COMBINING OVER- AND UNDER-APPROXIMATING PROGRAM ANALYSES FOR AUTOMATIC SOFTWARE TESTING

A Dissertation
Presented to
The Academic Faculty

by

Christoph Csallner

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in the
School of Computer Science

Georgia Institute of Technology
August 2008

COMBINING OVER- AND UNDER-APPROXIMATING PROGRAM ANALYSES FOR AUTOMATIC SOFTWARE TESTING

Approved by:

Dr. Yannis Smaragdakis, Advisor
School of Computer Science
Georgia Institute of Technology

Dr. Matthew Dwyer
Department of Computer Science & Engineering
University of Nebraska

Dr. Alessandro Orso
School of Computer Science
Georgia Institute of Technology

Dr. Santosh Pande
School of Computer Science
Georgia Institute of Technology

Dr. Spencer Rugaber
School of Computer Science
Georgia Institute of Technology

Date Approved: June 30th, 2008

To my family,

for their huge love and support

ACKNOWLEDGEMENTS

This dissertation would not have been possible without the love and support of many people. These people taught me the right concepts, brought me to the right places, introduced me to their colleagues and friends, gave me feedback, kept me motivated, and paid *me* for receiving all these good deeds. After all that, they still make me believe it was my accomplishment. How could I not love these people?

Professor Yannis Smaragdakis has helped me tremendously throughout my graduate student career. Working with Yannis is the best thing that has happened to me at Georgia Tech. He has helped me with all aspects of my career, professional and beyond. There is no way I can repay what he has done for me. I will try to copy and adapt his style as well as I can. My committee members, Professors Matthew Dwyer, Alessandro (Alex) Orso, Santosh Pande, and Spencer Rugaber, were very generous in offering advice, too. They provided many detailed comments on different versions of this dissertation, and I wish I had taken more of them to heart. I am also fortunate that my committee has supported this work from very early on. Alex, for example, besides many other things, has helped me with my JABA experiments.

Professor Jochen Ludewig introduced me to software engineering. He taught a series of classes at Universität Stuttgart that laid the software engineering foundation for this dissertation. Professor Mary Jean Harrold introduced me to the idea of combining static and dynamic program analyses. She posed it as an important challenge for software engineering research in one of the first lectures I attended at Georgia Tech. Professor Panagiotis (Pete) Manolios taught Formal Methods at Georgia Tech, one of the best classes I have ever taken. It gave me much of the background I needed to finish this work.

Professor Michael Ernst gave us a lot of good feedback on our Daikon-related papers. Working with his Daikon system was very nice, due to the high quality of the entire system and documentation. Thank you, Jeff Perkins and the other people of Mike's group who have supported our uses of Daikon. Professor Tao Xie was my first collaborator outside Georgia Tech, an experience I am very grateful about. Many audiences at conferences, workshops, and job talks, as well as anonymous conference and journal reviewers were very generous with their time and gave us good feedback and suggestions that have helped us to improve this work significantly. Thank you, Drs. Mark Grechanik, Martin Bravenboer, Michal Young, Willem Visser, and many others.

Nikolai Tillmann and Dr. Wolfram Schulte hired me for an internship at Microsoft Research, where I could work on one of the most advanced dynamic symbolic program analysis systems I know of. Nikolai has strongly supported me before, during, and after my internship, which made this experience especially nice for me. Even getting this internship was a big motivation for me, let alone working with and learning from these people. Thank you, Drs. Carlos Pacheco, Daan Leijen, Manuel Fähndrich, Nikolaj Bjørner, Patrice Godefroid, Peli de Halleux, Rustan Leino, and Thomas Ball. At my other internship, at Google in Mountain View, Drs. John Penix and YuQian Zhou mentored me to explore automatic test generation in an industrial setting. This was a very exciting time for me, my first job in the Silicon Valley. The other interns made work and free Google gear even more fun. Thank you, Brett Cannon, Flavio Lerda, Joseph Ruthruff, and Nikolaos Vasiloglou. Molham Aref and his team at LogicBlox gave us interesting real-world problems, tools, and data to work with. Thank you for supporting me!

Shan Shan Huang has been a great officemate, teaching me a lot about research in programming languages, type systems, and software engineering. I am also fortunate to have had her as a friend, which has helped me a lot in the last two years of my

Ph.D. work. The SPARC students at Georgia Tech taught me a lot about software engineering. On top of that, they are also a very pleasant group. It has always been a joy to interact with them and learn from them. They made the weekly brown bag seminars and reading group meetings great forums for presenting, discussing, and hearing about new ideas. Thank you, Arjan Seesing, Brian McNamara, Chris Parnin, Daron Vroon, David Fisher, David Zook, David Zurow, Eli Tilevich, George Baah, James (Jim) Bowring, James (Jim) Clause, James (Jim) Jones, Matthew Might, Pavan Chittimalli, Peter Dillinger, Raul Santelices, Saswat Anand, Sudarshan Srinivasan, Taweesup (Term) Apiwattanapong, and William (GJ) Halfond.

Carlton Parker of the World Student Fund at the Georgia Tech YMCA has helped me to get admitted at Georgia Tech in the first place. His constant care and support has helped me to keep going. I have generally been blessed with a great group of friends, who have supported me throughout the years at Georgia Tech. Thank you, all members of the extended CS gang, including Amy Wing, Aparna Bajaj, Brigitte Ting, Jeannie Lee, Pallavi Garg, Hasan Abbasi, Leonardo Chen, Nick Diakopoulos, Nikitas Liogkas, Patrick McAndrew, Uk Roho, and my Amis friends William and Esther Clair. Weina Wang, my girlfriend, keeps supporting me, despite everything.

Dr. Peter Csallner and Marianne Csallner, my parents, as well as my sister Gerlinde Csallner have supported me at all times, no matter what I did. Thank you!

TABLE OF CONTENTS

DEDICATION . . . iii

ACKNOWLEDGEMENTS . . . iv

LIST OF TABLES . . . xii

LIST OF FIGURES . . . xiii

SUMMARY . . . xiv

I INTRODUCTION . . . 1

1.1 My thesis . . . 1

1.2 Automatic software testing . . . 1

1.3 Technical positioning . . . 2

1.4 Technical context . . . 4

1.5 Overview and contributions . . . 5

II BACKGROUND AND TERMINOLOGY . . . 7

2.1 Program paths, states, and behavior . . . 8

2.2 Program analysis and its fundamental limitations . . . 8

2.3 Over- and under-approximating program analyses . . . 8

2.4 Sound and complete program analyses . . . 10

2.4.1 Language- and user-level sound bug-finding . . . 12

2.5 Static and dynamic program analyses . . . 14

2.5.1 Common definition . . . 14

2.5.2 Capturing usage data . . . 14

2.5.3 Third definition . . . 15

III THE PROBLEM: FALSE BUG WARNINGS . . . 16

3.1 Path-imprecision in static program analysis tools . . . 16

3.2 Example static program analysis: ESC/Java . . . 17

3.3 Investigated paths: Exceptions and errors in Java . . . 19

3.4 Path-precise analysis and automated bug finding 21
3.5 Problem 1: Language-level path-imprecision 23
3.5.1 Intra-procedurality 23
3.5.2 Floating point arithmetic 24
3.5.3 Big Integers 24
3.5.4 Multiplication 24
3.5.5 Reflection 25
3.5.6 Aliasing and data-flow incompleteness 25
3.6 Problem 2: User-level path-imprecision 25
3.7 Problem 3: Results can be hard to understand 26
IV THE JCRASHER AUTOMATIC RANDOM TESTING SYSTEM . . . 28
4.1 Overview 29
4.2 Test-case generation 31
4.2.1 Parameter-graph 32
4.2.2 Test-case selection 34
4.3 Runtime support 35
4.3.1 Test-case execution and exception filtering 35
4.3.2 Grouping similar exception together 36
4.4 Experience 36
4.4.1 Pragmatics 37
4.4.2 Use of JCrasher 37
4.4.3 Raytrace benchmark 38
4.4.4 Student Homeworks 42
4.4.5 UB-Stack 44
V SOLUTION BACK-END: BUG WARNINGS TO TEST CASES 45
5.1 Overview 45
5.2 Benefits relative to ESC/Java 47
5.2.1 Guaranteeing language-level soundness 47

5.2.2 Improving the clarity of reports 47
5.3 Example 47
5.4 Structure 49
5.5 Constraint solving 50
5.5.1 Primitive types: Integers 50
5.5.2 Complex types: Objects 51
5.5.3 Complex types: Arrays 52
5.6 Benefits relative to JCrasher 54
5.7 Experience 57
5.7.1 JABA and JBoss JMS 57
5.7.2 Check 'n' Crash usage and critique 60
VI SOLUTION FRONT-END: OBSERVE USAGE INVARIANTS TO FOCUS THE BUG SEARCH 63
6.1 Overview 63
6.2 Dynamic-static-dynamic analysis pipeline 66
6.3 Tool background: Daikon 68
6.4 Design and scope of DSD-Crasher 68
6.4.1 Treatment of inferred invariants as assumptions or requirements 70
6.4.2 Inferred invariants excluded from being used 71
6.4.3 Adaptation and improvement of tools being integrated . . . 71
6.5 Benefits 72
6.6 Metrics for evaluating dynamic-static hybrid tools 73
6.6.1 Formal specifications and non-standard test-suites 74
6.6.2 Coverage 75
6.6.3 Invariant precision and recall 76
6.6.4 Goals 77
6.7 Experience 78
6.7.1 More precise than the static-dynamic Check 'n' Crash . . . 79
6.7.2 More efficient than the dynamic-dynamic Eclat 81

6.7.3 Summary of benefits . . . 84
6.8 Applicability and limitations . . . 85
6.8.1 Test suite . . . 85
6.8.2 Scalability . . . 85
6.8.3 Kinds of bugs caught . . . 86
VII ADDING BEHAVIORAL SUBTYPING TO USAGE-OBSERVING PROGRAM ANALYSIS . . . 89
7.1 Overview . . . 89
7.2 The Java modeling language JML and behavioral subtying . . . 90
7.3 Problem 4: Need to support behavioral subtyping in dynamic invariant detection . . . 92
7.4 Solution design . . . 95
7.5 Solution implementation . . . 96
7.5.1 Keeping track of static receivers . . . 97
7.5.2 Checking invariants during execution . . . 98
VIII RESETTING CLASS STATE BETWEEN TEST EXECUTIONS . . . 100
8.1 Overview . . . 100
8.2 Tool background: JUnit . . . 101
8.3 Problem 5: Need to reset static state during long-running test executions . . . 102
8.4 First solution: Multiple class loaders . . . 103
8.5 Second solution: Load-time class re-initialization . . . 105
8.6 Experience . . . 108
IX LESSONS LEARNT AND FUTURE WORK . . . 110
9.1 Design trade-offs . . . 110
9.1.1 Low-level interfaces between analysis stages . . . 110
9.1.2 Generating test cases inside the Simplify automated theorem prover . . . 112
9.1.3 Integration into an integrated development environment . . . 113
9.2 Applying this work to related contexts . . . 115

9.2.1 Program properties of interest beyond runtime exceptions . 115
9.2.2 Program analysis: From ESC/Java to FindBugs and Java PathFinder 116
9.2.3 Programming language: From Java to C# and C++ 119
9.3 Critical review of the performed evaluation 120
9.3.1 Evaluation on large subjects 121
9.3.2 Definite results for medium-sized or large subjects 121
9.3.3 Standard benchmark 122
9.3.4 Bug finding competition 123
X RELATED WORK 125
10.1 Improving path precision at the language level 125
10.2 Improving path precision at the language level and the user level . 128
10.3 Component analyses 129
10.3.1 Inferring specifications to improve user-level path precision . 129
10.3.2 The core bug search component: Overapproximating analysis for bug-finding 130
10.3.3 Finding feasible executions 131
XI CONCLUSIONS 133
APPENDIX A EXAMPLE OUTPUT OF ESC/JAVA 135
REFERENCES 139

LIST OF TABLES

1 JCrasher results on several smaller testees 39

2 JCrasher's resource consumption in experiments 39

3 JCrasher and Check 'n' Crash results on several smaller testees 55

4 Check 'n' Crash results on JABA and JBoss JMS 58

5 DSD-Crasher and Check 'n' Crash results on Groovy 80

6 DSD-Crasher and Eclat results on JBoss JMS 82

7 DSD-Crasher and Eclat results on Groovy 83

8 Experience with SIR subjects 87

9 Overview of the performed evaluations 121

LIST OF FIGURES

1 Program analysis classification via precision and recall 9

2 Overview of the Java exception terminology 19

3 Program analysis classification via user- and language-level path-precision 26

4 JCrasher workflow: From Java testee source file input to test case source file and exception report output 30

5 JCrasher's internal representation of the potential test cases for a method under test 33

6 JCrasher calculating the maximum number of test cases it can generate for a method under test 35

7 Check 'n' Crash workflow: From Java testee source file input to test case source file and exception report output 46

8 Overview of DSD-Crasher's three-stage program analysis in terms of program values and execution paths 66

9 DSD-Crasher workflow: From Java testee and test case source file input to test case source file and exception report output 69

10 Overview of the proposed two-stage algorithm for dynamic invariant inference 95

11 Static receiver problem: The static receiver type is not readily available to the dynamic receiver of a method call 97

12 Static receiver solution: Encode the static receiver type in the method name 98

13 Overview of JUnit's internal data-structures at runtime 102

14 Using multiple class loaders to undo class state change 103

15 Adapted execution model for using multiple class loaders 104

16 DSD-Crasher design overview 111

17 Integration into the Eclipse IDE 114

SUMMARY

This dissertation attacks the well-known problem of path-imprecision in static program analysis. Our starting point is an existing static program analysis that over-approximates the execution paths of the analyzed program. We then make this over-approximating program analysis more precise for automatic testing in an object-oriented programming language. We achieve this by combining the over-approximating program analysis with usage-observing and under-approximating analyses. More specifically, we make the following contributions.

We present a technique to eliminate language-level unsound bug warnings produced by an execution-path-over-approximating analysis for object-oriented programs that is based on the weakest precondition calculus. Our technique post-processes the results of the over-approximating analysis by solving the produced constraint systems and generating and executing concrete test-cases that satisfy the given constraint systems. Only test-cases that confirm the results of the over-approximating static analysis are presented to the user. This technique has the important side-benefit of making the results of a weakest-precondition based static analysis easier to understand for human consumers. We show examples from our experiments that visually demonstrate the difference between hundreds of complicated constraints and a simple corresponding JUnit test-case.

Besides eliminating language-level unsound bug warnings, we present an additional technique that also addresses user-level unsound bug warnings. This technique pre-processes the testee with a dynamic analysis that takes advantage of actual user data. It annotates the testee with the knowledge obtained from this pre-processing step and thereby provides guidance for the over-approximating analysis.

We also present an improvement to dynamic invariant detection for object-oriented programming languages. Previous approaches do not take behavioral subtyping into account and therefore may produce inconsistent results, which can throw off automated analyses such as the ones we are performing for bug-finding.

Finally, we address the problem of unwanted dependencies between test-cases caused by global state. We present two techniques for efficiently re-initializing global state between test-case executions and discuss their trade-offs.

We have implemented the above techniques in the JCrasher, Check 'n' Crash, and DSD-Crasher tools and present initial experience in using them for automated bug finding in real-world Java programs.

CHAPTER I

INTRODUCTION

1.1 *My thesis*

> An existing static program analysis that over-approximates the execution paths of the analyzed program can be made more precise for automatic testing in an object-oriented programming language—by combining the over-approximating analysis with usage-observing and under-approximating analyses.

