dynamic program analysis tool Randoop consistently produced the lowest numbers of warnings across both effort categories and all AUT sizes (Figure 6.2(d)). Moreover, Randoop differed from the other RAT tools in that it produced fewer warnings with increasing AUT LOC sizes. Since this result is counter-intuitive we examine it more closely in Section 6.4.3.

Fourth, most tools had vastly different average runtimes between their minimum and maximum configuration. PMD was an exception and had very similar minimum and maximum configuration runtimes (Figure 6.2(c)).

Finally, JLint had the lowest execution time among all RAT tools in both effort categories and all AUT sizes (Figure 6.2(b)). The only exception was the largest AUT size of 5M LOC for which Randoop had a lower average execution time. However this caveat may also be related to the counter-intuitive behavior of Randoop discussed in Section 6.4.3.

6.4.3 RQ2: RUGRAT Found RAT Bugs/Issues

As a by-product of benchmarking, the RUGRAT-generated programs let us independently rediscover several issues in RAT tools, i.e., in FindBugs and in Randoop. While not dramatic, these results demonstrate the potential usefulness of RUGRAT for testing and debugging.

6.4.3.1 FindBugs

FindBugs may skip classes and miss bugs. I.e., in the second experiment, which used wider parameter ranges simulating stress-testing, we encountered the situation depicted in Figure 6.3(b), where FindBugs did not show its usual execution time and warning behavior. Instead, it terminated quickly and reported only few warnings. Further investigation revealed that FindBugs has two limitations, which cause it to skip some code. Specifically, if a class has more than 1,000 methods or is larger than 1MB, FindBugs declares it to be too large and skips it. In the generated AUT, the majorities of classfiles were larger than 1MB. FindBugs thus skipped almost the entire AUT and terminated quickly, reporting few warnings. FindBugs has no configuration option to prevent such skipping. We confirmed with the tool authors that the recommend solution is to instead modify the FindBugs source code.

One may argue that such limitations only affect analysis of generated programs. However, we have found real (manually written as well as generated but then manually edited) applications on SourceForge that have such large classes, including Apache Derby, DoctorJ, Drools, and OpenJDK. Reducing the number of methods for some of the applications caused FindBugs to report warnings where it was previously skipping the analysis.

The FindBugs results are also an example of the influence of the RUGRAT parameters. While FindBugs skipped classes and thus produced few warnings in the second experiment with non-standard RUGRAT parameters (Figure 6.3(b)), FindBugs did not exhibit this erratic behavior in the first experiment, which uses RUGRAT with its default parameter settings (Figure 6.3(a)).

6.4.3.2 Randoop

While the other analysis tools generated more warnings for larger programs, Randoop, surprisingly, did the opposite; i.e., the larger the programs the fewer warnings Randoop generated (Figure 6.2(d)). We verified this behavior in a separate experiment, in which we increased the time allotted to Randoop's execution from 40 minutes to up to 8 hours, which would mirror an overnight run as part of an automated build and integration system. Doing so did not change the average number of warnings produced by Randoop, and therefore yields the same plot as Figure 6.2(d).

Increasing the runtime to up to 8 hours also led us to independently discover another issue with Randoop. This issue has been reported previously as Issue 14 in Randoop's issue tracking system[1]. Specifically, in the test generation phase, if no test is generated after 10 seconds of the last generated test, Randoop terminates without writing any tests, not even the last generated test.

A third issue we discovered is that for larger programs, Randoop does not terminate after 100 seconds as it was supposed to in the default setting (our minimum configuration).

[1]http://code.google.com/p/randoop/issues/detail?id=14

Table 6.5: Continued from Table 6.4.

AUT	Ef	Tool	W_{min}	W_{max}	W_{mean}	T_{min}	T_{max}	T_{mean}
1M	Min	Checkst.	4,652,104	14,226,538	7,553,695	144	445	248
		FindBugs	9,920	72,593	38,263	597	1,794	945
		JLint	1,880	153,682	60,802	7	171	35
		PMD	25,860	165,267	88,084	42	504	226
		Randoop	0	2	0	105	132	116
	Max	Checkst.	6,663,438	20,924,135	11,369,848	249	879	449
		FindBugs	41,185	560,864	287,923	733	2,009	1,166
		JLint	27,850	191,993	104,567	72	217	121
		PMD	305,925	1,738,556	863,091	90	504	241
		Randoop	0	17	2	2,406	2,543	2,429
2.5M	Min	Checkst.	8,980,257	43,034,144	20,266,502	394	1,304	695
		FindBugs	20,455	299,848	113,640	1,221	5,114	2,540
		JLint	2,596	163,099	42,259	14	167	64
		PMD	25,754	370,177	226,312	93	1,569	593
		Randoop	0	0	0	107	219	140
	Max	Checkst.	13,188,717	59,743,379	29,621,014	716	3,237	1,506
		FindBugs	71,731	1,417,767	628,937	2,006	5,834	3,173
		JLint	52,112	261,345	146,593	147	607	342
		PMD	268,086	4,689,904	2,056,934	213	1,560	631
		Randoop	0	0	0	2,187	2,497	2,413
5M	Min	Checkst.	13,252,180	84,472,865	42,495,951	997	3,108	1,649
		FindBugs	8,686	767,965	319,103	1,795	13,837	7,184
		JLint	4,611	732,496	175,002	27	1,029	364
		PMD	30,281	913,082	391,262	116	3,921	1,577
		Randoop	0	0	0	116	816	286
	Max	Checkst.	20,717,406	129,182,401	64,471,844	1,976	12,401	5,061
		FindBugs	55,583	4,809,618	1,691,975	4,259	21,606	11,239
		JLint	87,128	1,114,570	371,335	238	1,398	642
		PMD	785,208	8,079,371	4,095,392	335	3,881	1,618
		Randoop	0	13	1	1,774	2,984	2,456

Chapter 7

Related Work

In this chapter we discuss the related work to the data structure repair and automatic random program generation approaches. Specifically, the first section covers the related work on the data structure repair and the second section covers the related work on the automatic random program generations.

7.1 Related Work on Data Structure Repair

In this section we discuss other existing data structure repair approaches. We also briefly discuss related repair approaches that do not repair data structures but try to repair a buggy program. Finally we discuss some related approaches that use dynamic symbolic execution techniques in detecting program invariants.

Two other resources that contain a good summary of the related works on data structure repair are Demsky and Bassem's Ph.D. theses [49, 54]. This section benefited greatly from these two sources.

7.1.1 Manual Data Structure Repair

Non-generic data structure repair is not new; traditionally, repair routines have been written manually for the specific data structure at hand [157, 117]. Classic examples include the IBM MVS/XA operating system [117] and the Lucent 5ESS telephone switch [73, 77]. In both the cases, developers used manual error detection and recovery or repair procedures. These resulted in an order of magnitude increase in system reliability [71]. Other such hand-coded repair tools are fsck [3] and chkdsk [1].

These tools scan the file system at boot time and repair any inconsistencies. But these tools use domain-specific repairs that are not generalizable. Generic data structure repair, pioneered by Demsky and Rinard [48, 50], is a relatively new area of research.

7.1.2 Specification or Model-based Repair

In [48, 50], Demsky and Rinard proposed specification based data structure repair technique. They use specification languages to define rules to transform concrete data structures and their consistency constraints into models (consisting of sets and relations). They then check for model consistency against the constraints and when violations are detected they apply repair actions in the model to conform to the constraints. Later concrete data structure updates are made to reflect the changes done in the model. The main limitation of this approach is that, developer has to learn these specification languages other than the programming language to transform the data structure and it's constraints into the models. Apart from having deep understanding of the data structure to define model definition rules and constraints correctly to make the repair effective, a user might also need to define cost factors for selecting the best repair action among possible repairs.

7.1.3 Assertion-based Repair

Khurshid, Garcia and Suen are the pioneers of assertion-based data structure repair [93, 66, 153]. Based on these work, Elkarablieh et al. proposed their repair approach which are closely related to our approach: [56, 57, 58], They consider the last accessed field in the repOk method as the corrupted field and make this field their target and tires to mutate it with all possible values iteratively until repOk passes. After trying all possible values, if no fix is possible, it then backtracks in the list of accessed fields and continues the process.

To prune the search space, they applied static analysis in [57] to find the recurrent field that is used to traverse the data structure. If the error is in such a field, they only consider *forward* or not yet seen object values in the traversal as solutions. In [58] they introduced checkpoint based repair where they store the data structure state and do not run the complete repOk after each repair attempt.

7.1.4 Behavioral Specification and Contract-based Repair

Recently Zaeem et al. [176] introduced behavioral specification language (they used Alloy[85]) on top of Juzi [56] to leverage the constraint solvers to repair a corrupted data structure. They take a method's pre- and post-condition as well as the class invariant written in the specification language into consideration and use Kodkod [159] - a SAT-based constraint solver to get the solution. Kodkod works in first order logic with relations, transitive closure and partial models and expects a finite bound for the search space for each of the types. Zaeem et al. also observed the importance of fields other than the one accessed last in the repOk method in repairing a data structure [175].

In attempt to repair, they first invoke the SAT solver and try to find a solution only by mutating the 'write' fields. If unsuccessful, they repeat the process but mutate the read fields. In the worst case, when the corrupted field is the one that was not touched (written or read) during the repOk execution, they use the SAT solver produced minimum unsatisfiable constraint core and analyse it to find the field. Unfortunately, none of the prototype tools of these approaches are currently available for a performance comparison with our approach.

Samimi et al. developed a tool called PBnJ that also uses behavioral specification language [139]. PBnJ has its own prototype compiler that augments Java with the facility of executable specifications [162, 96] supporting class invariants and method post-conditions in a form of first order relational logic. Similar to Zaeem et al.'s work [176], PBn uses the Alloy specification language and uses Kodkod constraint solver as the back-end of its analyzer.

But the approaches differ in other aspects. For example, PBnJ converts the specifications automatically into Java methods in such a way that the programmers can use them as conventional Java code. These methods can be used to dynamically detect the contract violation. But Zaeem et al.'s approach does not convert the specifications into Java code. For the expected finite bound on the search space for Kodkod, PBnJ allows the programmer to specify these bounds resulting in a pre-bounded search for the constraint solver. On the other hand, Zaeem et al.'s approach [176] depends on heuristics and iterative based algorithms to specify the bounds and uses similarity metric to evaluate the effectiveness of the chosen bound. Moreover, while PBnJ only takes the pre-state of the execution, ignoring the post-state and the method implementation, and depends completely on the Kodkod SAT solver, Zaeem et al.'a approach [176] considers both the pre- and post-state of the execution.

Both the approaches incur the overhead of manually transforming the contracts into the specification language; correctness of which determines the performance and the validity of the whole technique.

7.1.5 Abstraction-based Repair

To avoid repeated search for a solution for similar errors, Zaeem etal. recently proposed a memoization approach of repair in [177]. The main idea of this approach is to learn from previous errors and capture their corresponding fixes. In the future, for any data structure corruption, they propose to attempt these fixes first before invoking any expensive search for the solution. For a successful such repair, they prioritize the applied fix, and for an unsuccessful attempt they call the usual repair algorithm to get the solution but captures the repair to train the system. The core of the approach is basically abstraction of previous actual fixes and later, concretization of them into actual mutations for a repair attempt. It still is a iterative process and suffers if the fixes are too specific for a data structure to generalize. This approach works well if there is a repeated error of similar kinds in the runtime. For newer errors it will carry the the overhead of unsuccessful attempts of previous fixes, invoking the underlying repair module and then capturing and abstracting out the repair.

7.1.6 Other Repair Approaches

Our technique which updates the heap for successful execution of a program, is related to but differs from other techniques that work to fix a buggy program. For example, in [167] Weimer et al. applied genetic programming concepts in automatic repair of a buggy program. It takes as input an unannotated program, a set of successful test cases and a single failing test case. It then applies genetic programming to evolve the program by source code modification until it passes all the test cases including the failing one. Based on this approach, Schulte et al. in [143], took the repair at the assembly code level and exposed some of the limitations of [167]. Since they mutate the assembly code (for C/C++) and bytecode (for Java) programs instead of the higher level source code, their approach extends to repairing bugs in programs written in different programming languages.

Perkins et al. in [128] proposed a repair approach for deployed windows x86 programs where no source code or developer assistance is available. It can fix out-of-bound memory write and illegal control flow transfer attacks on deployed programs. Chang et al. [32] proposed a technique that handles compatibility and integration issues of programs that have third party library or component invocation. Wei et al.

in [166] presented an advisory tool that suggests corrections for faulty implementations of programs written in the Eiffel programming language. Lewis et al. in [100] utilized distributed event-monitoring of a software execution to repair software faults at runtime.

Long et al. proposed automatic input rectification technique in [104]. Database researchers also have developed repair algorithms that enforce consistency constraints at the level of tuples and relations in a database [31, 160]. But these do not repair the lower-level data structures that implement the abstraction.

7.1.7 Existing Approaches for Invariant Detection

Existing data structure repair algorithms expect the user to provide the integrity constraints or the repOk method. Writing these repOk methods is hard and require deep understanding of the underlying data structure. For a complex structure with complex constrains, it can be error prone to produce the precise predicates. There are some frameworks to help users synthesize these constraints. These frameworks utilize static and dynamic techniques to suggest various forms of specifications. For example, [40, 140] can be used to find loop invariants, [115, 138] helps detect heap abstractions and [155, 168] for API level specifications. There are several works for dynamically discovering likely program invariants [61, 62, 42, 41, 127]. Another tool that generates constraints for complex data structure is Deryaft [108]. But unfortunately the tool executable is not available for experimentation.

7.2 Related Work on Random Program Generation

In the remainder of this section we focus on related grammar-based test input generation techniques. Grammar-based test input generation was pioneered by Hanford [74] and Purdom [132] in the 1970s and can be roughly divided into two broad categories, random and systematic.

In the following, we discuss pieces of related work in more detail that are either representative or closely related. Additional related work can be found in a survey article on generating programs for compiler testing [19].

7.2.1 Probabilistic Grammar-Based Program Generation

Several earlier pieces of work have used probabilistic grammar-based random program generation before [118, 112, 148, 173, 26, 147, 172, 44]. However, earlier work mostly focused on testing and debugging. These approaches thus tried to systematically cover corner cases and bugs that are otherwise hard to find. To simplify debugging, the focus was on triggering these corner cases with minimized, focused programs or program fragments. From our perspective, the earlier approaches could be described as generating a collection of maximally diverse micro-benchmarks of rare program shapes. We aim at end-to-end benchmarking and therefore generate large, complex benchmark applications that are close to realistic applications but satisfy specific user-defined constraints.

An early random or probabilistic program generator that is guided by a programming language grammar is presented by Murali and Shyamasundar [118]. The technique targets the PL compiler for a subset of Pascal, the canonical procedural programming language.

An early expressive language for grammar-based random program generation is presented by Maurer [112]. That is, the Data-Generation Language or DGL is more expressive than context-free languages, as it supports various actions. The approach generates test suites in the C programming language for functional testing of VLSI circuits.

Burgess describes a system for testing optimizing Fortran compilers [26]. The user specifies the Fortran syntax in an attribute grammar and uses the attributes to express complex correctness rules. The user can also assign probabilities or weights to individual production rules and thereby control how frequently they are utilized in program generation. The generated programs are relatively small, with a size of up to 4kLOC, compared to up to 5MLOC by RUGRAT.