1.2 *Automatic software testing*

The software executing around us contains bugs and some pose grave risks to the public. For example, software bugs continue to kill humans.[1] And the situation is getting worse, because we will rely more on software in the future. Software will manage all aspects of our lives: our military, transportation systems, general elections, and even our own bodies. Software will continue to contain bugs, and some will cause disaster.

Many programs have an infinite number of potential internal execution states, and bugs are often restricted to very specific execution states. Finding bugs is like finding a few needles in a haystack of possible program executions. Relatively little research has focused on finding individual bugs. Instead, much work has centered around guaranteeing their absence from all possible program executions, mostly using program analyses that over-approximate the set of bugs. Compilers and type-systems are

[1] The Forum On Risks To The Public In Computers And Related Systems is collecting examples at http://catless.ncl.ac.uk/Risks/

examples of over-approximating analyses. But due to fundamental decidability limitations, many bugs in real-world software will remain out of reach of such guarantee-based approaches. (If guarantee-based approaches were to catch all bugs, guarantee-based approaches would also produce too many false bug warnings to remain useful in practice.) Meanwhile, industry (and thereby society as a whole) is spending a lot of resources on finding individual bugs that are out of reach of guarantee-based approaches. The ensuing bug-finding activities are often manual and ad-hoc, and there is little research on finding individual bugs systematically and efficiently.

This dissertation is part of a body of work that tries to advance the state of the art in automatic bug-finding or automatic testing. Automatic testing employs program analysis to automatically explore specific program paths. Testing, or bug-finding, is interested in program paths that lead to a bug. But it seems easy to transfer the employed analyses to different applications, which may be interested in program paths that lead to some other program state or property.

1.3 Technical positioning

Following is the technical reasoning for picking the thesis stated in Section 1.1. This should give an easy technical introduction into the nature of this work.

- "Existing" restricts our scope in a subtle way. As opposed to conducting a purely theoretical investigation, we implement our solution in a realistic setting. This leads us to interesting technical problems in existing analyses, which we address with novel algorithms for dynamic invariant detection and efficiently re-initializing global program state between test case executions.

- "Static program analysis that over-approximates the execution paths of the analyzed program" can be read in the narrow sense of static, compiler-like program analyses that make conservative assumptions about the feasibility of program

execution paths. But the core problems of these analyses, falsely claiming infeasible paths as feasible and producing hard to read reports, are also found in many other analyses such as model-checking, which may not be strictly over-approximating and may execute the analyzed program concretely or symbolically. By addressing these common problems, our approach should equally apply to this broader interpretation.

- "More precise" can be read as relating to each of the two problems of static analysis that over-approximates execution paths. The first interpretation refers to the rate of infeasible paths the analysis falsely claims as feasible. The second interpretation has a human-computer interaction flavor, since it makes the results of the over-approximating static analysis more concise and easier to understand for human users—by complementing huge abstract constraint systems with concrete and concise test-cases.

- "Automatic" makes clear that we are interested in a fully automated program analysis. This is meant to complement, but not replace, manually crafted test cases. To the contrary, our technique relies on an existing test suite or at least the user manually labeling executions as correct or incorrect.

- "Testing" makes clear that we are analyzing single execution paths to find bugs, as opposed to verification approaches that analyze all paths. Also, verification approaches often have access to formal specifications, which is usually not the case in real-world testing. This poses a significant challenge for testing, in addition to the common challenge of reasoning about program execution paths.

- "Object-oriented" hints at a technical highlight of this work. In contrast to previous work, we provide better support for the following programming language features, which are at the core of object-oriented programming languages and their specification languages.

  - The ability to redefine or provide multiple implementations of a method
  - Subcontracting or behavioral subtyping

  These features, together with subtype polymorphism and inheritance, distinguish many modern object-oriented programming languages from many traditional, procedural programming languages, such as Pascal [108, 61], C [62, 63], and Modula-2 [109, 110].

- "Usage-observing" refers to inferring specifications from actual user data. Many approaches are commonly labeled as specification inferrence, but some do not take actual user data into account. Instead, they produce general, usage-independent program summaries, which are very valuable for many tasks, but do not give us additional constraints on the expected program usage. Observing actual program usage is our substitute for formal specifications, which are usually not available to real-world automatic testing.

1.4 Technical context

We investigate the thesis of Section 1.1 in the context of Java [47, 48], ESC/Java [42], and detecting runtime exceptions. This is a good context for such an investigation because of the practical importance, wide availability, and good documentation of Java, ESC/Java, and runtime exceptions.

- Java [47, 48] is currently one of the most popular object-oriented programming languages, with wide use in industry, teaching, and research.

- The extended static checker for Java (ESC/Java) is a research tool for relatively deep static analysis of Java programs. While not being strictly over-approximating (it also misses feasible paths), ESC/Java shares the major problems of powerful, strictly over-approximating, static analyses—false bug warnings and an overly complicated presentation of bug warnings. ESC/Java is

well-known in the verification and program analysis research communities. For example, as of December 2007, Google Scholar lists over 500 citations to the main ESC/Java paper [42]. ESC/Java is available in binary and source form, and has so far been actively maintained as ESC/Java2 [20]. The ESC/Java authors have published multiple research papers on different aspects and applications of ESC/Java [73, 41, 72, 66], including the underlying constraint-solver and automated theorem-prover Simplify [33].

- Automatic testing is an important application of program analysis, in which the analysis tries to find program execution paths that violate a given program correctness condition. A representative example correctness condition is the absence of certain runtime exceptions, which often indicate program bugs and may result in abrupt program termination ("crash").

Other contexts may be similarily valid and interesting, but it is less likely to gain significant new insight from replicating the core ideas from one context to another.

1.5 Overview and contributions

This dissertation attacks the well-known problem of path-imprecision in static program analysis. Our starting point is an existing static program analysis that over-approximates the execution paths of the analyzed program. We then make this over-approximating program analysis more precise for automatic testing in an object-oriented programming language. We achieve this by combining the over-approximating program analysis with usage-observing and under-approximating analyses. More specifically, we make the following contributions.

1. We present a technique to eliminate language-level unsound bug warnings produced by an execution-path-over-approximating analysis for object-oriented programs that is based on the weakest precondition calculus. The technique post-processes the results of the over-approximating analysis by solving the produced

constraint systems and generating and executing concrete test-cases that satisfy the given constraint systems. Only test-cases that confirm the results of the over-approximating static analysis are presented to the user.

2. Our technique of converting constraint systems to concrete test-cases has the important side-benefit of making the results of a weakest-precondition based static analysis easier to understand for human consumers. We will show examples from our experiments that visually demonstrate the difference between hundreds of complicated constraints and a simple corresponding JUnit test-case.

3. Besides eliminating language-level unsound bug warnings, we present an additional technique that also addresses user-level unsound bug warnings. This technique pre-processes the testee with a dynamic analysis that takes advantage of actual user data. It annotates the testee with the knowledge obtained from this pre-processing step and thereby provides guidance for the over-approximating analysis.

4. We present an improvement to dynamic invariant detection for object-oriented programming languages. Previous approaches do not take behavioral subtyping into account and therefore may produce inconsistent results, which can throw off automated analyses such as the ones we are performing for bug-finding.

5. We address the problem of unwanted dependencies between test-cases caused by global state. We present two techniques for efficiently re-initializing global state between test-case executions and discuss their trade-offs.

6. We have implemented the above techniques in the JCrasher, Check 'n' Crash, and DSD-Crasher tools and present initial experience in using them for automated bug finding in real-world Java programs.

CHAPTER II

BACKGROUND AND TERMINOLOGY

At the heart of most program analyses is reasoning about the feasibility of program states and program execution paths. Automated bug-finding and testing are no exception. We can always derive execution states from knowledge about execution paths, so to keep the terminology short and intuitive, we focus it on execution paths. This chapter aims at clarifying our use of the following program analysis terms.

- "Precise"
- "Over-approximating" and "under-approximating"
- "Sound" and "complete"
- "Language-level" and "user-level"
- "Dynamic" and "static"

Our discussion should not be restricted to our specific investigation in the context of testing object-oriented programs. Instead, we hope to provide a guide for terms commonly used in program analysis. When defining a term we try to be careful to qualify the defined term with our context of analyzing individual execution paths (which is useful for testing). This should clearly contrast our definitions from often inverse definitions in the context of analyzing all paths (which is common in the verification community). After defining the terms we will often omit the explicit single-path qualification from the text, which then should always be read with an implicit qualification to our testing context in mind.

2.1 Program paths, states, and behavior

In this section we clarify our usage of the terms program paths, program states, and program behavior.

2.2 Program analysis and its fundamental limitations

Recall that general-purpose programming languages are generally Turing-complete, which makes the halting-problem undecidable for them. All mainstream, imperative, functional, and object-oriented languages are in this category (such as C, Pascal, Lisp, ML, Haskell, C++, C#, and Java). For programs written in such languages, there can be no analysis that determines all feasible execution paths or states, without marking infeasible ones as feasible. But this does not mean we should give up on program analysis. To the contrary, there is a large number of valid and interesting trade-offs that program analyses can implement to achieve different goals.

2.3 Over- and under-approximating program analyses

Due to the fundamental decidability limitations discussed in Section 2.2, program analyses are really approximation algorithms (for a theoretical exposition, see the rich literature on abstract interpretation [21, 85, 49, 50]). They typically approximate the set of feasible program execution paths or the set of feasible program states. Path feasibility can simulate many additional criteria of interest. E.g., we could translate many program correctness conditions directly to additional program assertions. So depending on the context we can read "feasible" as "relevant", "specified", or "of interest". We adopt the standard precision and recall definitions from the information retrieval community to classify program analyses. With this basic terminology in hand we can then easily distinguish over- and under-approximating program analyses. Following is our precision metric, which, given our testing background, we define as the precision of reported paths being feasible. The paths reported by a more precise

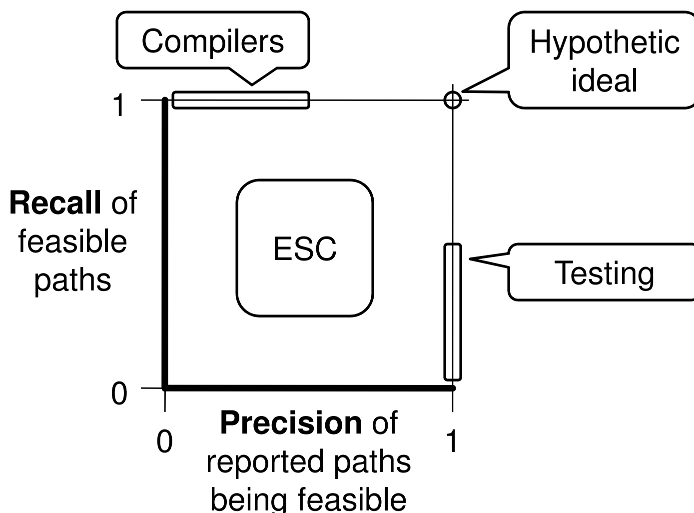


Figure 1: Program analysis classification via precision and recall. Different program analyses make different trade-offs between reporting many feasible paths (high recall) and reporting few infeasible ones (high precision). Compilers, testing, and the extended static checker ESC are three prominent examples for different trade-offs in this space.

analysis are more likely to be feasible.

Definition (Path precision)**.** *"(feasible ∩ reported) / reported" execution paths*

For example, a 100% precise analysis ensures that each path it reports is feasible. Such an analysis reports a subset of the goal set of exactly the feasible paths, which we call under-approximating, short for under-approximating the set of feasible paths.

Definition (Under-approximating)**.** *precision = 1*

Testing is the classic example of a 100% precise program analysis; executing the analyzed program makes it easy for the analysis to ensure that every path reported to be feasible really is feasible. On the other hand, precision does not talk about how many feasible paths the analysis reports. To capture "how many" we use our following recall metric, short for recall of feasible execution paths. An analysis with a higher recall reports more of the feasible paths.

Definition (Path recall). *"(feasible $\cap$ reported) / feasible" execution paths*

For example, an analysis that has 100% recall reports all feasible paths, and may report additional non-feasible ones. It therefore reports a superset of our goal set of exactly the feasible paths. We call such an analysis over-approximating, short for over-approximating the set of feasible paths.

Definition (Over-approximating). *recall = 1*

Compilers are the classic example of program analysis that has 100% recall. Compilers do not miss any feasible execution path but may include paths that are not feasible in their reasoning, and hence claim infeasible ones as feasible, which leads to false bug warnings in our context.

Figure 1 illustrates that precision and recall are orthogonal, i.e., they are not the opposite of each other. The unrealistic, perfect program analysis is in the right upper corner, at the intersection of 100% precision and 100% recall, which makes it both under- and over-approximating. While many analyses are either over- or under-approximating, there is a huge area between these two extremes. These analyses are neither over- nor under-approximating, which means that they combine the conceptual problems of both, they may miss feasible paths and report infeasible ones as feasible. ESC/Java is an example analysis. Note that such analyses are not necessarily bad, since, depending on the task at hand, a small loss of precision may be tolerable in exchange for higher recall, and vice versa.

2.4 Sound and complete program analyses

The terms "sound" and "complete" are commonly used in logic, verification, and program analysis to denote the two extremes of a program analysis spectrum. The "sound" extreme is often intuitively considered to be the most worthy location in the spectrum. But many well-respected analyses readily admit to be "unsound".

ESC/Java, for example, is "unsound" and "incomplete" and therefore somewhere in between the two extremes.

Compared to the traditional verification and program analysis literature, we are using a non-standard definition of sound and complete. Much of the previous work, including the main ESC/Java paper [42], defines sound and complete relative to a correctness condition and the entire program, meaning *all* paths of the programs. Instead, we define sound and complete relative to a predicate and a *single* path. All paths are a good fit for proving program correctness, but our focus is program incorrectness. To prove a program incorrect, we are looking for a counterexample, a single program path that satisfies the negation of a verification condition. Switching the focus from all paths to one path leads us to the opposite of the definitions used by ESC/Java and much prior work. I.e., whenever we refer to a sound system, we mean a complete correctness proving system, and whenever we refer to a complete system we mean a sound correctness proving system.

In mathematical logic, a sound inference system is powerful enough that each statement it infers is true. Whereas a complete system is powerful enough that it could infer any true statement. A sound system may be incomplete and therefore fail to infer some true statements. A complete system may be unsound and therefore able to infer false statements (besides all true ones).[1] The following definitions summarize these notions for a given statement or predicate in our testing context, where we are interested in single paths that satisfy a given bug property, such as throwing a runtime exception.

Definition (Path sound)**.** *can_infer(predicate, path)* $\Rightarrow$ *path satisfies predicate*

[1] An inference system in mathematical logic is really just a set of derivation rules (such as the sequent calculus). Mathematical logic is mainly concerned with the power of these rules and does not tell us in which order to apply rules in order to derive a given fact. But we can think of a simple breadth-first search approach that will eventually enumerate all derivable facts, and thereby implement a slow derivation algorithm that looks more like the program analyses we are used to.

Definition (Path complete). *path satisfies predicate ⇒ can_infer(predicate, path)*

This makes sound a synonym for 100% precision and for under-approximating, and unsound a synonym for being less precise. Similarily, complete becomes a synonym for 100% recall and for over-approximating, and incomplete a synonym for having less than 100% recall.

2.4.1 Language- and user-level sound bug-finding

In practice, there are two levels of soundness for bug-finding. The lower level is being sound with respect to the execution semantics. This means we only ask questions or predicates over the basic feasibility of a path with respect to the execution semantics of the implementation language such as Java. But we do not encode any user constraints such as user specifications. This means that a language-level sound bug report corresponds to a feasible execution path of a program module, although the input that caused this execution may not be one that would arise in normal program runs. We call this language-level soundness because it can be decided by checking the language specification alone. Many bug-finding approaches concern themselves only with this soundness level, and several of them do not achieve it. A stronger form of soundness consists of also being sound with respect to the intended usage of the program. We call this user-level soundness, as it means that a bug report will be relevant to a real user of the program. This is an important distinction because developers have to prioritize their energy on the cases that matter most to their users. From their perspective, a language-level sound but user-level unsound bug report may be as annoying as one that is unsound at the language level.