Sirer and Bershad [148] describe probabilistic testing with production grammars. A production grammar is a context-free grammar that can be enhanced with probabilities and actions. The work also introduces the concrete domain specific language (DSL) *lava* for specifying production grammars. The lava language was used to generate Java bytecode programs for testing Java virtual machines. The generated programs ranged up to 60k bytecode instructions. On the other hand, in our experiments we generate large (up to 2.5MLOC) Java source code programs and compare source code to bytecode compilers.

In recent work, Csmith constructs legal C programs randomly using a subset of the C language production rules [172]. Specifically, Csmith consults a probability table, similar to our stochastic selection. Csmith systematically avoids generating programs that use language features classified as undefined or unspecified by the C language. To achieve the goal, CSmith employs selective construction and analysis of the generated programs. Unlike RUGRAT, Csmith does not support object-oriented language features.

Other than testing C compilers, Cuoq et al. used Csmith for testing static analyzers [44]. They tested Frama-C, a 300kLOC size framework for analysis and transformation of C programs and found 50 bugs.

In the domain of object-oriented programs, a random program generator has been used to test Java just-in-time compilers [173]. This generator takes the number of desired classes and branches as input. Then, it generates branches and fills them randomly with bytecode instructions. In contrast to RUGRAT, this generator does not allow features such as recursive calls. Moreover, it was evaluated only on small programs with up to ten classes, ten methods per class, and less than 100 bytecode instructions per method. We were unable to obtain the tool to compare it with RUGRAT.

7.2.2 Test Program Generation by Combinatorial Grammar Production Rule Coverage

Combinatorial coverage of grammar production rules is a an alternative to stochastic production rule coverage. In the following we briefly review representative and closely related papers.

Purdom has defined a pioneering algorithm for generating small test programs from a given programming language grammar. That is, Purdom's algorithm generates programs that cover each production rule of a given context-free grammar [132].

Celentano et al. describe an early implementation of Purdom's algorithm [30]. This work uses multi-level grammars to support complex correctness rules that cannot be expressed in a context-free grammar alone (such as "define before use"). However it is not clear how this approach scales to complex Java-like languages [78].

Boujarwah et al. implement Purdom's algorithm for a subset of Java [20]. However the implementation has not been applied to generate entire programs and no empirical results are available.

Lämmel and Schulte [98] describe the general-purpose syntax-driven test-data generator *Geno*. Geno works on grammars written in a hybrid of EBNF and algebraic signatures. Geno systematically achieves a user-defined combinatorial coverage of the grammar's production rules. Geno supports computations during test data generation, yielding expressiveness similar to attribute grammars. However Geno does not address the complex correctness rules of Java-like programming languages (such as “define before use”, visibility, and inheritance). Geno is also not available for experimentation.

Fischer, Lämmel, and Zaytsev use a grammar-guided test case generator to compare different concrete grammars of the same grammar specification [64]. For example, the work compares various ANTLR grammars of the Java 5 language specification. However, the program generation technique ignores semantic rules (such as “define before use”) and removes all such rules from the input ANTLR grammars, yielding context-free grammars.

Harm and Lämmel extend test case generation from systematically covering context-free grammar production rules to systematically covering production rules of attribute grammars [76]. For generating benchmark programs, this technique may enable generating programs that satisfy semantic correctness rules (such as “define before use”). However the scalability of the technique is unclear [98] and the technique has not been applied to Java-like programming languages.

In recent work, Hoffman et al. present YouGen, a practical tool for combinatorial production rule coverage [79]. Similar to earlier work, YouGen takes as input a context-free grammar. YouGen has a wider range of configuration options than previous combinatorial production rule coverage generators.

7.2.3 Exhaustive Test Program Generation

Exhaustive test program generation aims at enumerating all possible test programs up to a given size. In the following we discuss three representative recent approaches.

Coppit and Lian describe *yagg*, a generator for test data generators that exhaustively enumerate all possible test data up to a given length [38]. The yagg tool supports context-free input grammars that can be enriched with semantic actions.

ASTGen by Daniel et al. systematically generates small Java programs [47]. However, ASTGen requires the user to combine several generators. More importantly, many generated programs have compile errors, and they do not have complex structures (e.g., only value equality (==) is supported in conditions and no deep **if** nesting is possible).

Majumdar and Xu describe a directed test program generation technique that attempts to exhaust the execution paths of a particular compiler or program analysis tool under test [106]. The technique converts a given context-free grammar into a symbolic grammar, exhaustively derives all possible symbolic strings (programs) up to a certain size, and uses these strings in a dynamic symbolic or concolic execution as inputs to the program under test. This directed search yields a small set of representative test programs, as the symbolic reasoning prevents the generation of concrete input programs that cover the same path in the program under test. On the other hand, symbolic reasoning is very expensive, which limits the scalability of the technique. The corresponding tool, *CESE*, has been used to generate small test programs. RUGRAT on the other hand can quickly generate very large random test programs independent of any particular program under test.

7.2.4 Model-Based Test Program Generation

Beyond grammar production rules, other models of programming language specifications exist. Such models often encode rich semantic information and can be covered systematically by program generators. Given the richness of the information encoded in these models, test case generators are typically slower and focus on generating small programs that are focused on testing specific features.

For example, Zhao et al. capture the rules under which individual compiler optimizations can be applied in temporal logic [179]. The JTT tool then systematically generates focused test programs to test individual compiler optimizations. However it is not clear how this approach scales to entire applications and especially large-scale benchmark applications.

7.2.5 Random Test Data Generation for Different Domains

Random test data generators have been applied to domains related to object-oriented programming such as generating valid XML files and generating SQL queries. In the following we focus on SQL as an example domain.

Probabilistic test data generation has been successfully used in testing relational database engines, where complex SQL statements are generated using a random SQL statement generator [149]. RUGRAT extends this idea by applying it to imperative languages such as C++ and Java in that RUGRAT generalizes the approach to generate applications with predefined properties while the SQL statement generator is designed only for a declarative language such as SQL.

A few other approaches are created for generating SQL statements and query sets. One of them is QGEN, a flexible, high-level query generator optimized for decision support system evaluation. QGEN generates arbitrary query sets, which conform to a selected statistical profile without requiring that the queries be statically defined or disclosed prior to testing [130]. QGEN links query syntax with abstracted data distributions, enabling users to parameterize their query workload to match an emerging access pattern or data set modification.

Another recent approach for random SQL generation is a work by Khurshid et al. that generates syntactically and semantically correct SQL queries as inputs for testing relational databases [6]. They leverage the SAT-based Alloy tool-set to reduce the problem of generating valid SQL queries into a SAT problem. With their approach, SQL query constraints are translated into Alloy models, which enable it to generate valid queries that cannot be automatically generated using conventional grammar-based generators. Both this approach and QGEN are complementary to RUGRAT, since the latter can use generated SQL statements to integrate in its generated Java and C++ programs to interact with backend databases. This is our work in progress that gives positive initial results.

Interestingly, generating random images is widely used to evaluate image processing and pattern recognition algorithms [109, 135]. Essentially, finding images with desired properties to evaluate specific algorithms is difficult and laborious; not always these images can be located on the Internet. Yet it is important to obtain images that have specific geometric figures that highlight certain properties of algorithms that use these images. Generating images with desired properties is a standard practice in image processing and pattern recognition [25, 87, 43, 111].

7.2.6 Other Non-Generated Benchmarks

Other benchmarks of test programs have been developed besides the already discussed widely used DaCapo Java benchmarks [17].

For example, Sewe et. al introduce a Scala benchmark based on the popular DaCapo benchmark for the JVM [146]. Several programming languages (e.g., Scala, Clojure, Groovy, JRuby, and Jython) are typically compiled to Java bytecode and target the JVM. But in JVM research, benchmarks written in these languages are not commonly in use. The authors address this issue by presenting a Scala benchmark and comparing it with the popular DaCapo benchmark on different bytecode metrics. The results show differences between Scala and Java code.

Chapter 8

Conclusion

We have shown that generic automatic data structure repair can be implemented with full dynamic symbolic execution. We have implemented our approach in a prototype and compared its performance with one of the state-of-the-art tool Juzi in repairing corrupted data structures. We have shown that such an implementation can solve some of the problems of the existing generic repair approaches.

We have also explored the potential of automatic program generation for program analysis tool evaluation. We picked the recent automatic program generator tool RUGRAT to generate dozens of Java applications, ranging from 300 LOC to 5 MLOC. We used these generated programs to benchmark several versions of a popular Java source to bytecode compiler as well as popular program analysis and testing tools. We have shown that RUGRAT can be used in benchmarking program analysis tools and detect bugs in them.

References

[1] "Chkdsk product documentation page," http://tinyurl.com/qopyc, windows XP.

[2] "Cosmic rays may have caused Qantas jet's plunge," http://goo.gl/8X224d, published online in shm.com.au on November 18, 2009.

[3] "Ext2 fsck manual page," http://e2fsprogs.sourceforge.net, unix file system consistency checker.

[4] "Intel plans to tackle cosmic ray threat," http://goo.gl/gNGLWQ, published online at bbc.com.uk on April 8, 2008.

[5] "International standard - ISO/IEC 14764 IEEE std 14764-2006 software engineering 2013; software life cycle processes 2013; maintenance," *ISO/IEC 14764:2006 (E) IEEE Std 14764-2006 Revision of IEEE Std 1219-1998*, pp. 1–46, 2006.

[6] S. Abdul Khalek and S. Khurshid, "Automated SQL query generation for systematic testing of database engines," in *Proc. IEEE/ACM International Conference on Automated Software Engineering (ASE).* ACM, Sep. 2010, pp. 329–332.

[7] A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, Jan. 1986.

[8] A. Arcuri and X. Yao, "A novel co-evolutionary approach to automatic software bug fixing," in *Evolutionary Computation, 2008. CEC 2008.(IEEE World Congress on Computational Intelligence). IEEE Congress on.* IEEE, 2008, pp. 162–168.

[9] C. Artho and K. Havelund, “Applying Jlint to space exploration software,” in *Proc. 5th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI).* Springer, Jan. 2004, pp. 297–308.

[10] A. Austin and L. Williams, “One technique is not enough: A comparison of vulnerability discovery techniques,” in *Proc. 5th International Symposium on Empirical Software Engineering and Measurement (ESEM).* IEEE, Sep. 2011, pp. 97–106.

[11] A. Avizienis, “The methodology of n-version programming,” *Software fault tolerance*, vol. 3, pp. 23–46, 1995.

[12] C. Barrett and S. Berezin, “CVC Lite: A new implementation of the cooperating validity checker,” in *Computer Aided Verification.* Springer, 2004, pp. 515–518.

[13] C. Barrett, M. Deters, L. de Moura, A. Oliveras, and A. Stump, “6 Years of SMT-COMP,” *Journal of Automated Reasoning*, pp. 1–35, 2012.

[14] C. W. Barrett, L. M. de Moura, and A. Stump, “Design and results of the first satisfiability modulo theories competition (SMT-COMP 2005),” *Journal of Automated Reasoning*, vol. 35, no. 4, pp. 373–390, Nov. 2005.

[15] ——, “SMT-COMP: Satisfiability modulo theories competition,” in *Proc. 17th International Conference on Computer Aided Verification (CAV).* Springer, Jul. 2005, pp. 20–23.

[16] B. Beizer, *Software testing techniques.* Dreamtech Press, 2003.

[17] S. M. Blackburn *et al.*, “The DaCapo benchmarks: Java benchmarking development and analysis,” in *Proc. 21st Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA).* ACM, Oct. 2006, pp. 169–190.

[18] ——, “Wake up and smell the coffee: Evaluation methodology for the 21st century,” vol. 51, no. 8, pp. 83–89, Aug. 2008.

[19] A. S. Boujarwah and K. Saleh, “Compiler test case generation methods: A survey and assessment,” *Information & Software Technology (IST)*, vol. 39, no. 9, pp. 617–625, 1997.

[20] A. S. Boujarwah, K. Saleh, and J. Al-Dallal, “Testing syntax and semantic coverage of java language compilers,” *Information & Software Technology (IST)*, vol. 41, no. 1, pp. 15–28, 1999.

[21] C. Boyapati, S. Khurshid, and D. Marinov, “Korat: Automated testing based on Java predicates,” in *Proc. ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA).* ACM, Jul. 2002, pp. 123–133.

[22] D. Brumley, J. Caballero, Z. Liang, J. Newsome, and D. Song, “Towards automatic discovery of deviations in binary implementations with applications to error detection and fingerprint generation,” in *Usenix Security Symposium*, 2007.

[23] D. Brumley, P. Poosankam, D. Song, and J. Zheng, “Automatic patch-based exploit generation is possible: Techniques and implications,” in *Security and Privacy, 2008. SP 2008. IEEE Symposium on.* IEEE, 2008, pp. 143–157.

[24] É. Bruneton, R. Lenglet, and T. Coupaye, “ASM: a code manipulation tool to implement adaptable systems,” in *Proc. ACM SIGOPS France Journées Composants 2002: Systèmes à composants adaptables et extensibles (Adaptable and extensible component systems)*, Nov. 2002.

[25] C. Buehler, W. Matusik, L. McMillan, and S. Gortler, “Creating and rendering image-based visual hulls,” Cambridge, MA, USA, Tech. Rep., 1999.

[26] C. J. Burgess, “The automated generation of test cases for compilers,” *Software Testing, Verification & Reliability (STVR)*, vol. 4, no. 2, pp. 81–99, Jun. 1994.

[27] J. Caballero, P. Poosankam, S. McCamant, D. Song *et al.*, “Input generation via decomposition and re-stitching: Finding bugs in malware,” in *Proceedings*

of the 17th ACM conference on Computer and communications security. ACM, 2010, pp. 413–425.

[28] C. Cadar and D. R. Engler, "Execution generated test cases: How to make systems code crash itself," in *Proc. 12th International SPIN Workshop on Model Checking of Software.* Springer, Aug. 2005, pp. 2–23.

[29] C. Cadar, P. Godefroid, S. Khurshid, C. S. Păsăreanu, K. Sen, N. Tillmann, and W. Visser, "Symbolic execution for software testing in practice: preliminary assessment," in *Proceedings of the 33rd International Conference on Software Engineering.* ACM, 2011, pp. 1066–1071.

[30] A. Celentano, S. Crespi-Reghizzi, P. D. Vigna, C. Ghezzi, G. Granata, and F. Savoretti, "Compiler testing using a sentence generator," vol. 10, no. 11, pp. 897–918, Nov. 1980.

[31] S. Ceri, P. Fraternali, S. Paraboschi, and L. Tanca, "Automatic generation of production rules for integrity maintenance," *ACM Transactions on Database Systems (TODS)*, vol. 19, no. 3, pp. 367–422, 1994.

[32] H. Chang, L. Mariani, and M. Pezzè, "In-field healing of integration problems with COTS components," in *Proc. 31st ACM/IEEE International Conference on Software Engineering (ICSE).* IEEE, May 2009, pp. 166–176.