Language-level soundness is the lower bar for bug-finding analysis. An analysis that is unsound with respect to execution semantics may flag execution paths that can never occur, under any inputs or circumstances. ESC/Java uses such an analysis. In the absence of pre-conditions and post-conditions describing the assumptions and

effects of called methods, ESC/Java analyzes each method in isolation without taking the semantics of other methods into account. For instance, in the following example, ESC/Java will report potential errors for `get0() < 0` and `get0() > 0`, although neither of these conditions can be true.

```
public int get0() {return 0;}


public int meth() {
  int[] a = new int[1];
  return a[get0()];
}
```

A user-level sound analysis has to satisfy not only language semantics but also user-level specifications. Thus, user-level soundness is generally impossible to achieve for automated tools since user-level specifications are mostly informal. Common forms of user-level specifications are code comments, emails, or web pages describing the program. Often these informal specifications only exist in the developers' minds. It is clear that user-level soundness implies language-level soundness, since the users care only about bugs that can occur in real program executions. So the user-level sound bug reports are a subset of the language-level sound bug reports.

ESC/Java may produce spurious error reports that do not correspond to actual program usage. For instance, a method `forPositiveInt(int i)` under test may be throwing an exception if passed a negative number as an argument. Even if ESC/Java manages to produce a language-level sound warning about this exception it cannot tell if this case will ever occur in practice. A negative number may never be passed as input to the method in the course of execution of the program, under any user input and circumstances. That is, an implicit precondition that the programmer has been careful to respect makes the language-level sound warning unsound at the user-level.

2.5 *Static and dynamic program analyses*

2.5.1 Common definition

The common usage of the terms dynamic and static is to distinguish testee-executing and testee-non-executing program analyses. For example, the definition in the popular Wikipedia archive states that: [2]

> "Static code analysis is the analysis of computer software that is performed without actually executing programs built from that software (analysis performed on executing programs is known as dynamic analysis)."

We adopt this common terminology to avoid confusion, but not without pointing out conflicting usage of these terms.

2.5.2 Capturing usage data

The standard definition of Section 2.5.1 is not quite satisfying, since it does not capture the core benefit of executing a program, that is, precisely observing the program paths executed by actual users. It may therefore be useful to distinguish analyses based on the data they take into account during their reasoning. Many analyses execute the program they are analyzing and are therefore dynamic according to standard terminology, but they only invoke the analyzed program to pass it some pre-defined, hard-coded data, such as -1, 0, and 1. This may remind us of compile-time or static constants, in the sense that these analyses ignore which specific usage scenarios are important to actual users.

Compilers are the classic example of static analysis, both in the common usage and in the sense of ignoring specific uses of the analyzed program. Compilers make general judgements about a program that hold for all possible uses. Testing is the classic example of dynamic analysis, in both contexts, since it usually concentrates on

[2] http://en.wikipedia.org/wiki/Static_code_analysis as of December 2007

the execution paths that are important to the user, and thereby takes user constraints into account.

2.5.3 Third definition

There is a third interpretation of static and dynamic, which is due to Jackson and Rinard [59]. They tie static to analyzing all paths and dynamic to analyzing individual paths.

> "Sound static analyses produce information that is guaranteed to hold on all program executions; sound dynamic analyses produce information that is guaranteed to hold for the analyzed execution alone."

Interestingly, all three definitions are orthogonal to each other, and therefore seem to capture interesting program analysis properties.

CHAPTER III

THE PROBLEM: FALSE BUG WARNINGS

This chapter describes the first set of problems we want to investigate. These problems center around the path-imprecision of static program analyses, as defined in Section 2.3. We are interested in three facets of this problem: language-level imprecision, user-level imprecision, and complicated reports. As motivated in Chapter 1, we investigate these problems in the context of ESC/Java and finding runtime exceptions, which is often used to find bugs.[1] Related contexts may involve other advanced programming languages, other kinds of offending behavior, and other static analyses. We briefly summarize the relevant behavior of ESC/Java and give examples of the problems investigated. While investigating these issues in the following Chapter 5 and Chapter 6, we will realize two additional problems, which are also not restricted to our concrete setting of ESC/Java and finding runtime exceptions. We will discuss these additional problems and our solutions in Chapter 7 and Chapter 8.

3.1 Path-imprecision in static program analysis tools

The path imprecision problem of static analyses has been reported in several studies [91, 119, 105]. For example, Rutar et al. [91] examine ESC/Java2, among other analysis tools, and conclude that it can produce many spurious warnings when used without context information (method annotations). One specific problem, which we revisit in Chapter 6, is that of ESC/Java's numerous `NullPointerException` warnings. For five testees with a total of some 170 thousand non-commented source statements, ESC/Java warns of a possible null dereference over nine thousand times.

[1] We use the terms "fault", "error", and "bug" interchangeably, similarly the terms "report" and "warning".

Rutar et al., thus, conclude that "there are too many warnings to be easily useful by themselves."

The surveys of automated bug-finding tools conducted by Zitser et al. [119] and Wagner et al. [105] also concur with our estimate that an important problem is not just reporting potential errors, but minimizing false bug warnings so that inspection by humans is feasible. [119] evaluate five static analysis tools on 14 known buffer overflow bugs. They found that the tool with the highest detection rate (PolySpace) suffered from one false alarm per twelve lines of code. They conclude "[..] that while state-of-the-art static analysis tools like Splint and PolySpace can find real buffer overflows with security implications, warning rates are unacceptably high." [105] evaluates three automatic bug finding tools for Java (FindBugs [57], PMD, and QJ Pro). They conclude that "as on average two thirds of the warnings are false positives, the human effort could be even higher when using bug finding tools because each warning has to be checked to decide on the relevance of the warning." [91] evaluates five tools for finding bugs in Java programs, including ESC/Java2, FindBugs, and JLint. The number of reports differs widely between the tools. For example, ESC/Java2 reported over 500 times more possible null dereferences than FindBugs, 20 times more than JLint, and six times more array bounds violations than JLint. Overall, Rutar et al. conclude: "The main difficulty in using the tools is simply the quantity of output."

3.2 Example static program analysis: ESC/Java

The Extended Static Checker for Java (ESC/Java) [42] is our representative static program checker. It searches the analyzed code for potential invariant violations. ESC/Java compiles the Java source code under test to a set of predicate logic formulae. ESC/Java checks each method `m` in isolation, expressing as logic formulae the properties of the class to which the method belongs, as well as Java semantics. Each method call or invocation of a primitive Java operation in `m`'s body is translated to

a check of the called entity's precondition followed by assuming the entity's post-condition. ESC/Java recognizes invariants stated in the Java Modeling Language (JML) [71]. (We consider the ESC/Java2 system [20]—an evolved version of the original ESC/Java, which supports Java 1.4 and JML specifications.) In addition to the explicitly stated invariants, ESC/Java knows the implicit pre- and postconditions of primitive Java operations—for example, array access, pointer dereference, class cast, or division. Violating these implicit preconditions means accessing an array out-of-bounds, dereferencing null pointers, mis-casting an object, dividing by zero, etc. ESC/Java uses the Simplify theorem prover [33] to derive error conditions for a method. We use ESC/Java to derive abstract conditions under which the execution of a method under test may terminate abnormally. Abnormal termination means that the method would throw a runtime exception because it violated the precondition of a primitive Java operation. In many cases this will lead to a program crash as few Java programs catch and recover from unexpected runtime exceptions.

Like many other static analysis based bug finding systems, ESC/Java is language-level unsound (and therefore also user-level unsound): it can produce spurious error reports because of inaccurate modeling of the Java semantics. ESC/Java is also incomplete: it may miss some errors—for example, because ESC/Java ignores all iterations of a loop beyond a fixed limit. Being incomplete, ESC/Java is therefore not over-approximating, strictly speaking, according to our definition in Section 2.3. While this shortcoming of ESC/Java may be problematic for real-world use, it is less of a problem for our investigation, since ESC/Java shares the conceptual problem of over-approximating analyses we want to address: path-imprecision and the resulting false bug warnings. The ESC/Java authors assess the problem of "warnings about non-bugs" (false bug warnings) produced by their ESC/Java tool as follows:

> "[T]he tool has not reached the desired level of cost effectiveness. In particular, users complain about an annotation burden that is perceived

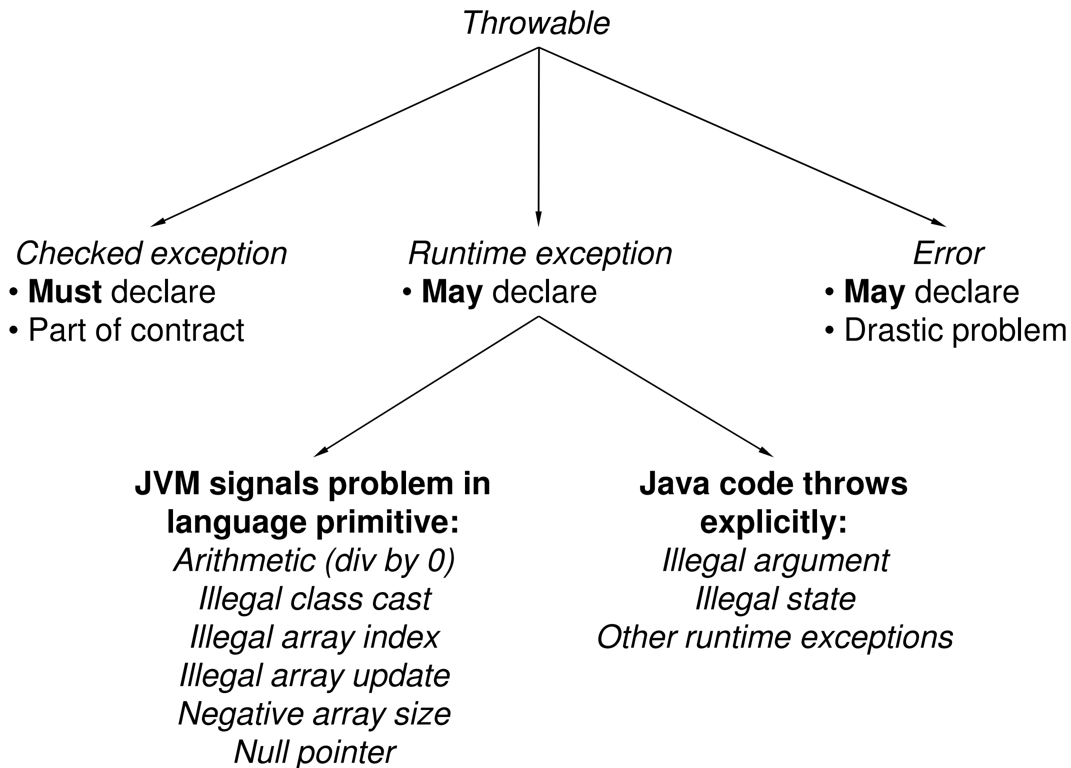


Figure 2: Overview of the Java exception terminology. An arrow is drawn from super-type to direct or transitive subtype. `Error`, (checked) `Exception`, `RuntimeException`, and the depicted subclasses of `RuntimeException` are, like `Throwable`, found in package `java.lang`. JVM is a Java virtual machine.

> to be heavy, and about excessive warnings about non-bugs, particularly on unannotated or partially-annotated programs."

3.3 Investigated paths: Exceptions and errors in Java

While our observation of imprecision and our solution concepts apply to static analysis in general, for this investigation we concentrate on a path subset. We focus on paths that lead to runtime exceptions, which in Java often indicates a potential bug in the caller, callee, or the specification. In this section we review the Java exception terminology to avoid confusions with similar terminology used in other contexts.

Figure 2 illustrates the Java exception terminology, which is organized as a subtype hierarchy. Each exception type is a subtype of `Throwable`. At a high level,

Java distinguishes serious problems the user is not supposed to handle (`Error`) and problems the user is supposed to handle (`Exception`).

- Java suggests throwing an `Error` in case of a serious problem. For example, the Java virtual machine throws an `OutOfMemoryError` if it runs out of memory. Many of these problems are only indirectly related to the currently executing code, and often relate more to the particular execution environment (such as the virtual machine being restricted to a small amount of heap memory). Most applications are not supposed to catch errors, and ESC/Java does not search for potential errors.

- *Checked exceptions* can only be thrown or passed on by methods or constructors that declare to throw them. While checked exceptions are important in practice, we do not concern ourselves with them in this investigation.

- *Unchecked exceptions* are of type `RuntimeException`. Each method and constructor may throw an unchecked exception, regardless of whether this has been declared in the method's signature or not. There seem to be the following two kinds of unchecked exceptions.

- An exception of the *first group of unchecked exceptions* is typically thrown by low-level functions implemented by the Java virtual machine (and we ignore cases in which some user code explicitly throws an exception from this group.) For primitive language operations no user method is called and therefore no function frame is put onto the execution stack. Therefore the user method providing the wrong precondition is on top of the execution stack. These runtime exceptions often indicate a bug in the caller, who triggered this exception in the Java virtual machine. ESC/Java searches for these runtime exceptions and we concentrate our investigation on them.

- An exception from the *second group of unchecked exceptions* usually indicates that the precondition of the called method or constructor has been violated. These runtime exceptions are explicitly thrown by regular Java code. Such exceptions include `IllegalArgumentException` and `IllegalStateException`. Here the method whose preconditions have been violated is the top stack element. For this investigation, we do not concern ourselves with such explicitly thrown exceptions.

3.4 Path-precise analysis and automated bug finding

Testing and automated bug-finding are very popular applications of path-precise program analysis. Clearly, testing is very important in practice, as briefly described in Section 1.2. While having a path-precise analysis is good for testing, it is certainly not enough. More important is knowing what constitutes a bug and what not, ideally in the form of a formal specification. A big problem, which we cannot solve either, is that real-world testing usually has no access to formal specifications. If there is any specification, it is often incomplete or may contain errors itself. In Chapter 6 we will try to mitigate this problem by inferring specification from existing test cases, but this can only be a best-effort heuristic. We therefore cannot reach our ultimate goal in automated testing, which is a *fully automated* tool for *modern object-oriented* languages that finds *bugs* but produces *no false bug warnings*. A fully automated bug finding tool should require zero interaction with the software developer using the tool. In particular, using the bug-finding tool should not require any manual efforts to write additional specifications or test cases. The tool should also not require manual inspection of the produced bug warnings.

Initially, no program behavior constitutes a bug. Only specifications (implicit or explicit) allow us to distinguish expected and buggy behavior. Implicit specifications are common. For example, program comments typically consist of informally stated

pre- and postconditions. A common precondition of object-oriented programs is that null is never a legal input value, unless explicitly stated in a code comment. A common postcondition is that a public method should not terminate by throwing a class cast exception. Beyond such general properties, most specifications are very specific, capturing the intended semantics of a given method. In the following we use *pre* and *post* when referring to satisfying pre- and postcondition, respectively.

1. *pre AND post* is the specified behavior: the method is working in the intended input domain as expected by the postcondition.

2. *pre AND NOT post* is a bug: the method deviates from the expected behavior in the intended input domain.

3. *NOT pre* is a false bug report, since it reports behavior outside the intended input domain of the method.