[33] L. Chen and A. Avizienis, "N-version programming: A fault-tolerance approach to reliability of software operation," in *Proc. 8th IEEE Int. Symp. on Fault-Tolerant Computing (FTCS-8)*, 1978, pp. 3–9.

[34] G. A. Cohen, J. S. Chase, and D. L. Kaminsky, "Automatic program transformation with Joie," in *Proc. USENIX Annual Technical Symposium.* USENIX, Jun. 1998, pp. 167–178.

[35] S. Cohen and B. Kimelfeld, "Querying parse trees of stochastic context-free grammars," in *Proc. 13th International Conference on Database Theory (ICDT).* ACM, Mar. 2010, pp. 62–75.

[36] C. Collberg, G. Myles, and M. Stepp, "An empirical study of Java bytecode programs," *Software—Practice & Experience*, vol. 37, no. 6, pp. 581–641, May 2007.

[37] T. Copeland, *PMD Applied.* Centennial Books, Nov. 2005.

[38] D. Coppit and J. Lian, "yagg: An easy-to-use generator for structured test inputs," in *Proc. 20th IEEE/ACM International Conference on Automated Software Engineering (ASE).* ACM, 2005, pp. 356–359.

[39] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms (3. ed.).* MIT Press, 2009.

[40] P. Cousot and N. Halbwachs, "Automatic discovery of linear restraints among variables of a program," in *Proceedings of the 5th ACM SIGACT-SIGPLAN symposium on Principles of programming languages.* ACM, 1978, pp. 84–96.

[41] C. Csallner and Y. Smaragdakis, "Dynamically discovering likely interface invariants," in *Proc. 28th ACM/IEEE International Conference on Software Engineering (ICSE), Emerging Results Track.* ACM, May 2006, pp. 861–864.

[42] C. Csallner, N. Tillmann, and Y. Smaragdakis, "DySy: Dynamic symbolic execution for invariant inference," in *Proc. 30th ACM/IEEE International Conference on Software Engineering (ICSE).* ACM, May 2008, pp. 281–290.

[43] K. Culik, II and S. Dube, "Affine automata: a technique to generate complex images," in *Proceedings on Mathematical foundations of computer science 1990*, ser. MFCS '90. New York, NY, USA: Springer-Verlag New York, Inc., 1990, pp. 224–231. [Online]. Available: http://dl.acm.org/citation.cfm?id=88581.88646

[44] P. Cuoq, B. Monate, A. Pacalet, V. Prevosto, J. Regehr, B. Yakobowski, and X. Yang, "Testing static analyzers with randomly generated programs," in *Proc. 4th NASA Formal Methods Symposium (NFM).* Springer, Apr. 2012, pp. 120–125.

[45] V. Dallmeier and T. Zimmermann, "Extraction of bug localization benchmarks from history," in *Proc. 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE).* ACM, 2007, pp. 433–436.

[46] M. d'Amorim, C. Pacheco, T. Xie, D. Marinov, and M. D. Ernst, "An empirical comparison of automated generation and classification techniques for object-oriented unit testing," in *Proc. 21st IEEE/ACM International Conference on Automated Software Engineering (ASE).* IEEE, Sep. 2006, pp. 59–68.

[47] B. Daniel, D. Dig, K. Garcia, and D. Marinov, "Automated testing of refactoring engines," in *Proc. 15th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE).* ACM, Sep. 2007, pp. 185–194.

[48] B. Demsky and M. C. Rinard, "Goal-directed reasoning for specification-based data structure repair," *IEEE Transactions on Software Engineering*, vol. 32, no. 12, pp. 931–951, 2006.

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

[50] B. Demsky and M. C. Rinard, "Automatic detection and repair of errors in data structures," in *Proc. 18th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA).* ACM, Oct. 2003, pp. 78–95.

[51] H. Do, S. Elbaum, and G. Rothermel, "Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact," *Empirical Software Engineering (ESE)*, vol. 10, no. 4, Oct. 2005.

[52] M. Dowson, "The ariane 5 software failure," *ACM SIGSOFT Software Engineering Notes*, vol. 22, no. 2, p. 84, 1997.

[53] B. Dufour, K. Driesen, L. Hendren, and C. Verbrugge, "Dynamic metrics for Java," in *Proc. 18th Annual ACM SIGPLAN Conference on Object-Oriented Programing, Systems, Languages, and Applications (OOPSLA).* ACM, Oct. 2003, pp. 149–168.

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

[55] B. Elkarablieh, I. Garcia, Y. L. Suen, and S. Khurshid, "Assertion-based repair of complex data structures," in *Proc. 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE).* ACM, Nov. 2007, pp. 64–73.

[56] B. Elkarablieh and S. Khurshid, "Juzi: A tool for repairing complex data structures," in *Proc. 30th ACM/IEEE International Conference on Software Engineering (ICSE).* ACM, May 2008, pp. 855–858.

[57] B. Elkarablieh, S. Khurshid, D. Vu, and K. S. McKinley, "Starc: Static analysis for efficient repair of complex data," in *Proc. 22nd ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA).* ACM, Oct. 2007, pp. 387–404.

[58] B. Elkarablieh, D. Marinov, and S. Khurshid, "Efficient solving of structural constraints," in *Proc. ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA).* ACM, Jul. 2008, pp. 39–50.

[59] B. Elkarablieh, Y. Zayour, and S. Khurshid, "Efficiently generating structurally complex inputs with thousands of objects," in *ECOOP 2007–Object-Oriented Programming.* Springer, 2007, pp. 248–272.

[60] I. Erete and A. Orso, "Optimizing constraint solving to better support symbolic execution," in *Software Testing, Verification and Validation Workshops*

(ICSTW), 2011 IEEE Fourth International Conference on. IEEE, 2011, pp. 310–315.

[61] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin, "Dynamically discovering likely program invariants to support program evolution," in *Proc. 21st ACM/IEEE International Conference on Software Engineering (ICSE).* ACM, May 1999, pp. 213–224.

[62] ——, "Dynamically discovering likely program invariants to support program evolution," *IEEE Transactions on Software Engineering (TSE)*, vol. 27, no. 2, pp. 99–123, Feb. 2001.

[63] B. J. Evans and M. Verburg, *The Well-Grounded Java Developer: Vital techniques of Java 7 and polyglot programming.* Manning Publications, Jul. 2012.

[64] B. Fischer, R. Lämmel, and V. Zaytsev, "Comparison of context-free grammars based on parsing generated test data," in *Proc. 4th International Conference on Software Language Engineering (SLE).* Springer, Jul. 2011, pp. 324–343.

[65] V. Ganesh, T. Leek, and M. Rinard, "Taint-based directed whitebox fuzzing," in *Software Engineering, 2009. ICSE 2009. IEEE 31st International Conference on.* IEEE, 2009, pp. 474–484.

[66] I. García, "Enabling sysmbolic execution of Java programs using bytecode instrumentation." 2005.

[67] R. L. Glass, "Frequently forgotten fundamental facts about software engineering," *Software, IEEE*, vol. 18, no. 3, pp. 112–111, 2001.

[68] P. Godefroid, N. Klarlund, and K. Sen, "Dart: Directed automated random testing," in *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI).* ACM, Jun. 2005, pp. 213–223.

[69] P. Godefroid, M. Y. Levin, D. A. Molnar *et al.*, "Automated whitebox fuzz testing." in *NDSS*, vol. 8, 2008, pp. 151–166.

[70] S. Govindavajhala and A. W. Appel, “Using memory errors to attack a virtual machine,” in *Security and Privacy, 2003. Proceedings. 2003 Symposium on.* IEEE, 2003, pp. 154–165.

[71] J. Gray and A. Reuter, *Transaction processing.* Kaufmann, 1993.