For our investigation, we treat every public method as a library method, that is, we assume it can be called in any context. This assumption may be too liberal for production use of our solutions as automated bug-finding tools, but it should suffice for this investigation. This means that we consider an input to be valid if manual inspection reveals no program comments prohibiting it, if invariants of the immediately surrounding program context (e.g., class invariants) do not disqualify the input, and if program values produced during actual execution seem (to the human inspector) consistent with the input. We do not, however, try to confirm the validity of a method's input by producing whole-program inputs that give rise to it. In other words, we consider the *program as a library*: We assume that its public methods can be called for any values not specifically disallowed, as opposed to only values that can arise during whole-program executions with valid inputs to the program's `main` method. This view of "program as library" is common in modern development environments, especially in the Java or .Net world, where code can be

dynamically loaded and executed in a different environment. Coding guidelines for object-oriented languages often emphasize that public methods are an interface to the world and should minimize the assumptions on their usage.[2] Furthermore, this view is convenient for experimentation, as it lets us use modules out of large software packages, without worrying about the scalability of analyzing (either automatically or by hand) the entire executable program. Finally, the per-method checking is defensive enough to withstand most changes in the assumptions of how a class is used, or what are valid whole-program inputs. Over time such assumptions are likely to change, while the actual method implementation stays the same. Examples include reusing a module as part of a new program or considering more liberal external environments: buffer overflows were promoted in the last two decades from obscure corner case to mission-critical bugs.

3.5 Problem 1: Language-level path-imprecision

Following we list some of the sources of language-level path imprecision, as exhibited by ESC/Java. These cases are typical, in the sense that increasing precision in these cases would often require prohibitively expensive reasoning.

3.5.1 Intra-procedurality

In the absence of pre-conditions and post-conditions describing the assumptions and effects of called methods, ESC/Java analyzes each method in isolation without taking the semantics of other methods into account. In the following example, ESC/Java will report potential errors for `get0() < 0` and `get0() > 0`, although neither of these conditions can be true.

[2]For instance, Code Complete 2 [77, chapter 8] states "The class's public methods assume the data is unsafe, and they are responsible for checking the data and sanitizing it. Once the data has been accepted by the class's public methods, the class's private methods can assume the data is safe." Similarly, Meyer [79, chapter 23] explicitly bases his guidelines of class design on the design of libraries.

```
public int get0() {return 0;}

public int meth() {
  int[] a = new int[1];
  return a[get0()];
}
```

3.5.2 Floating point arithmetic

ESC/Java does not have good handling of floating point values. Consider the following simple example method.

```
int meth(int i) {
  int res = 0;
  if (1.0 == 2.0) res = 1/i;
  return res;
}
```

ESC/Java will produce a spurious error report suggesting that with `i == 0` this method will throw a divide-by-zero exception. If integer constants (i.e., `1`, `2` instead of `1.0`, `2.0`) had been used, no error would have been reported.

3.5.3 Big Integers

ESC/Java has similar imprecisions with respect to big integer numbers. Big integers are represented as symbols and the theorem prover cannot infer that, for instance, the following holds: `1000001 + 1000001 != 2000000`.

3.5.4 Multiplication

ESC/Java has no built-in semantics for integer multiplication. For input variables `i` and `j`, ESC/Java will report spurious errors (division-by-zero) both for the line:

```
if ((i == 1) && (j == 2)) res = 1/(i*j);
```

and also for:

```
if ((i != 0) && (j != 0)) res = 1/(i*j);
```

Note that the latter case is interesting because the possibility of error cannot be eliminated by testing only a small number of values for `i` and `j`. This is one example where generating a large number of test cases automatically with JCrasher can increase the confidence that the error report is indeed spurious.

3.5.5 Reflection

ESC/Java does not attempt to model any Java reflection properties, even for constant values. For instance, the following statement will produce a spurious error report for division-by-zero with input `i` being 0:

```
if (int.class != Integer.TYPE) res=1/i;
```

3.5.6 Aliasing and data-flow incompleteness

ESC/Java models reference assignments incompletely—e.g., type information is lost for elements stored in arrays. For classes `A` and `B`, with `B` a subclass of `A`, the following code will produce a spurious error report (for a class-cast-exception error):

```
A[] arr = new A[]{new B(), new A()};
B b = (B) arr[0];
```

3.6 Problem 2: User-level path-imprecision

Section 2.4.1 has already described the problem of user-level path-imprecision. We would just like to recall that user-level precision is strictly stronger and harder than language-level precision. It is stronger since user-level precision implies language-level precision, but not vice versa. It is harder to obtain as the criteria for user-level

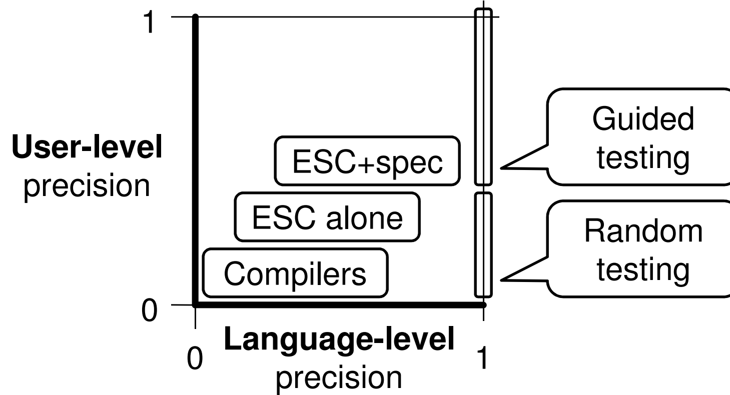


Figure 3: Program analysis classification via user- and language-level path-precision. Like the related precision-recall classification in Figure 1, this figure does not carry quantitative results, but should only provide an approximate, qualitative overview.

precision are seldomly available in real-world settings, i.e., for testing. To address this additional problem, we must revert to heuristic or other means of obtaining user specifications. Figure 3 summarizes our observations.

3.7 Problem 3: Results can be hard to understand

When an error is reported by a program analysis system, it is generally desirable to see not just where the error occurred but also an example showing the error conditions. Musuvathi and Engler [82] summarize their experiences (in a slightly different domain) as:

> "A surprise for us from our static analysis work was just how important ease-of-inspection is. Errors that are too hard to inspect might as well not be flagged since the user will ignore them (and, for good measure, may ignore other errors on general principle)."

ESC/Java can optionally emit the counterexamples produced by the Simplify theorem prover. Yet these counterexamples contain just abstract constraints instead of specific values. Furthermore, there are typically hundreds of constraints even for a small method. For instance, for a 15-line method (from one of the programs examined later in Section 5.6) ESC/Java emits a report that begins:

```
P1s1.java:345: Warning: Array index possibly too large (IndexTooBig)
 isTheSame(list[iList+1],pattern[iPatte ...
               ^
Execution trace information:
 Reached top of loop after 0 iterations in
  "P1s1.java", line 339, col 4.
 Executed then branch in "P1s1.java", line 340, col 59.
 Reached top of loop after 0 iterations in
  "P1s1.java", line 341, col 6.


Counterexample context:
 (patternNum@340.9-339.4#0-340.9:360.52 <= intLast)
 (intFirst <= tmp2:342.26)
 (tmp2:342.26 <= intLast)
 (arrayLength(pattern:334.53) <= intLast)
    ...
```

(113 lines follow.) The first few lines are part of the standard ESC/Java report, while the rest describe the counterexample. If the error conditions are clear from the location and path followed to reach the error site, then the report is quite helpful. If, however, the counterexample needs to be consulted, the report is very hard for human inspection. See Appendix A for a full example.

CHAPTER IV

THE JCRASHER AUTOMATIC RANDOM TESTING SYSTEM

This chapter presents JCrasher [23], which is our representative testing tool. Testing implements an extreme trade-off in the program analysis spectrum between precision and recall, trading recall for full precision, as shown in Figure 1 of Chapter 1. We include JCrasher for three reasons. First, it provides a fully precise dynamic baseline analysis against which we can evaluate our solutions for imprecise static analyses in Chapter 5 and Chapter 6. Second, our solutions presented in those chapters re-use several aspects of JCrasher. Finally, evaluating JCrasher will lead us to a conceptual problem in automated test execution, which we will explore in Chapter 8.

For conciseness, we describe here and later use a simplified version of JCrasher, which just covers the aspects that are useful for our investigation. I.e., we do not cover or use the following aspects.

- Taking into account the time allocated for testing
- Heuristics for determining whether a Java exception should be considered a program bug or that the inputs supplied by JCrasher have violated the code's preconditions
- Integration into the Eclipse IDE

These aspects support JCrasher's use as an independent stand-alone testing tool, but are less interesting for our investigation. The full JCrasher version with the above features is still available in the standard JCrasher distribution and is described in the JCrasher paper [23].

Certainly, JCrasher is not the first or only tool for automatic random testing. Several research tools [58, 70, 18, 43, 78] and commercial products [90, 94] employ some of the same ideas in their testing. Nevertheless, JCrasher offers some features that are unique—to different extents compared to different alternatives. JCrasher constructs test-cases at random, through the tested methods' parameter-spaces. Unlike other tools—Enhanced JUnit [94], for example—JCrasher takes the Java type system into account when constructing random inputs, so that well-formed inputs are constructed by chaining program methods. JCrasher produces test files for JUnit [8], which is a popular framework for test automation in the Java language. This has the following three advantages. First, there is significant ease of use and inspection of tests. Second, test plans are reified in code, so that if a developer decides that a test is good enough, he/she can permanently integrate it in a regression test-suite. Third, externalizing generated test-cases eases debugging and development, being important to the tool developer. Tomb et al., in recent related work [101], confirm our argument:

> "[..] JCrasher, for instance, creates external files containing JUnit test cases. In retrospect, JCrasher's approach seems more robust, and we plan to adopt it in the future."

4.1 Overview

JCrasher takes as input a program in Java bytecode form and produces a series of test-cases for the popular JUnit unit test framework. Random testing can be seen as a search activity: JCrasher is searching for inputs that will cause the target program to crash. The search is exhaustive under user-specified or inferred constraints, such as the maximum depth of method chaining. These user-level parameters induce the search space of a specific method. The different points represent value assignments of the formal parameters of the method's signature. We also refer to these values as the method's *parameter-space*.

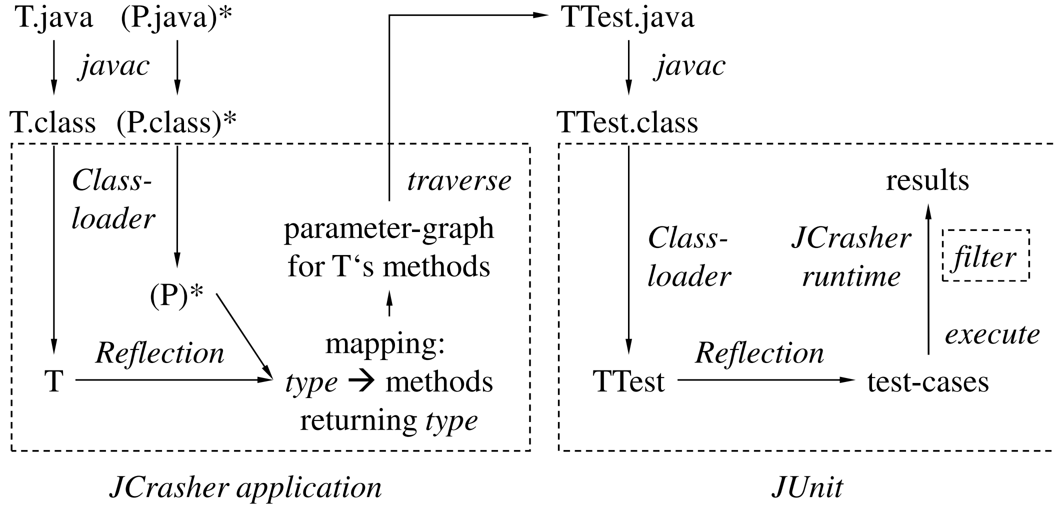


Figure 4: JCrasher workflow: From Java testee source file input to test case source file and exception report output. First, JCrasher generates a range of test-cases for the analyzed class `T` and writes them to `TTest.java`. Second, the test-cases can be executed with JUnit, and third, the JCrasher runtime allows exceptions filtering. The star * applied to some P stands for "zero or more of P".

The representation and traversal of the parameter-space in JCrasher is type-based. To test a method, JCrasher examines the types of the method's parameters and computes the possible ways to produce values of these types. JCrasher keeps an abstract representation of the parameter-space in the form of a mapping from types to either pre-set values of the type or methods that return a value of the type. This mapping is the data structure that lets JCrasher estimate how many different tests it can produce for a certain method.

Figure 4 illustrates the context for which JCrasher was originally developed. The user wants to check a Java class `T.class` for robustness. In a first step he/she invokes the JCrasher application and passes it the name `T` of the class to be tested. The JCrasher class loader reads the `T.class` bytecode from the file system, analyzes it using Java reflection [15], and finds for each of the methods declared by `T` and by their transitive parameter types `P` a range of suitable parameter combinations. It selects some of these combinations and writes these as `TTest.java` back into the file system. After compiling `TTest.java` the test-cases can be executed with JUnit.

Exceptions thrown during test-case execution are caught by the JCrasher runtime. Only exceptions found to violate the robustness heuristic are passed on to JUnit. JUnit collects these exceptions and reports them as errors to the user. The above process generalizes to multiple classes straightforwardly. If the user passes a set of classes as input to JCrasher, the analysis will examine combinations of all their methods in order to construct testing inputs. In this way, the class can be tested within its current application environment instead of being tested in isolation.

By producing JUnit test-cases, JCrasher enables human inspection of the auto-generated tests. Particularly successful auto-generated tests can then be kept as part of a regression test-suite. Additionally, JUnit has an active community and integrating with it enables mutual benefit. For instance, a number of extensions[1] to JUnit already exist.

4.2 Test-case generation

In this section we describe in detail the test-case generation logic of JCrasher. Note that this part of the system has engineering novelty but little conceptual novelty: similar ideas have been used in different settings—scenarios for testing GUIs [78], for example. Nevertheless, this section is necessary for a complete description of the system, and we will re-use the discussed technique for our solutions presented in Chapter 5 and Chapter 6.

For each method `f` declared by a given class `T`, JCrasher generates a random sample of suitable test-cases. Each test-case passes a different parameter combination to `f`. This generation is done by the following steps. First, JCrasher uses Java reflection to identify method parameter types, types returned by methods, subtyping relations and visibility constraints—`private`, for example. JCrasher adds each accessible method to an in-memory data-structure mapping a type to some pre-set values and methods

[1]See JUnit's web-site, `http://www.junit.org`

returning the type. Second, using the above mapping, JCrasher determines for each method `f` how many and which test-cases to generate and writes them as a JUnit test-class `TTest.java` into the file system.

The remainder of this section introduces the JCrasher representation of the parameter-space, followed by the JCrasher test-case selection algorithm.

4.2.1 Parameter-graph

To start with an example, imagine that JCrasher analyzes a method `f` of a class `C` with signature `f(A1, A2, ..., AN) returns R`. From this, JCrasher infers that in order to test this method it has to know how to construct values of types `C`, `A1`, `A2`, . . . , `AN`. Additionally, JCrasher infers that it can construct an object of type `R` or any of `R`'s supertypes—the classes `R` extends and the interfaces it implements, as long as it can construct a `C` and an `A1` and an `A2`, etc.

Additionally, JCrasher has implicit knowledge of how to create certain well-known values of different types. For instance, JCrasher knows that it can use the `null` value for any object type. Similarly, a few pre-set values are used to test primitive types—for example, 1.0, 0.0, and -1.0 for `double`.