[72] M. Grechanik, C. McMillan, L. DeFerrari, M. Comi, S. Crespi-Reghizzi, D. Poshyvanyk, C. Fu, Q. Xie, and C. Ghezzi, “An empirical investigation into a large-scale Java open source code repository,” in *Proc. 4th International Symposium on Empirical Software Engineering and Measurement (ESEM).* ACM, Sep. 2010.

[73] N. K. Gupta, L. J. Jagadeesan, E. Kouteofios, and D. M. Weiss, “Auditdraw: Generating audits the FAST way,” in *Requirements Engineering, 1997., Proceedings of the Third IEEE International Symposium on.* IEEE, 1997, pp. 188–197.

[74] K. V. Hanford, “Automatic generation of test cases,” *IBM Systems Journal*, 1970.

[75] S. Hangal and M. S. Lam, “Tracking down software bugs using automatic anomaly detection,” in *Proc. 24th ACM/IEEE International Conference on Software Engineering (ICSE).* ACM, May 2002, pp. 291–301.

[76] J. Harm and R. Lämmel, “Two-dimensional approximation coverage,” *Informatica (Slovenia)*, vol. 24, no. 3, Nov. 2000.

[77] R. Haugk, Lax and Williams, “The 5ESS(TM) switching system: maintenance capabilities.”

[78] M. Hennessy and J. F. Power, “Analysing the effectiveness of rule-coverage as a reduction criterion for test suites of grammar-based software,” *Empirical Software Engineering (ESE)*, vol. 13, no. 4, pp. 343–368, Aug. 2008.

[79] D. Hoffman, D. Ly-Gagnon, P. A. Strooper, and H.-Y. Wang, “Grammar-based test generation with YouGen,” vol. 41, no. 4, pp. 427–447, Apr. 2011.

[80] D. Hovemeyer and W. Pugh, “Finding bugs is easy,” in *Companion to the 19th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA).* ACM, Oct. 2004, pp. 132–136.

[81] I. Hussain and C. Csallner, “DSDSR: A tool that uses dynamic symbolic execution for data structure repair,” in *Proc. 8th International Workshop on Dynamic Analysis (WODA).* ACM, Jul. 2010, pp. 20–25.

[82] ——, “Dynamic symbolic data structure repair,” in *Proc. 32nd ACM/IEEE International Conference on Software Engineering (ICSE), Volume 2, Emerging Results Track.* ACM, May 2010, pp. 215–218.

[83] I. Hussain, C. Csallner, M. Grechanik, C. Fu, Q. Xie, S. Park, K. Taneja, and B. M. Hossain, “Evaluating program analysis and testing tools with the RUGRAT random benchmark application generator,” in *Proc. 10th International Workshop on Dynamic Analysis (WODA).* ACM, Jul. 2012, pp. 1–6.

[84] M. Islam and C. Csallner, “Dsc+mock: A test case + mock class generator in support of coding against interfaces,” in *Proc. 8th International Workshop on Dynamic Analysis (WODA).* ACM, Jul. 2010, pp. 26–31.

[85] D. Jackson, “Alloy: A lightweight object modelling notation,” *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 11, no. 2, pp. 256–290, Apr. 2002.

[86] D. Jackson, M. Thomas, L. I. Millett *et al.*, *Software for Dependable Systems: Sufficient Evidence?* National Academies Press, 2007.

[87] Z. Jin, D. Yang, L. Yong-jun, W. Ying, and W. Ru-long, “Survey on simplified olfactory bionic model to generate texture images,” in *Proceedings of the 19th international conference on Neural Information Processing - Volume Part II,*

ser. ICONIP'12. Berlin, Heidelberg: Springer-Verlag, 2012, pp. 316–323. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-34481-7_39

[88] A. Joshi, L. Eeckhout, R. H. Bell, Jr., and L. K. John, "Distilling the essence of proprietary workloads into miniature benchmarks," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 5, no. 2, pp. 10:1–10:33, Sep. 2008.

[89] C. Kaner, "Exploratory testing," in *Quality Assurance Institute Worldwide Annual Software Testing Conference, Orlando, FL*, 2006.

[90] Y. Kannan and K. Sen, "Universal symbolic execution and its application to likely data structure invariant generation," in *Proc. ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA).* ACM, Jul. 2008, pp. 283–294.

[91] K. Kanoun and L. Spainhower, *Dependability Benchmarking for Computer Systems.* Wiley-IEEE, Jul. 2008.

[92] P. A. Keiller and D. R. Miller, "On the use and the performance of software reliability growth models," *Reliability Engineering & System Safety*, vol. 32, no. 1, pp. 95–117, 1991.

[93] S. Khurshid, I. García, and Y. L. Suen, "Repairing structurally complex data," in *Model Checking Software, 12th International SPIN Workshop*, vol. 3639, 2005, pp. 123–138.

[94] A. Kieyzun, P. J. Guo, K. Jayaraman, and M. D. Ernst, "Automatic creation of SQL injection and cross-site scripting attacks," in *Software Engineering, 2009. ICSE 2009. IEEE 31st International Conference on.* IEEE, 2009, pp. 199–209.

[95] J. C. Knight and N. G. Leveson, "An experimental evaluation of the assumption of independence in multiversion programming," *Software Engineering, IEEE Transactions on*, no. 1, pp. 96–109, 1986.

[96] B. Krause and T. Wahls, "jmle: A tool for executing JML specifications via constraint programming," in *Proc. FMICS/PDMC*. Springer, Aug. 2006, pp. 293–296.

[97] G. Kudrjavets, N. Nagappan, and T. Ball, "Assessing the relationship between software assertions and faults: An empirical investigation," in *Proc. 17th IEEE International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, Nov. 2006, pp. 204–212.

[98] R. Lämmel and W. Schulte, "Controllable combinatorial coverage in grammar-based testing," in *Proc. 18th IFIP TC6/WG6.1 International Conference on Testing of Communicating Systems (TestCom)*. Springer, May 2006, pp. 19–38.

[99] N. G. Leveson and C. S. Turner, "An investigation of the therac-25 accidents," *Computer*, vol. 26, no. 7, pp. 18–41, 1993.

[100] C. Lewis and J. Whitehead, "Runtime repair of software faults using event-driven monitoring," in *Proc. 32nd ACM/IEEE International Conference on Software Engineering (ICSE), Volume 2*. ACM, May 2010, pp. 275–280.

[101] S. D. Liburd, "An n-version electronic voting system." Ph.D. dissertation, Citeseer, 2004.

[102] B. P. Lientz, E. B. Swanson, and G. E. Tompkins, "Characteristics of application software maintenance," *Communications of the ACM*, vol. 21, no. 6, pp. 466–471, 1978.

[103] B. Liskov and J. Guttag, *Program Development in Java: Abstraction, Specification, and Object-Oriented Design*, 1st ed. Addison-Wesley, 2000.

[104] F. Long, V. Ganesh, M. Carbin, S. Sidiroglou, and M. Rinard, "Automatic input rectification," http://hdl.handle.net/1721.1/66170, 2011.

[105] M. R. Lyu *et al.*, *Handbook of software reliability engineering.* IEEE computer society press CA, 1996, vol. 3.

[106] R. Majumdar and R.-G. Xu, "Directed test generation using symbolic grammars," in *Proc. 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE).* ACM, Nov. 2007, pp. 134–143.

[107] M. Z. Malik, K. Ghori, B. Elkarablieh, and S. Khurshid, "A case for automated debugging using data structure repair," in *Proc. 24th IEEE/ACM International Conference on Automated Software Engineering (ASE).* IEEE, Nov. 2009, pp. 620–624.

[108] M. Z. Malik, A. Pervaiz, and S. Khurshid, "Generating representation invariants of structurally complex data," in *Tools and Algorithms for the Construction and Analysis of Systems.* Springer, 2007, pp. 34–49.