One way to encode the above knowledge is as Prolog-like inference rules. Every method and every well-known way to construct values of a type can be seen to correspond to such an inference rule. For instance, a method `f(A1, A2, ..., AN) returns R` in class `C` corresponds to a rule `R` ← `C`, `A1`, `A2`, . . . , `AN`. Constructing the well-known value 1.0 of type `double` corresponds to the rule `double` ← `1.0`. Creating test-cases corresponds to a search action or Prolog-like inference in the space induced by these inference rules.

Instead of storing such inference rules in text form, we represent them as a graph data structure that we call the *parameter-graph*. This representation is convenient both for computation and for illustration. The parameter-graph is computed by

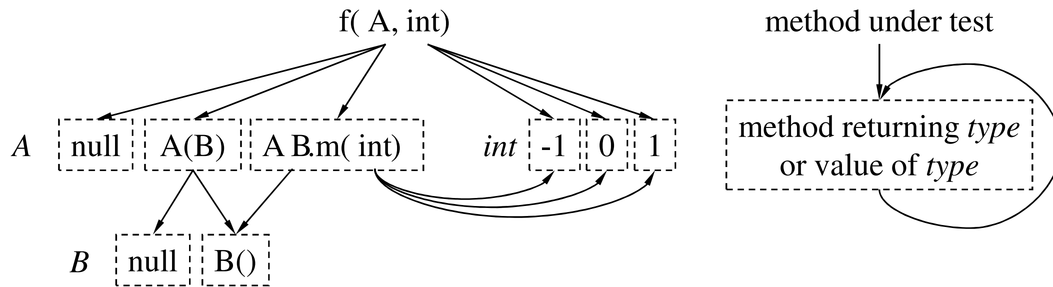


Figure 5: Left: JCrasher's internal representation of the potential test cases for a method `f(A, int)` under test. Right: Key. JCrasher has found `A`-returning methods `A(B)` and `B.m(int)`, and the `B`-returning method `B()`. For `int` JCrasher uses the predefined values `-1`, `0`, and `1`; `null` is predefined for reference types such as `A` and `B`. JCrasher derives test-cases for `f` from such a parameter-graph by choosing parameter combinations, for example, `f(null, -1)`, `f(null, 0)`, `f(null, 1)`, `f(new A(null), -1)`, `...`, `f(new B().m(1), 1)`.

examining the methods of the current program under test. Figure 5 illustrates the concept of test-case generation from a parameter-graph. A node `f(A,int)` in this graph means that to test method `f` we need both a value of type `A` *and* a value of type `int`. An edge in the graph goes from a type to a method or a well-known value and reflects an inference rule, or a way to get a value of the type. So there is an edge from type `A` to method `m` if `m` can be used to produce a value of type `A`. Multiple edges with the same source node are alternative ways to create a value of this type.

The parameter-graph is an abstract representation of the parameter-space of the program's methods. Creating different parameter combinations for a method under test can be done by traversing the graph. As discussed in the next section, the representation is useful because it allows us to easily bound the depth of method chaining. The following properties of this approach are worth noting.

- Ideally, each test-case should provide a method with a different combination of its parameter-types' value-range. In our approach it is possible that two or more test-cases produce the same value. This can happen if a `T`-returning method

returns the same value for different parameter combinations, or two `T`-returning methods have overlapping return-value ranges.

- If test-cases are picked at random then a method with more parameter combinations is tested more often. This is advantageous, as more parameter combinations tend to produce a bigger variety of parameter state. It can be assumed that the method is more complicated as it needs to handle more cases. It is good to generate more test-cases for a more complicated method.

- Possible side-effects of methods via variables are ignored in test-case selection. JCrasher does not attempt to deliberately search the space of possible side-effects by exploring all possible combinations of method calls. For this reason void-returning methods are currently excluded from the parameter-graph. This limitation is entirely pragmatic: if void-returning methods are considered, the test parameter space becomes huge very quickly. The issue of side-effects between test cases is discussed separately in Chapter 8.

4.2.2 Test-case selection

By representing the parameter-space implicitly—via the parameter-graph—we can efficiently compute the number of test-cases for a given search depth, that is, compute the size of the parameter-space. Similarly, we can have random access to the test cases. Figure 6 illustrates the size computation of a sub-parameter-space.

Computing the size of the parameter-space without creating all possible tests is important because we typically do not have enough resources to generate and execute all the test cases we could generate. This means we have to restrict ourselves to a subset or sample of all the test cases we could generate. Knowing the total size of the parameter-space, JCrasher can select to output randomly a certain percentage of the test-cases.

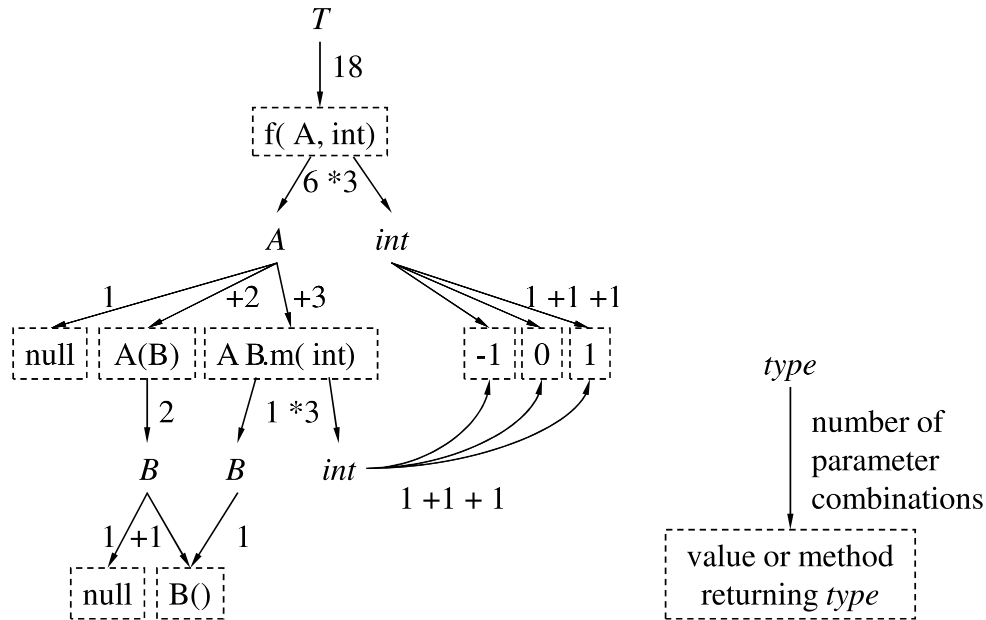


Figure 6: JCrasher calculating the maximum number of test cases it can generate for a method `f(A, int)` under test. The size of a sub-parameter-space is calculated bottom up by adding the sizes of a type's value and method spaces and multiplying the sizes of a method's parameter type spaces. Left: example continued from Figure 5. Right: key.

4.3 Runtime support

4.3.1 Test-case execution and exception filtering

Each test-case consists of one or more method or constructor invocations contained in a single `try` block:

```
public void test1() throws Throwable {
  try {
    //test-case
  }
  catch (Exception e) {
    dispatchException(e);
  }
}
```

Each `Exception` thrown by a test-case is caught and passed to the JCrasher runtime. JCrasher supports different heuristics for filtering exceptions. In this context, we will not use the default JCrasher heuristic described in the JCrasher paper [23], which takes into account the type of the exception and the method call history that led to the exception. All heuristics have in common that they either consider an exception worth reporting to the user and therefore pass it on to JUnit, or bogus, possibly due to an ill-formed generated test case, and therefore suppress the exception.

4.3.2 Grouping similar exception together

An important issue is the grouping of exceptions. Since JCrasher can produce many isomorphic test cases, thousands of exceptions could be caused by the same error—or non-error. We currently have a heuristic way to group exceptions before presenting them to the user. The grouping is done according to the contents of the call-stack when the exception is thrown. Grouping is significant from the usability standpoint because it eliminates many of the complexities of dealing with false positives during the automatic testing process. For a hypothetical scenario, imagine that 1 million tests are run and 50,000 of them fail. It is likely that the independent causes of failure are no more than a handful of value combinations. By grouping all exceptions based on where they were thrown from and what other methods are on the stack, we offer the user a scalable way of browsing the results of JCrasher and separating errors from false positives.

4.4 Experience

In this section we discuss practical considerations concerning the use of JCrasher, as well as some experiments and results. We conduct these experiments with the full version of JCrasher, using its default filtering heuristics, to provide examples of the capabilities of a dynamic baseline analysis, and as context for the experiments in Chapter 5 and Chapter 6.

4.4.1 Pragmatics

JCrasher can easily produce large amounts of file system data. For 1 million tests the disk space occupied is 200-300 MB for source code and another 100 to 150 MB for bytecode. Also, compiling the test cases is expensive—about 7 ms per test case on average in our environment. One could argue that the JCrasher batch mode could have higher performance if it avoided the explicit use of JUnit and did not store the test cases as files. If, instead, the testing was done by JCrasher while the test cases are being produced and only the failed test cases were exported as JUnit files, then performance could increase significantly. Although for inspection reasons it is beneficial to reify all test cases as files, we plan to explore the above alternative in future work. Note that a faster runtime would make our fast static state resetting techniques discussed in Chapter 8 even more important.

4.4.2 Use of JCrasher

This section presents experiments on the use of JCrasher. We demonstrate the types of problems that JCrasher finds, the problems it does not find, its false positives, as well as performance metrics on the test cases.

In our experiments, we tried JCrasher 0.27 on the Raytracer application from the SPEC JVM benchmark suite, on homework submissions for an undergraduate programming class, and on the uniqueBoundedStack class used previously in the testing literature [97, 113]. Xie and Notkin refer to this class as UB-Stack, a name that we adopt for brevity. We have changed all methods of UB-Stack from package-visible to public as JCrasher only tests public methods. Besides this, we did not modify the testees for our experiments. JCrasher 0.27 and the selected test cases are available on the JCrasher project web site.

As described previously, JCrasher attempts to detect robustness failures, but robustness failures do not always imply bugs. Hence, the user needs to inspect the

JCrasher output and determine whether the test input was valid, whether the program behavior constitutes a robustness failure, and whether the program behavior constitutes a bug. We present next selected instances of robustness failures, although not all are bugs. In the case of student homeworks, we had access to many more submissions but we limited the number of testees to a small set that covers all the different exception types reported, but did not otherwise bias the selection, which should be fairly representative. By picking only a few testees we could easily examine the JCrasher reports and determine whether they constitute bugs. In general, JCrasher detected a few shallow problems in the testees. These problems can usually also be found during unstructured testing or by running simple test cases. The value of JCrasher is that it automatically detects these shallow problems and therefore allows the developer to concentrate on interesting problems.

Table 1 and Table 2 give program and testing performance metrics for the testees. We have conducted our experiments in the testing environment described in Section 8.6. The numbers shown are for the JUnit text interface—which gives faster test execution than the graphical interface. We show how many tests JCrasher generates, how long it takes to generate these tests, how much disk space they occupy, how long it takes to execute them, how many problems are reported to the user, how much disk space these reports occupy, how many of these reports we consider redundant, and how many reports can reasonably be considered bugs.

To enhance readability we have condensed the layout of the source code shown in this section. Omitted code is represented by `//[..]`.

4.4.3 Raytrace benchmark

We ran JCrasher on the Raytracer benchmark of the SPEC JVM suite. This experiment was not primarily intended to find errors since the application is very mature. Instead, we wanted to show the number and size of test cases (see Table 1 and Table 2)

Table 1: JCrasher results on several smaller testees. *Test cases* gives the total number of test cases generated for a testee when searching up to a method-chaining depth of three. *Crashes* denotes the number of errors or exceptions thrown when executing these test cases. *Problem reports* denotes the number of distinct groups of robustness failures reported to the user—all crashes in a group have an identical call-stack trace. *Redundant reports* gives the number of problem reports we have manually classified as redundant. *Bugs* denotes the number of problem reports that reveal a violation of the testee's specification.

Testee			Tests				
Class name	Author	Public methods	Test Cases	Crashes	Problem reports	Redundant reports	Bugs
Canvas	SPEC	6	14382	12667	3	0	1?
P1	s1	16	104	8	3	0	1 (2?)
P1	s1139	15	95	27	4	0	0
P1	s2120	16	239	44	3	0	0
P1	s3426	18	116	26	4	0	1
P1	s8007	15	95	22	2	0	1
BSTree	s2251	24	2000	941	4	2	1
UB-Stack	Stotts	11	16	0	0	0	0

Table 2: JCrasher's resource consumption in our experiments. The execution time and disk space required for generated test cases. *Report* is the output of the JUnit text interface redirected to a text file.

Testee		Tests				
Class name	Author	Test Cases	Creation time [s]	Source size [kB]	Execution time [s]	Report size [kB]
Canvas	SPEC	14382	5.0	6000	14.9	30
P1	s1	104	0.3	20	1.0	1
P1	s1139	95	0.3	19	0.7	2
P1	s2120	239	0.3	55	1.3	2
P1	s3426	116	0.3	23	0.6	2
P1	s8007	95	0.3	19	0.5	1
BSTree	s2251	2000	0.9	564	3.4	6
UB-Stack	Stotts	16	0.3	4	0.5	0

generated for a class in a realistic application, where a lot of methods can be used to create type-correct data. Since we have no specification for the application, we cannot tell what behavior constitutes a bug, but we try to make reasonable estimates.

In the Canvas class of the Raytracer, we found a constructor method that is particularly interesting. The constructor below throws a `NegativeArraySizeException` when passed parameters `(-1, 1)`.

```
/* spec.benchmarks._205_raytrace */
public Canvas(int width, int height) {
  Width = width; Height = height;
  if (Width < 0 || Height < 0) {
    System.err.print("Invalid window size!" + "\n");
    //System.exit(-1);
  }
  Pixels = new int[Width * Height];
}
```

In this example, it is clear that passing in a negative `width` would be an erroneous input. Indeed, the original developer of this code agreed that it would be an error to continue after this input—a line to exit the program existed. Nevertheless, the exit command is now commented out, possibly because the Raytracer is used as part of a larger program. But it is clear that continuing with a negative width or height is a robustness error. It is not even necessarily the case that the error will be caught during the allocation of the `Pixels` array: if both `height` and `width` are negative, their product will be positive and the allocation will succeed, letting the error propagate further in the program.

The test case below generated by JCrasher calls the following `Write` method of Canvas, which in turn calls its private `SetRed` method. But the `SetRed` methods throws an `ArrayIndexOutOfBoundsException` because of the -1 and 0 values passed

to Write.

```
public void test93() throws Throwable {
  try {
    Color c4 = new Color(-1.0f, -1.0f, -1.0f);
    Canvas c5 = new Canvas(-1, -1);
    c5.Write(-1.0f, -1, 0, c4);
  } // [..]
}

public void Write(float brightness, int x, int y, Color color) {
 color.Scale(brightness);
 float max = color.FindMax();
 if (max > 1.0f) color.Scale(1.0f / max);
 SetRed(x, y, color.GetRed() * 255); //[..]
}

private void SetRed(int x, int y, float component) {
 int index = y * Width + x;
 Pixels[index] = Pixels[index] & 0x00FFFF00 | ((int) component);
}
```

This is a case of a robustness failure that probably does not represent a bug—it can be argued that the input violates the preconditions of the routine. Nevertheless, it would be a good programming practice to use the new Java assertion facility to enforce the preconditions of method `Write`.

4.4.4 Student Homeworks

We applied JCrasher in testing homework submissions from a second-semester Java class. This is a good application domain, as beginning programmers are likely to introduce shallow errors in their programs. Furthermore, since the tasks required are small, self-contained, and strictly specified—the homework handout describes in detail the input assumptions—it is easy to distinguish bugs from normal behavior. On the other hand, these student programs are usually too small to exhibit interesting constructor nesting depth or a large number of test cases for a single program.

We next describe a few selected cases of robustness failures. One task in homework assignment P1 required the coding of a pattern-matching routine. Given a pattern and a list of integers, both represented as arrays, the routine should attempt to find the pattern in the list. Passing `([0], [0])` to the following method `findPattern` by programmer s1 causes an `ArrayIndexOutOfBoundsException`, although the input is perfectly valid. The cause is a badly coded routine:

```
public static int findPattern(int[] list, int[] pattern) {
  int place = -1; int iPattern = 0; int iList;
  for (iList = 0; iList < list.length; iList++)
    if (isTheSame(list[iList], pattern[iPattern]) == true)
      for (iPattern = 0;
           ((iPattern <= pattern.length)
             && (isTheSame(list[iList], pattern[iPattern]) == true));
           iPattern++)
      { place = iList + 1;
        isTheSame(list[iList + 1],pattern[iPattern + 1]);
      }
}
```

A similar error is reported in a different statement when the input is `([0], [])`, which is also a bug, but can be argued to violate the programmer's understood precondition.

Another robustness failure that can be considered a bug is programmer s3426's testee P1 throwing a `NumberFormatException` when passing String `" "` to the following main method.

```
public static void main(String[] argv) {
  int tests = Integer.parseInt(argv[0]); //[..]
}
```

This is a common case of receiving unstructured input from the user, without checking whether it satisfies the programmer's assumptions.

Another bug consists of a `NegativeArraySizeException` reported for programmer s8007's testee P1 when passing -1 to the following method `getSquaresArray`.

```
public static int[] getSquaresArray(int length) {
  int[] emptyArray = new int [length];  // [..]
}
```

This is a programmer error since the homework specification explicitly states "If the value passed into the method is negative, you should return an empty array."

An interesting case of error is exhibited in programmer s2251's BSTree homework assignment. The BSTNode class contains code as follows:

```
public BSTNode(Object dat){
  setData(dat); //[..]
}
public void setData(Object dat){
  data = (Comparable) dat;
}
```

That is, the constructor checks at run-time that the data it receives support the `Comparable` interface. This caused an exception for a JCrasher-generated test case. Changing the signature of the above `BSTNode` constructor from `BSTNode(Object dat)` to `BSTNode(Comparable dat)` fixes the problem. As a result of this change, JCrasher produces only 332 test cases (instead of 2000 before), detects 60 crashes (941), and reports three problems (four). The reason is that JCrasher finds fewer possibilities to create a `Comparable` than an `Object`. The remaining three reported problems for the BSTNode program are caused by passing a `null` reference to a method that does not expect it. Hence, two of the problem reports are redundant and none of the three are bugs.

The rest of our tests from Table 1 and Table 2 reported similar robustness failures—due to negative number inputs, null pointer values, etc.—that did not, however, constitute bugs.

4.4.5 UB-Stack

We finally tested JCrasher with the UB-Stack class, previously used as a case study in the testing literature. JCrasher does not report any problems for UB-Stack. Stotts, Lindsey, and Antley have hand-coded two sets of eight and 16 test cases, respectively. The smaller set does not reveal any bugs but the bigger set reveals a bug [97]. The bug occurs when executing a sequence of the testee's methods. JCrasher cannot detect this bug since it currently does not produce test cases that explore executing different methods in sequence, just for their side-effects. Xie and Notkin [113] have presented an iterative invariant and test case generation technique, and used the two hand-coded sets as initial sets. For both initial sets they generate test cases that reveal at least one bug in UB-Stack.

CHAPTER V

SOLUTION BACK-END: BUG WARNINGS TO TEST CASES

In this chapter we address the language-level imprecision of a static analysis tool like ESC/Java. The following chapter will address user-level imprecision. Here, we combine static checking and concrete test-case generation in the Check 'n' Crash tool [24]. The approach consists of taking the abstract error conditions inferred using theorem proving techniques by a static checker (ESC/Java), deriving specific error conditions using a constraint solver, and producing concrete test cases (with the JCrasher tool) that are executed to determine whether a path truly exists. The combined technique has advantages over both static checking and automatic testing individually. Compared to ESC/Java, we eliminate spurious path reports and improve the ease-of-comprehension of reports through the production of Java counterexamples. Compared to JCrasher, we eliminate the blind search of the input space, thus reducing the testing time and increasing the test quality.

5.1 Overview

Check 'n' Crash [24] is a tool for automatic bug finding. It combines ESC/Java and the JCrasher random testing tool [23]. Check 'n' Crash takes error conditions that ESC/Java infers from the testee, derives variable assignments that satisfy the error condition (using a constraint solver), and compiles them into concrete test cases that are executed with JCrasher to determine whether the error is language-level sound. Figure 7 shows the elements of Check 'n' Crash pictorially. Compared to ESC/Java alone, Check 'n' Crash's combination of ESC/Java with JCrasher eliminates spurious

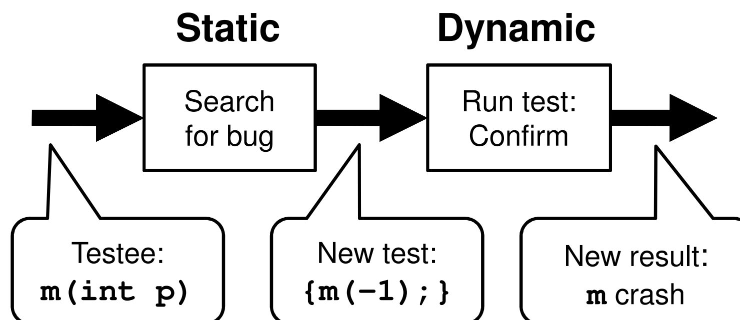


Figure 7: Check 'n' Crash workflow: From Java testee source file input to test case source file and exception report output. Check 'n' Crash uses ESC/Java to statically check the testee for potential bugs. In this example, ESC/Java warns about a potential runtime exception in the analyzed method when passing a negative parameter (the ESC/Java warning is not shown). Check 'n' Crash then compiles ESC/Java's bug warnings to concrete test cases to eliminate those warnings that cannot be reproduced in actual executions. In this example, Check 'n' Crash produces a test case that passes -1 into the method and confirms that it throws the runtime exception ESC/Java has warned about.

warnings and improves the ease of comprehension of error reports through concrete Java counterexamples.

Check 'n' Crash takes as inputs the names of the Java files under test. It invokes ESC/Java, which derives error conditions. Check 'n' Crash takes each error condition as a constraint system over a method `m`'s parameters, the object state on which `m` is executed, and other state of the environment. Check 'n' Crash extends ESC/Java by parsing and solving this constraint system. A solution is a set of variable assignments that satisfy the constraint system. [24] discusses in detail how we process constraints over integers, arrays, and reference types in general.

Once the variable assignments that cause the error are computed, Check 'n' Crash uses JCrasher to compile some of these assignments to JUnit [8] test cases. The test cases are then executed under JUnit. If the execution does not cause an exception, then the variable assignment was a false warning: no error actually exists. Similarly, some runtime exceptions do not indicate errors and JCrasher filters them out. For instance, throwing an `IllegalArgumentException` exception is the recommended

Java practice for reporting illegal inputs. If the execution does result in one of the tracked exceptions, an error report is generated by Check 'n' Crash.

5.2 Benefits relative to ESC/Java

Check 'n' Crash has two advantages over using ESC/Java alone. First, Check 'n' Crash ensures that all errors it reports are indeed reproducible: they are possible for some combination of values. Second, Check 'n' Crash offers ease of inspection of error cases and concrete test cases that can be integrated in a regression test suite.

5.2.1 Guaranteeing language-level soundness

Every case reported by Check 'n' Crash is guaranteed to be language-level sound. This is guaranteed by observing concrete test case executions in a standard execution environment (such as a production-level Java virtual machine). Only cases that could be confirmed by such concrete executions are reported to the user.

5.2.2 Improving the clarity of reports

Check 'n' Crash reduces all constraints to a small test case. In the above case, the generated test method is just 10 lines long. Furthermore, users are likely to be more familiar with Java syntax than with the conditions produced by Simplify. Finally, having a concrete test case gives the user the option to integrate it in a regression test suite for later use.

5.3 Example

To see the difference between an error condition generated by ESC/Java and the concrete test cases output by Check 'n' Crash, consider the following method `swapArrays`, taken from a student homework solution. The method's informal specification states that the method swaps the elements from `fstArray` to `sndArray` and vice versa. If the arrays differ in length the method should return without modifying any parameter.

```
public static void swapArrays(double[] fstArray, double[] sndArray)
{ //..
  for(int m=0; m<fstArray.length; m++)
  { //..
    fstArray[m]=sndArray[m]; //..
  }
}
```

ESC/Java issues the following warning, which indicates that `swapArrays` might crash with an array index out-of-bounds exception.

```
Array index possibly too large (IndexTooBig)
  fstArray[m]=sndArray[m];
                      ^
```

Optionally, ESC/Java emits the error condition in which this crash might occur. This condition is a conjunction of constraints. For `swapArrays`, which consists of five instructions, ESC/Java emits some 100 constraints. `0 < fstArray.length` and `sndArray.length == 0` are the most relevant ones (formatted for readability).

Check 'n' Crash parses the error condition generated by ESC/Java and feeds the constraints to its constraint solvers. In our example, Check 'n' Crash creates two integer variables, `fstArray.length` and `sndArray.length`, and passes their constraints to the POOC integer constraint solver [92]. Then Check 'n' Crash requests a few solutions for this constraint system from its constraint solvers and compiles each solution into a JUnit [8] test case. For this example, the test case will create an empty and a random non-empty array. This will cause an exception when executed and JCrasher will process the exception according to its heuristics and conclude it is a language-level sound failure and not a false bug warning.

5.4 Structure

Check 'n' Crash combines ESC/Java and JCrasher. Check 'n' Crash takes as input the names of the Java files under test. It invokes ESC/Java 2.07a, which compiles the Java source code under test to a set of predicate logic formulae [42, 72]. ESC compiles each method `m` under test to its weakest precondition $wp(\texttt{m}, \texttt{true})$. This formula specifies the states from which the execution of `m` terminates normally. We use `true` as the postcondition, as we are not interested in which state the execution terminates as long as it terminates normally. The states that do not satisfy the precondition are those from which the execution "goes wrong".

We are interested in the following specific cases of the execution going wrong [73, Chapter 4].

- Assigning a supertype to an array element.
- Casting to an incompatible type.
- Accessing an array outside its domain.
- Allocating an array of negative size.
- Dereferencing null.
- Dividing by zero.

These cases are statically detected using ESC/Java but they also correspond to Java runtime exceptions (program crashes) that will be caught during JCrasher-initiated testing.

A state from which the execution of `m` goes wrong is also called a "counterexample". ESC uses the Simplify theorem prover [33] to derive counterexamples from the conjunction of $wp(\texttt{m}, \texttt{true})$ and additional formulae that encode the class to which `m` belongs as well as Java semantics.

We view such a counterexample as a constraint system over `m`'s parameters, the object state on which `m` is executed, and other state of the environment. Check 'n' Crash extends ESC by parsing and solving this constraint system. A solution is a set of variable assignments that satisfy the constraint system. Section 5.5 shows examples of constraints and discusses in detail how we process constraints over integers, arrays, and reference types in general.

Once the variable assignments that cause the error are computed, Check 'n' Crash uses JCrasher to compile some of these assignments to JUnit test cases. The test cases are then executed under JUnit. If the execution does not cause an exception, then the variable assignment was a false positive: no error actually exists. Similarly, some runtime exceptions do not indicate errors and JCrasher filters them out. For instance, throwing an `IllegalArgumentException` exception is the recommended Java practice for reporting illegal inputs. If the execution does result in one of the tracked exceptions, an error report is generated by Check 'n' Crash.

5.5 Constraint solving

Constraint solving is the Check 'n' Crash glue that keeps together ESC/Java and JCrasher. We next discuss how we solve the abstract constraints extracted from ESC/Java counterexamples to generate values that are used in JCrasher test cases. Note that all constraint solving is by nature a heuristic technique: we are not always able to solve constraints, even if they are indeed solvable.

5.5.1 Primitive types: Integers

We first discuss our approach for integer numbers.[1] Check 'n' Crash uses the integer constraint solver included in the POOC platform for object-oriented constraint programming [92]. As an example, consider the following method under test.

[1]Floating point constraints are currently not supported in Check 'n' Crash (random values are used, just as in JCrasher) but would be handled similarly.

```
public int m1(int a, int b, int c) {
  if (0<=a && a<b && a!=c && c>=0) {
    return (a+b)/c;
  }
  else {return 0;}
}
```

ESC/Java correctly detects a division by zero and returns the following constraint system for explanation, which we have pruned and simplified for readability.

```
a<b; 0<=a; c==0; c!=a
```

The first solution POOC returns for the above constraint system is (1, 2, 0). Check 'n' Crash outputs a corresponding JUnit test case that first creates an instance of the class defining `m1` and then calls `m1` with parameters (1, 2, 0). The test case catches any exception thrown during execution and, if the exception indicates an error (as in this case), a report is produced.

5.5.2 Complex types: Objects

Constraints in abstract counterexamples may involve aliasing relations among object references. We solve these with a simple equivalence-class-based aliasing algorithm, similar to the well-known alias analysis by Steensgard [96].

Our implementation maintains an equivalence relation on reference types. For each reference type we keep a mapping from field identifier to variable. This allows us to store constraints on fields. A reference constraint `a=b` specifies that `a` and `b` refer to the same object. We model this by merging the equivalence classes of `a` and `b`, creating a new equivalence class that contains both. After processing all constraints we generate a test case by creating one representative instance per equivalence class. This instance will be assigned to all members of the class.

For an example, consider the following method.

```
public class Point{public int x;}


public int m2(Point a, Point b) {
  if (a==b) {return 100/b.x;}
  else {return 0;}
}
```

ESC/Java generates a counterexample from which we extract the following constraint system.

```
a.x=0, a=b
```

After the two constraints are processed, `a` and `b` are in the same equivalence class. A single `Point` object needs to be created, with both `a` and `b` pointing to it and its `x` field equal to 0. (In this case, the integer constraint is straightforward but generally the integer constraint solver will be called to resolve constraints.) We use reflection to set object fields in the generated test case. In our example:

```
Point p1 = new Point();
Point.class.getDeclaredField("x").set(p1, new Integer(0));
Point p2 = p1;
Testee t = new Testee();
t.m2(p1, p2);
```

5.5.3 Complex types: Arrays

Check 'n' Crash uses a simple approach to deal with arrays. Array expressions with a constant index (such as `a[1]`) are treated as regular variables. Array expressions with a symbolic index (such as `a[b]` or `a[m()]`) are replaced with a fresh variable and the resulting constraint system (with regular references and primitives) is solved. The solutions returned are used one-by-one to turn array expressions with symbolic

indices into array expressions with constant indices. If the resulting constraint system is solvable (i.e., has no contradictory constraints) then the solution is appropriate for the original constraint system.

An example illustrates the approach. Consider the method:

```
public int m3(int[] a, int b) {
  if (a[a[b]] = 5) {return 1/0;}
  else {return 0;}
}
```

For the case of division by zero, Check 'n' Crash extracts the following constraint system from the ESC/Java counterexample report:

```
0 <= b
a[b] < arrayLength(a)
0 <= a[b]
b < arrayLength(a)
a[a[b]] = 5
```

Check 'n' Crash then rewrites the constraint system as follows, using the names `x:=b`, `y:=a[b]`, and `z:=a[a[b]]`.

```
0 <= x < arrayLength(a)
0 <= y < arrayLength(a)
z = 5
```

For the rewritten constraint system our first solution candidate is `x:=0`, `y:=0`, and `z:=5`. But this causes a conflict in the array as `b=0` and `a[b]=a[0]=0` but `a[a[b]]=a[0]=5`. Therefore we discard this solution and query the integer constraint solver for another solution. The next solution `x:=0`, `y:=1`, `z:=5` satisfies the constraint system. So we generate the corresponding test case, which passes the parameter values (`0`, `new int[]{1,5}`) to `m3`.