[109] T. J. Mantere and J. T. Alander, "Automatic image generation by genetic algorithms for testing halftoning methods," vol. 4197, pp. 297–308, 2000.

[110] M. R. Marri, T. Xie, N. Tillmann, J. de Halleux, and W. Schulte, "An empirical study of testing file-system-dependent software with mock objects," in *Proc. 4th International Workshop on Automation of Software Test (AST).* IEEE, May 2009, pp. 149–153.

[111] B. Martin and R. M. Horton, "A java program to create simulated microarray images," in *Proceedings of the 2004 IEEE Computational Systems Bioinformatics Conference*, ser. CSB '04. Washington, DC, USA: IEEE Computer Society, 2004, pp. 564–565. [Online]. Available: http://dx.doi.org/10.1109/CSB.2004.10

[112] P. M. Maurer, "Generating test data with enhanced context-free grammars," *IEEE Software*, vol. 7, no. 4, pp. 50–55, Jul. 1990.

[113] S. McConnell, *Code Complete*, 2nd ed. Microsoft Press, Jul. 2004.

[114] G. McDaniel, *IBM Dictionary of Computing.* McGraw-Hill, Dec. 1994.

[115] A. Møller and M. I. Schwartzbach, "The pointer assertion logic engine," in *Proc. ACM PLDI*, 2001, pp. 221–231.

[116] L. d. Moura and N. Bjørner, "Z3: An efficient SMT solver," in *Proc. 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS).* Springer, Apr. 2008, pp. 337–340.

[117] S. Mourad and D. Andrews, "On the reliability of the IBM MVS/XA operating system," *IEEE Transactions on Software Engineering (TSE)*, vol. 13, no. 10, pp. 1135–1139, Oct. 1987.

[118] V. Murali and R. K. Shyamasundar, "A sentence generator for a compiler for PT, a Pascal subset," vol. 13, no. 9, pp. 857–869, Sep. 1983.

[119] M. Musuvathi and D. Engler, "Some lessons from using static analysis and software model checking for bug finding," in *Proc. Workshop on Software Model Checking (SoftMC).* Elsevier, Jul. 2003.

[120] G. J. Myers, C. Sandler, and T. Badgett, *The art of software testing.* John Wiley & Sons, 2011.

[121] L. Nagy, R. Ford, and W. Allen, "N-version programming for the detection of zero-day exploits," in *IEEE Topical Conference on Cybersecurity, Daytona Beach, Florida, USA*, 2006.

[122] M. Newman, "Software errors cost us economy $59.5 billion annually," *NIST Assesses Technical Needs of Industry to Improve Software-Testing*, 2002.

[123] C. Pacheco and M. D. Ernst, "Randoop: Feedback-directed random testing for Java," in *Companion to the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2007*, 2007, pp. 815–816.

[124] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball, "Feedback-directed random test generation," in *Proc. 29th ACM/IEEE International Conference on Software Engineering (ICSE).* IEEE, May 2007, pp. 75–84.

[125] J. Pan, "Software reliability," *Dependable Embedded Systems, Carnegie Mellon University*, 1999.

[126] S. Park, I. Hussain, C. Csallner, K. Taneja, B. M. Hossain, M. Grechanik, C. Fu, and Q. Xie, "CarFast: Achieving higher statement coverage faster," in *Proc. 20th ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE).* ACM, Nov. 2012, pp. 35:1–35:11.

[127] J. H. Perkins and M. D. Ernst, "Efficient incremental algorithms for dynamic detection of likely invariants," in *Proc. 12th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE).* ACM, Oct. 2004, pp. 23–32.

[128] J. H. Perkins, S. Kim, S. Larsen, S. Amarasinghe, J. Bachrach, M. Carbin, C. Pacheco, F. Sherwood, S. Sidiroglou, G. Sullivan, W.-F. Wong, Y. Zibin, M. D. Ernst, and M. Rinard, "Automatically patching errors in deployed software," in *Proc. 21st ACM Symposium on Operating Systems Principles (SOSP).* ACM, Oct. 2009, pp. 87–102.

[129] I. Peterson, *Fatal defect: Chasing killer computer bugs.* Mckay, David, 1996.

[130] M. Poess and J. M. Stephens, Jr., "Generating thousand benchmark queries in seconds," in *Proc. 13th International Conference on Very Large Data Bases (VLDB).* Morgan Kaufmann, Aug. 2004, pp. 1045–1053.

[131] P. Poosankam, "Advanced dynamic symbolic execution techniques for security applications," Ph.D. dissertation, Carnegie Mellon University.

[132] P. Purdom, "A sentence generator for testing parsers," *BIT Numerical Mathematics*, vol. 12, no. 3, pp. 366–375, 1972.

[133] D. Qi, A. Roychoudhury, and Z. Liang, "Test generation to expose changes in evolving programs," in *Proceedings of the IEEE/ACM international conference on Automated software engineering.* ACM, 2010, pp. 397–406.

[134] J. Radatz, A. Geraci, and F. Katki, "IEEE standard glossary of software engineering terminology," *IEEE Std*, vol. 610121990, p. 121990, 1990.

[135] H. K. Reghbati and A. Y. C. Lee, *Computer graphics hardware - image generation and display: tutorial.* IEEE, 1988.

[136] N. Rutar, C. B. Almazan, and J. S. Foster, "A comparison of bug finding tools for Java," in *Proc. 15th IEEE International Symposium on Software Reliability Engineering (ISSRE).* IEEE, Nov. 2004, pp. 245–256.

[137] R. H. Saavedra and A. J. Smith, "Analysis of benchmark characteristics and benchmark performance prediction," *ACM Transactions on Computer Systems (TOCS)*, vol. 14, no. 4, pp. 344–384, Nov. 1996.

[138] M. Sagiv, T. Reps, and R. Wilhelm, "Solving shape-analysis problems in languages with destructive updating," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 20, no. 1, pp. 1–50, 1998.

[139] H. Samimi, E. D. Aung, and T. D. Millstein, "Falling back on executable specifications," in *Proc. 24th European Conference on Object-Oriented Programming (ECOOP).* Springer, Jun. 2010, pp. 552–576.

[140] S. Sankaranarayanan, H. B. Sipma, and Z. Manna, "Non-linear loop invariant generation using gröbner bases," *ACM SIGPLAN Notices*, vol. 39, no. 1, pp. 318–329, 2004.

[141] P. Saxena, P. Poosankam, S. McCamant, and D. Song, "Loop-extended symbolic execution on binary programs," in *Proceedings of the eighteenth international symposium on Software testing and analysis.* ACM, 2009, pp. 225–236.

[142] S. Schmeelk, “Towards a unified fault-detection benchmark,” in *Proc. 9th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE).* ACM, Jun. 2010, pp. 61–64.

[143] E. Schulte, S. Forrest, and W. Weimer, “Automated program repair through the evolution of assembly code,” in *Proc. 25th IEEE/ACM International Conference on Automated Software Engineering (ASE).* ACM, Sep. 2010, pp. 313–316.

[144] E. J. Schwartz, T. Avgerinos, and D. Brumley, “All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask),” in *Security and Privacy (SP), 2010 IEEE Symposium on.* IEEE, 2010, pp. 317–331.

[145] K. Sen and G. Agha, “Cute and jCute: Concolic unit testing and explicit path model-checking tools,” in *Proc. International Conference on Computer Aided Verification (CAV).* Springer, Aug. 2006, pp. 419–423.

[146] A. Sewe, M. Mezini, A. Sarimbekov, and W. Binder, “Da Capo con Scala: Design and analysis of a Scala benchmark suite for the Java virtual machine,” in *Proc. ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA).* ACM, 2011, pp. 657–676.