5.6 Benefits relative to JCrasher

Check 'n' Crash is a strict improvement over using JCrasher alone for automatic testing. By employing the power of ESC/Java, Check 'n' Crash guides the test case generation so that only inputs likely to produce errors are tested. Thus, a small number of test cases suffice to find all problems that JCrasher would likely find. To confirm this claim, we reproduced the experiments from the JCrasher paper [23] using Check 'n' Crash. The programs under test include the Raytracer application from the SPEC JVM 98 benchmark suite, homework submissions for an undergraduate programming class, and the `uniqueBoundedStack` class used previously in the testing literature [97, 113]. (Xie and Notkin refer to this class as UB-Stack, a name that we adopt for brevity.) These testees are mostly small and occasionally have informal specifications. For instance, in the case of homework assignments, we review the homework handout to determine what kinds of inputs should be handled by each method and how. Thus, we can talk with some amount of certainty of bugs instead of just error reports and potential bugs.

Table 3 summarizes the results of running JCrasher 0.2.7 and Check 'n' Crash 0.4.10 on the set of testees. The bugs are reported as a range containing some uncertainty, as occasionally it is clear that a program fragment represents a bad practice, yet there is a possibility that some implicit precondition makes the error scenario infeasible. Check 'n' Crash has more flexibility than JCrasher in its settings, therefore for these tests we chose the Check 'n' Crash settings so that they emulate the default JCrasher behavior. Specifically, Check 'n' Crash parameterizes ESC so that it searches for the potential problems listed in Section 5.4, other than dereferencing null. ESC unrolls each loop 1.5 times. We use no JML specifications of the Java class libraries. Using these JML specifications included in the ESC distribution does not change the reports Check 'n' Crash produces or the bugs it finds for this experiment. We use the runtime heuristics reported in the JCrasher paper [23] for both JCrasher

Table 3: JCrasher and Check 'n' Crash results on several smaller testees.

Public methods gives the number of public methods and constructors declared by public classes. Constructors of abstract classes are excluded. *Test cases* gives the total number of test cases generated for a testee when JCrasher searches up to a method-chaining depth of three and Check 'n' Crash creates up to ten test cases per ESC generated counterexample. *Crashes* denotes the number of errors or exceptions of interest thrown when executing these test cases. *Reports* denotes the number of distinct groups of failures reported to the user—all crashes in a group have an identical call-stack trace. *Bugs* denotes the number of problem reports that reveal a violation of the testee's specification.

Testee			Tests							
Class	Author	Public	Test Cases		Crashes		Reports		Bugs	
name		methods	JCrasher	CnC	JCrasher	CnC	JCrasher	CnC	JCrasher	CnC
Canvas	SPEC	6	14382	30	12667	21	3	2	0–1	0–1
P1	s1	16	104	40	8	40	3	3	1–2	2–3
P1	s1139	15	95	40	27	40	4	4	1–2	2–3
P1	s2120	16	239	50	44	50	3	2	0	1
P1	s3426	18	116	45	26	45	4	4	2	3
P1	s8007	15	95	10	22	10	2	1	1	1
BSTNode	s2251	24	3872	13	1547	8	4	2	1	1
UB-Stack	Stotts	11	16	110	0	0	0	0	0	0

and Check 'n' Crash. These heuristics determine which exceptions should be ignored because they probably do not constitute program errors. For identical exceptions produced with the same method call stack, only one error report is output.

As can be seen in the table, Check 'n' Crash detects all the errors found by JCrasher with only a fraction of the test cases (except for UB-Stack, where JCrasher found few opportunities to create random data conforming to the class's interface) and slightly fewer reports. This confirms that the search of Check 'n' Crash is much more directed and deeper, yet does not miss any errors uncovered by random testing. (Note: The numbers reported for JCrasher are identical to those in reference [23] with two exceptions. First, the bug count for s1139 and s3426 is increased by one, since on further review two more reports were shown to be bugs. Second, for the Binary Search Tree homework submission, we run both programs on the BSTNode class, which contains the error, instead of on the BSTree front-end class.)

For a representative example of a reported bug, the following `getSquaresArray` method of user s8007's testee P1 causes a `NegativeArraySizeException`. (We have formatted the testees for readability, "`//..`" indicates code we have omitted.)

```
public static int[] getSquaresArray(int length) {
  int[] emptyArray = new int [length]; //..
}
```

This is a bug, since the homework specification explicitly states "If the value passed into the method is negative, you should return an empty array." The constraints reported by ESC/Java for this error essentially state that `length` should be negative. Our constraint solving then produces the value -1000000 and uses JCrasher to output a test case to demonstrate the error.

In addition to JCrasher's results, Check 'n' Crash also found the example bug of Section 5.3, in all P1 testees except s8007's. When passing arrays of different lengths

(e.g., `([1.0], [])`) to the `swapArrays` method, these testees crash by accessing an array out of bounds. We classify this as a bug, since the homework specification allows arrays of different lengths as input. JCrasher did not discover this error because its creation of random array values is limited.

5.7 Experience

We next describe our experience in using Check 'n' Crash on complete applications and our observations on the strengths and weaknesses of the tool.

5.7.1 JABA and JBoss JMS

To demonstrate the uses of Check 'n' Crash in practice, we applied it to two realistic applications: the JABA bytecode analysis framework[2] and the JMS module of the JBoss[3] open source J2EE application server. The latter is an implementation of Sun's Java Message Service API [53]. Specifically, we ran Check 'n' Crash on all the `jaba.*` packages of JABA, which consist of some 18 thousand non-comment source statements (NCSS), and on the JMS packages of JBoss 4.0 RC1, which consist of some five thousand non-comment source statements. We should note that there is no notion of testing the scalability of Check 'n' Crash since the time-consuming part of its analysis is the intra-procedural ESC/Java analysis. Hence, in practice, the Check 'n' Crash running time scales roughly linearly with the size of the input program.

We tested both applications without any annotation or other programmer intervention. None of the tested applications has JML specifications in its code. This is indeed appropriate for out test, since JML annotations are rare in actual projects. Furthermore, if we were to concentrate on JML-annotated programs, we would be unlikely to find interesting behavior. JML-annotated code is likely to have already

[2]http://www.cc.gatech.edu/aristotle/Tools/jaba.html
[3]http://www.jboss.org/

Table 4: Check 'n' Crash results on JABA and JBoss JMS.

Testee		CnC			
Package	Size [NCSS]	Creation [min:s]	Test Cases	Crashes	Reports
jaba	17.9 k	25:58	18.3 k	4.4 k	56
jboss.jms	5.1 k	1:35	3.9 k	0.6 k	95

been tested with ESC/Java and have the bugs that Check 'n' Crash detects already fixed.

Table 4 presents statistics of running Check 'n' Crash on JABA and JBoss JMS. The analysis and test creation time was measured on a 1.2 GHz Pentium III-M with 512 MB of RAM. We use the Check 'n' Crash configuration from the previous experiment (Section 5.6), but also search for null dereferences.

Since we are not familiar with the internal structure of either of these programs, we are not typically able to tell whether an error report constitutes a real bug or some implicit precondition in the code precludes the combination of inputs that exhibit a reported crash. Exhaustive inspection of all the reports by an expert is hard due to the size of the applications (especially JABA) and, consequently, the number of Check 'n' Crash reports. For instance, Check 'n' Crash (and ESC/Java) may report that a method can fail with a null pointer exception, yet it is not always clear whether the input can occur in the normal course of execution of the application. For this reason, we selected a few promising error reports and inspected them more closely to determine whether they reveal bugs. In the case of JBoss JMS, it is clear on a couple of occasions (see below) that a report corresponds to a bug. Similarly, we discussed five potential errors with JABA developers and two of them entered their list of bugs to be fixed.

For example, one of the constructors of the `ClassImpl` class in the JABA framework leaves its instance in an inconsistent state—the `name` field is not initialized. Discovering this error is not due so much to ESC/Java's reasoning capabilities but rather to the random testing of JCrasher, which explores all constructors of a class in order to produce random objects. ESC/Java directs Check 'n' Crash to check methods of this class, and calling a method on the incorrectly initialized object exposes the error. In a similar case, a method of the concrete class `jaba.sym.NamedReferenceTypeImpl` should never be called on objects of the class directly—instead the method should be overridden by subclasses. The superclass method throws an exception to indicate the error when called. This is a bad coding practice: the method should instead have been moved to an interface that the subclasses will implement. Although the offending method is protected, it gets called from a public method of the class, through a call chain involving two more methods. Check 'n' Crash again discovers this error mostly due to JCrasher creating a variety of random values of different types per suspected problem.

For an error that is caught due to the ESC/Java analysis, consider the following illegal cast in method `writeObject` of class `org.jboss.jms.BytesMessageImpl`:

```
public void writeObject(Object value) throws JMSException
{
  //..
  if (value instanceof Byte[]) {
    this.writeBytes((byte[]) value);
  }  //..
}
```

The type of `value` in the above is `Byte[]` and not `byte[]`. Check 'n' Crash finds this error because ESC/Java reports a possible illegal cast exception in the above.

Similarly, the potential of a class cast exception reveals another bad coding practice in method `getContainer` of class `org.jboss.jms.container.Container`. The formal argument type should be specialized to `Proxy`.

```
public static Container getContainer(Object object) throws Throwable
{
  Proxy proxy = (Proxy) object;  //..
}
```

5.7.2 Check 'n' Crash usage and critique

In our experience, Check 'n' Crash is a useful tool for identifying program errors. Nevertheless, we need to be explicit about the way Check 'n' Crash would be used in practice, especially compared to other general directions in software error detection. Check 'n' Crash's strengths and weaknesses are analogous to those of the tools it is based on: ESC/Java and JCrasher. The best use of Check 'n' Crash is during development. The programmer can apply the tool to newly written code, inspect reports of conditions indicating possible crashes, and possibly update the code if the error condition is indeed possible (or update the code preconditions if the inputs are infeasible and preconditions are being maintained). Generated tests can also be integrated in a JUnit regression test suite.

A lot of attention in the error checking community has lately focused on tools that we descriptively call "bug pattern matchers" [51, 116, 57]. These are program analysis tools that use domain-specific knowledge about incorrect program patterns and statically analyze the code to detect possible occurrences of the patterns. Example error patterns include uses of objects after they are deallocated, mutex locks without matching unlocks along all control flow paths, etc. We should emphasize that Check 'n' Crash is not a bug pattern matcher: it has only a basic preconceived notion of what the program text of a bug would look like. Thus, the domain of application

of Check 'n' Crash is different from bug pattern matchers, concentrating more on numeric properties, array indexing violations, errors in class casting, etc. Furthermore, Check 'n' Crash is likely to find new and unusual errors, often idiomatic of a programming or design style. Thus, it is interesting to use Check 'n' Crash to find a few potential errors in an application and then search for occurrences of these errors with a bug pattern matcher. A practical observation is that bug pattern matchers may not need to be very sophisticated in order to be useful: the greater value is often in identifying the general bug pattern, rather than in searching the program for the pattern.

We have already mentioned the strengths of Check 'n' Crash. It is a sound tool at the language level: any error reported can indeed happen for some combination of method inputs. It searches for possible error-causing inputs much more efficiently than JCrasher. It gives concrete, easy-to-inspect counterexamples. Nevertheless, Check 'n' Crash also has shortcomings. Although it is sound with respect to program execution semantics, it still suffers from false positives when the inputs are precluded by an unstated or informal precondition (e.g., JavaDoc comments), which we will address in Chapter 6. But, as we mentioned earlier, no automatic error checking system can solve this problem fully. Nevertheless, Check 'n' Crash possibly suffers more than bug pattern matching tools in this regard because it has no domain-specific or context knowledge. In contrast, a bug pattern matcher can often discover errors that are bugs with high probability: e.g., the use of an object after it has been freed. Nevertheless, due to the complexity of common bug patterns (e.g., needing to match data values and to recognize all control flow paths), bug pattern matchers typically suffer in terms of soundness. We speculate that users may be more willing to accept false positives due to unstated preconditions than due to unsoundness in the modeling of program execution. Another weakness of Check 'n' Crash is that it is less complete than ESC/Java because it cannot always derive concrete test cases from the

Simplify counterexamples. We have found that in practice we still prefer the higher incompleteness of Check 'n' Crash to the many spurious warnings of ESC/Java.

CHAPTER VI

SOLUTION FRONT-END: OBSERVE USAGE INVARIANTS TO FOCUS THE BUG SEARCH

After addressing language-level path imprecision in Chapter 5, we now turn to user-level path imprecision in this chapter. One might argue that a language-level precise analysis can always simulate a user-level precise analysis—when expressing the additional user-level constraints as explicit program assertions. If this were feasible in practice, we would be done with precision after Chapter 5. The reason why we need to separately address user-level precision is that user specifications are typically not available in a formal form in most of real-world software engineering. This is in sharp contrast with the specification of the host language itself. Java, for example, has a somewhat formal and fixed specification, which is therefore available to static analysis tools such as ESC/Java. ESC/Java can take advantage of much formal user specification, if it were available, in the form of JML specifications. In this chapter we attack this dilemma by trying to milk existing test cases for user specifications via dynamic invariant inference. We implement this technique in the DSD-Crasher tool [25, 95, 25]. Implementing this in the context of the object-oriented programming language Java will lead us to an interesting problem, which we will describe and address in the following chapter.

6.1 Overview

Dynamic program analysis offers the semantics and ease of concrete program execution. Static analysis lends itself to obtaining generalized properties from the program text. The need to combine the two approaches has been repeatedly stated in the

software engineering community [9, 24, 39, 113, 118]. Here, we describe DSD-Crasher [25, 95, 27]: a bug-finding tool that uses dynamic analysis to infer likely program invariants, explores the space defined by these invariants exhaustively through static analysis, and finally produces and executes test cases to confirm that the behavior is observable under some real inputs and not just due to overgeneralization in the static analysis phase. Thus, our combination has three steps: dynamic inference, static analysis, and dynamic verification (*DSD*).

More specifically, we employ the Daikon tool [40] to infer likely program invariants from an existing test suite. The results of Daikon are exported as JML annotations [71] that are used to guide our Check 'n' Crash tool [24]. Daikon-inferred invariants are not trivially amenable to automatic processing, requiring some filtering and manipulation (e.g., for internal consistency according to the JML behavioral subtyping rules, as discussed in Chapter 7). Check 'n' Crash employs the ESC/Java static analysis tool [42], applies constraint-solving techniques on the ESC/Java-generated error conditions, and produces and executes concrete test cases. The exceptions produced by the execution of generated test cases are processed in a way that takes into account which methods were annotated by Daikon, for more accurate error reporting. For example, a `NullPointerException` is not considered a bug if thrown by an un-annotated method, instead of an annotated method; otherwise, many false bug reports would be produced: ESC/Java produces an enormous number of warnings for potential `NullPointerException`s when used without annotations [91].

Several past research tools follow an approach similar to ours, but omit one of the three stages of our analysis. Check 'n' Crash is a representative of a static-dynamic (SD) approach. There are several representatives of a DD approach, with the closest one (because of the concrete techniques used) being the Eclat tool [88]. Just as our DSD approach, Eclat produces program invariants from test suite executions using Daikon. Eclat also generates test cases and disqualifies the cases that violate inferred

preconditions. Nevertheless, there is no static analysis phase to exhaustively attempt to explore program paths and yield a directed search through the test space. Instead, Eclat's test case generation is largely random. Finally, a DS approach is implemented by combinations of invariant detection and static analysis. A good representative, related to our work, is the Daikon-ESC/Java (DS) combination of [86].