[147] E. G. Sirer, “Testing Java virtual machines,” in *Proc. International Conference on Software Testing And Review (STAR WEST)*, Nov. 1999.

[148] E. G. Sirer and B. Bershad, “Using production grammars in software testing,” in *Proc. 2nd Conference on Domain-Specific Languages (DSL).* ACM, Oct. 1999, pp. 1–13.

[149] D. R. Slutz, “Massive stochastic testing of SQL,” in *Proc. 24rd International Conference on Very Large Data Bases (VLDB).* Morgan Kaufmann, Aug. 1998, pp. 618–622.

[150] D. Song, D. Brumley, H. Yin, J. Caballero, I. Jager, M. G. Kang, Z. Liang, J. Newsome, P. Poosankam, and P. Saxena, “BitBlaze: A new approach to computer security via binary analysis,” in *Information systems security.* Springer, 2008, pp. 1–25.

[151] S. Staber, B. Jobstmann, and R. Bloem, “Finding and fixing faults,” in *Correct Hardware Design and Verification Methods.* Springer, 2005, pp. 35–49.

[152] M. Stumptner and F. Wotawa, “Model-based program debugging and repair.” in *IEA/AIE*, 1996, pp. 155–160.

[153] Y. L. Suen, “Automatically repairing structurally complex data.” 2005.

[154] E. B. Swanson, “The dimensions of maintenance,” in *Proceedings of the 2nd international conference on Software engineering.* IEEE Computer Society Press, 1976, pp. 492–497.

[155] M. Taghdiri and D. Jackson, “Inferring specifications to detect errors in code,” *Automated Software Engineering*, vol. 14, no. 1, pp. 87–121, 2007.

[156] G. Tassey, “The economic impacts of inadequate infrastructure for software testing,” *National Institute of Standards and Technology, RTI Project*, vol. 7007, no. 011, 2002.

[157] D. J. Taylor and J. P. Black, “Principles of data structure error correction,” *IEEE Transactions on Computers*, vol. 31, pp. 602–608, 1982.

[158] N. Tillmann and J. de Halleux, “Pex - white box test generation for .Net,” in *Proc. 2nd International Conference on Tests And Proofs (TAP).* Springer, Apr. 2008, pp. 134–153.

[159] E. Torlak and D. Jackson, “Kodkod: A relational model finder,” in *Proc. 13th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS).* Springer, Mar. 2007, pp. 632–647.

[160] S. D. Urban and L. M. L. Delcambre, “Constraint analysis: A design process for specifying operations on objects,” *Knowledge and Data Engineering, IEEE Transactions on*, vol. 2, no. 4, pp. 391–400, 1990.

[161] S. Wagner, J. Jürjens, C. Koller, and P. Trischberger, “Comparing bug finding tools with reviews and tests,” in *Proc. 17th IFIP TC6/WG 6.1 International Conference on Testing of Communicating Systems (TestCom)*. Springer, May 2005, pp. 40–55.

[162] T. Wahls, G. T. Leavens, and A. L. Baker, “Executing formal specifications with concurrent constraint programming,” *Automated Software Engineering*, vol. 7, no. 4, pp. 315–343, Dec. 2000.

[163] S. Wang and J. Offutt, “Comparison of unit-level automated test generation tools,” in *Proc. 2nd International Conference on Software Testing Verification and Validation (ICST) Workshops*. IEEE, Apr. 2009, pp. 210–219.

[164] M. S. Ware and C. J. Fox, “Securing Java code: Heuristics and an evaluation of static analysis tools,” in *Proc. Workshop on Static Analysis (SAW)*. ACM, 2008, pp. 12–21.

[165] F. Wedyan, D. Alrmuny, and J. M. Bieman, “The effectiveness of automated static analysis tools for fault detection and refactoring prediction,” in *Proc. 2nd International Conference on Software Testing Verification and Validation (ICST)*. IEEE, Apr. 2009, pp. 141–150.

[166] Y. Wei, Y. Pei, C. A. Furia, L. S. Silva, S. Buchholz, B. Meyer, and A. Zeller, “Automated fixing of programs with contracts,” in *Proc. 19th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*. ACM, Jul. 2010, pp. 61–72.

[167] W. Weimer, T. Nguyen, C. Le Goues, and S. Forrest, "Automatically finding patches using genetic programming," in *Proc. 31st ACM/IEEE International Conference on Software Engineering (ICSE).* IEEE, May 2009, pp. 364–374.

[168] J. Whaley, M. C. Martin, and M. S. Lam, "Automatic extraction of object-oriented component interfaces," in *Proc. ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA).* ACM, Jul. 2002, pp. 218–228.

[169] J. William A. Ward, "Role of application benchmarks in the DoD HPC acquisition process," U.S. Army Engineer Research and Development Center, ERDC MSRC Resource, 2005.

[170] J. D. Winston, L. I. Millett *et al.*, *Summary of a Workshop for Software-Intensive Systems and Uncertainty at Scale.* National Academies Press, 2007.

[171] X. Xiao, T. Xie, N. Tillmann, and J. de Halleux, "Precise identification of problems for structural test generation," in *Proc. 33rd International Conference on Software Engineering (ICSE).* ACM, May 2011, pp. 611–620.

[172] X. Yang, Y. Chen, E. Eide, and J. Regehr, "Finding and understanding bugs in C compilers," in *Proc. 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI).* ACM, Jun. 2011, pp. 283–294.

[173] T. Yoshikawa, K. Shimura, and T. Ozawa, "Random program generator for Java JIT compiler test system," in *Proc. 3rd International Conference on Quality Software (QSIC).* IEEE, Nov. 2003, pp. 20–24.

[174] M. Young, *Software testing and analysis: process, principles, and techniques.* John Wiley & Sons, 2008.

[175] R. N. Zaeem, D. Gopinath, S. Khurshid, and K. S. McKinley, "History-aware data structure repair using sat," in *TACAS*, 2012, pp. 2–17.

[176] R. N. Zaeem and S. Khurshid, "Contract-based data structure repair using Alloy," in *Proc. 24th European Conference on Object-Oriented Programming (ECOOP).* Springer, Jun. 2010, pp. 577–598.

[177] R. N. Zaeem, M. Z. Malik, and S. Khurshid, "Repair abstractions for more efficient data structure repair," in *Runtime Verification.* Springer, 2013, pp. 235–250.

[178] H. Zhang and H. B. K. Tan, "An empirical study of class sizes for large Java systems," in *Proc. 14th Asia-Pacific Software Engineering Conference (APSEC).* IEEE, Dec. 2007, pp. 230–237.

[179] C. Zhao, Y. Xue, Q. Tao, L. Guo, and Z. Wang, "Automated test program generation for an industrial optimizing compiler," in *Proc. 4th International Workshop on Automation of Software Test (AST).* IEEE, May 2009, pp. 36–43.

[180] M. Zitser, R. Lippmann, and T. Leek, "Testing static analysis tools using exploitable buffer overflows from open source code," in *Proc. 12th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE).* ACM, Oct. 2004, pp. 97–106.

Biographical Information

Ishtiaque Hussain was born in Dhaka, Bangladesh. He received his B.S. degree from the University of Dhaka, Bangaldesh, in 2007, and Ph.D. from the University of Texas at Arlington (UTA) in 2014 - both in Computer Science. After his B.S. degree in 2007, he workd full-time for a year as a Lecturer in the computer science department of the State University of Bangladesh. Then he came to the U.S. to persue his Ph.D. in 2008. At UTA he worked full-time as a Graduate Research and Teaching Assistant at the computer science and engineering department. In Spring 2014, he worked as a part-time instructor for the same department in UTA. He also had summer internships at the Accenture Technology Labs (2010) and Microsoft Corporation (2013). His research interest is in software engineering, especially in program analysis and automated testing.