The benefit of DSD-Crasher over past approaches is either in enhancing the ability to detect bugs, or in limiting false bug warnings. For instance, compared to Check 'n' Crash, DSD-Crasher produces more precise error reports with fewer false bug warnings. Check 'n' Crash is by nature local and intra-procedural when no program annotations are employed. As the Daikon-inferred invariants summarize actual program executions, they provide assumptions on correct code usage. Thus, DSD-Crasher can disqualify illegal inputs by using the precondition of the method under test to exclude cases that violate common usage patterns. As a secondary benefit, DSD-Crasher can concentrate on cases that satisfy called methods' preconditions. This increases the chance of returning from these method calls normally and reaching a subsequent problem in the calling method. Without preconditions, Check 'n' Crash is more likely to cause a crash in a method that is called by the tested method before the subsequent problematic statement is reached. Compared to the Eclat tool, DSD-Crasher can be more efficient in finding more bugs because of its deeper static analysis, relative to Eclat's mostly random testing.

To demonstrate the potential of DSD-Crasher, we apply it to medium-size third-party applications (the Groovy scripting language and the JMS module of the JBoss application server). We show that, under controlled conditions (e.g., for specific kinds of errors that match well the candidate invariants), DSD-Crasher is helpful in removing false bug warnings relative to just using the Check 'n' Crash tool. Overall, barring engineering hurdles, we found DSD-Crasher to be an improvement over

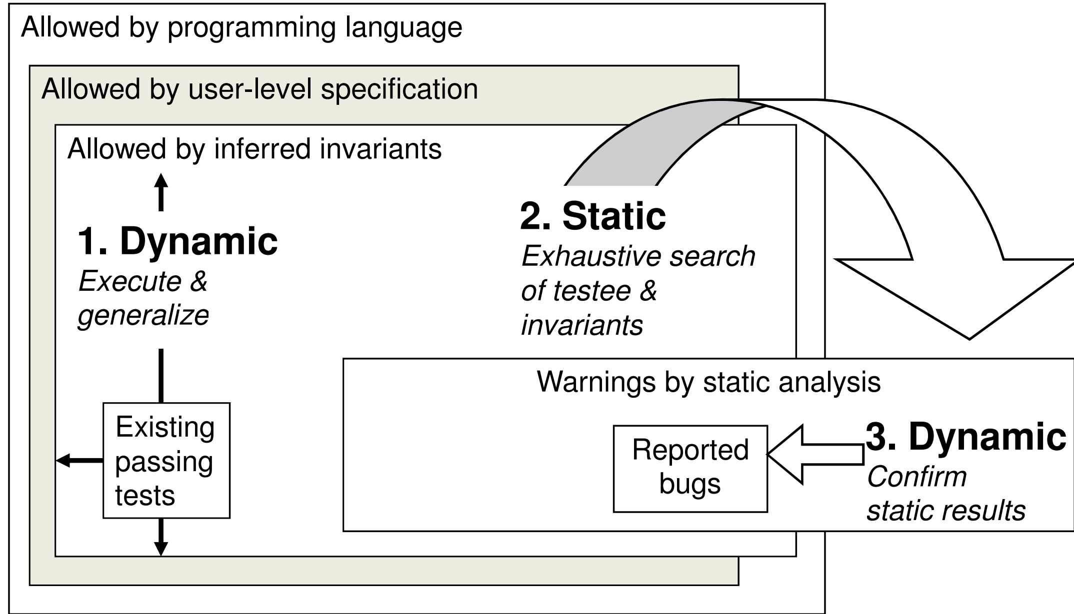


Figure 8: Overview of DSD-Crasher's three-stage program analysis in terms of program values and execution paths. The goal of the first dynamic step is to infer the testee's informal specification. The static step may generalize this specification beyond possible executions, while the final dynamic step will restrict the analysis to realizable problems. Each box represents a program domain. An arrow represents a mapping between program domains performed by the respective analysis. Shading should merely increase readability.

Check 'n' Crash, provided that the application has a regression test suite that exercises exhaustively the functionality under test. At the same time, the approach can be more powerful than Eclat, if we treat the latter as a bug finding tool. The static analysis can allow more directed generation of test cases and, thus, can uncover more errors in the same amount of time.

6.2 Dynamic-static-dynamic analysis pipeline

We use a dynamic-static-dynamic combination of analyses in order to increase the confidence in reported faults—i.e., to increase soundness for incorrectness. The main idea is that of using a powerful, exhaustive, but unsound static analysis, and then improving soundness externally using dynamic analyses.

Figure 8 illustrates the main idea of our DSD combination. The first dynamic

analysis step generalizes existing executions. This is a heuristic step, as it involves inferring abstract properties from specific instances. Nevertheless, a heuristic approach is our only hope for improving soundness for incorrectness. We want to make it more likely that a reported and reproducible error will not be dismissed by the programmer as "outside the intended domain of the method". If the "intended domain" of the method (i.e., the range of inputs that constitute possible uses) were known from a formal specification, then there would be no need for this step.

The static analysis step performs an exhaustive search of the space of desired inputs (approximately described by inferred properties) for modules or for the whole program. A static analysis may inadvertently consider infeasible execution paths, however. This is a virtually unavoidable characteristic of static analyses—they cannot be sound both for correctness and for incorrectness; therefore they will either miss errors or overreport them. Loops, procedure calls, pointer aliasing, and arithmetic are common areas where analyses are only approximate. Our approach is appropriate for analyses that tend to favor exhaustiveness at the expense of soundness for incorrectness.

The last dynamic analysis step is responsible for reifying the cases reported by the static analysis and confirming that they are feasible. If this succeeds, the case is reported to the user as a bug. This ensures that the overall analysis will only report reproducible errors.

Based on our earlier terminology, the last dynamic step of our approach addresses language-level soundness, by ensuring that executions are reproducible for *some* input. The first dynamic step heuristically tries to achieve user-level soundness, by making sure that the input "resembles" other inputs that are known to be valid.

6.3 Tool background: Daikon

Daikon [40] tracks a testee's variables during execution and generalizes their observed behavior to invariants—preconditions, postconditions, and class invariants. Daikon instruments a testee, executes it (for example, on an existing test-suite or during production use), and analyzes the produced execution traces. At each method entry and exit, Daikon instantiates some three dozen invariant templates, including unary, binary, and ternary relations over scalars, and relations over arrays (relations include linear equations, orderings, implication, and disjunction) [40, 87]. For each invariant template, Daikon tries several combinations of method parameters, method results, and object state. For example, it might propose that some method `m` never returns null. It later ignores those invariants that are refuted by an execution trace—for example, it might process a situation where `m` returned null and it will therefore ignore the above invariant. So Daikon summarizes the behavior observed in the execution traces as invariants and generalizes it by proposing that the invariants might hold in all other executions as well. Daikon can annotate the testee's source code with the inferred invariants as JML preconditions, postconditions, and class invariants [71].

6.4 Design and scope of DSD-Crasher

DSD-Crasher works by first running a regression test suite over an application and deriving invariants using a modified version of Daikon. These invariants are then used to guide the reasoning process of Check 'n' Crash, by influencing the possible errors reported by ESC/Java. The constraint solving and test case generation applied to ESC/Java-reported error conditions remains unchanged. Finally, a slightly adapted Check 'n' Crash back-end runs the generated test cases, observes their execution, and reports violations. Figure 9 illustrates this process with an example.

The scope of DSD-Crasher is the same as that of its component tools. In brief, the tool aims to find errors in sequential code, with fixed-depth loop unrolling used

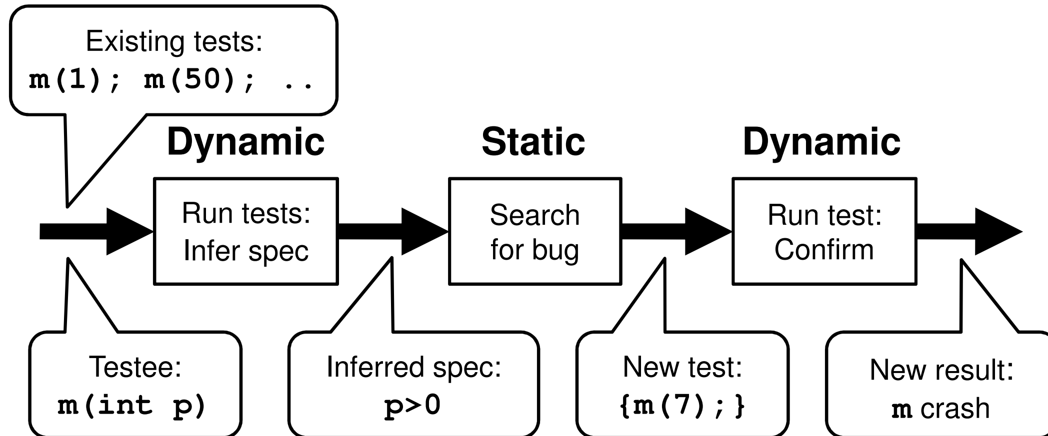


Figure 9: DSD-Crasher workflow: From Java testee and test case source file input to test case source file and exception report output. DSD-Crasher adds a dynamic analysis step at the front of the pipeline, to infer the intended program behavior from existing test cases. It feeds inferred invariants to Check 'n' Crash by annotating the testee. This enables DSD-Crasher to suppress bug warnings that are not relevant to the intended uses of the program. In this example, the inferred invariant excludes negative input values. DSD-Crasher therefore does not produce a warning about -1 causing an exception as Check 'n' Crash did in Figure 7.

to explore infinite loop paths. The errors that can be detected are of a few specific kinds [24]:

- Assigning an instance of a supertype to an array element.
- Casting to an incompatible type.
- Accessing an array outside its domain.
- Allocating an array of negative size.
- Dereferencing null.
- Division by zero.

These cases are statically detected using ESC/Java [73, Chapter 4] but they also correspond to Java runtime exceptions (program crashes) that will be caught during JCrasher-initiated testing.

6.4.1 Treatment of inferred invariants as assumptions or requirements

Daikon-inferred invariants can play two different roles. They can be used as *assumptions* on a method's formal arguments inside its body, and on its return value at the method's call site. At the same time, they can also be used as *requirements* on the method's actual arguments at its call site. Consider a call site of a method `int foo(int i)` with an inferred precondition of `i != 0` and an inferred postcondition of `\result < 0` (following JML notation, `\result` denotes the method's return value). One should remember that the Daikon-inferred invariants are only reflecting the behavior that Daikon observed during the test suite execution. Thus, there is no guarantee that the proposed conditions are indeed invariants. This means that there is a chance that Check 'n' Crash will suppress useful warnings (because they correspond to behavior that Daikon deems unusual). In our example, we will miss errors inside the body of `foo` for a value of `i` equal to zero, as well as errors inside a caller of `foo` for a return value greater or equal to zero. We are willing to trade some potential bugs for a lower false positive rate. We believe this to be a good design decision, since false bug warnings are a serious problem in practice. In our later evaluation, we discuss how this trade-off has not affected DSD-Crasher's bug finding ability (relative to Check 'n' Crash) for any of our case studies.

In contrast, it is more reasonable to ignore Daikon-inferred invariants when used as requirements. In our earlier example, if we require that each caller of `foo` pass it a non-zero argument, we will produce several false bug warnings in case the invariant `i != 0` is not accurate. The main goal of DSD-Crasher, however, is to reduce false bug warnings and increase soundness for incorrectness. Thus, in DSD-Crasher, we chose to ignore Daikon-inferred invariants as requirements and only use them as assumptions. That is, we deliberately avoid searching for cases in which the method under test violates some Daikon-inferred precondition of another method it calls. [113] partially follows a similar approach with Daikon-inferred invariants that are used to produce

test cases.

6.4.2 Inferred invariants excluded from being used

DSD-Crasher integrates Daikon and Check 'n' Crash through the JML language. Daikon can output JML conditions, which Check 'n' Crash can use for its ESC/Java-based analysis. We exclude some classes of invariants Daikon would search for by default as we deemed them unlikely to be true invariants. Almost all of the invariants we exclude have to do with the contents of container structures viewed as sets (e.g., "the contents of array `x` are a subset of those of `y`"), conditions that apply to all elements of a container structure (e.g., "`x` is sorted", or "`x` contains no duplicates"), and ordering constraints among complex structures (e.g., "array `x` is the reverse of `y`"). Such complex invariants are very unlikely to be correctly inferred from the hand-written regression test suites of large applications, as in the setting we examine. We inherited (and slightly augmented) our list of excluded invariants from the study of the Jov tool [113]. The Eclat tool [88] excludes a similar list of invariants.

6.4.3 Adaptation and improvement of tools being integrated

To make the Daikon output suitable for use in ESC/Java, we also had to provide JML specifications for Daikon's `Quant` class. Methods of this class appear in many Daikon-inferred invariants. ESC/Java needs the specifications of these methods in order to reason about them when used in such invariants.

DSD-Crasher also modifies the Check 'n' Crash back-end: the heuristics used during execution of the generated test cases to decide whether a thrown exception is a likely indication of a bug and should be reported to the user or not. For methods with no inferred annotations (which were not exercised enough by the regression test suite) the standard Check 'n' Crash heuristics apply, whereas annotated methods are handled more strictly. Most notably, a `NullPointerException` is only considered a bug if the throwing method is annotated with preconditions. This is standard

Check 'n' Crash behavior [24] and doing otherwise would result in many false error reports: as mentioned earlier, ESC/Java produces an enormous number of warnings for potential `NullPointerException`s when used without annotations [91]. Nevertheless, for a Daikon-annotated method, we have more information on its desired preconditions. Thus, it makes sense to report even "common" exceptions, such as `NullPointerException`, if these occur within the valid precondition space. Therefore, the Check 'n' Crash runtime needs to know whether or not a method was annotated with a Daikon-inferred precondition. To accomplish this we extended Daikon's Annotate feature to produce a list of such methods. When an exception occurs at runtime we check if the method on top of the call stack is in this list. One problem is that the call stack information at runtime omits the formal parameter types of the method that threw the exception. Thus, overloaded methods (methods with the same name but different argument types) can be a source for confusion. To disambiguate overloaded methods we use BCEL [1] to process the bytecode of classes under test. Using BCEL we retrieve the start and end line number of each method and use the line number at which the exception occurred at runtime to determine the exact method that threw it.

6.5 Benefits

The motivation applies to the specific features of our tools. DSD-Crasher yields the benefits of a DSD combination compared to just using its composite analysis. This can be seen with a comparison of DSD-Crasher with its predecessor and component tool, Check 'n' Crash. Check 'n' Crash, when used without program annotations, lacks interprocedural knowledge. This causes the following problems:

1. Check 'n' Crash may produce spurious error reports that do not correspond to actual program usage. For instance, a method `forPositiveInt` under test may throw an exception if passed a negative number as an argument: the automatic

testing part of Check 'n' Crash will ensure that the exception is indeed possible and the ESC/Java warning is not just a result of the inaccuracies of ESC/Java analysis and reasoning. Yet, a negative number may never be passed as input to the method in the course of execution of the program, under any user input and circumstances. That is, an implicit precondition that the programmer has been careful to respect makes the Check 'n' Crash test case invalid. Precondition annotations help Check 'n' Crash eliminate such spurious warnings.

2. Check 'n' Crash does not know the conditions under which a method call within the tested method is likely to terminate normally. For example, a method under test might call `forPositiveInt` before performing some problematic operation. Without additional information Check 'n' Crash might only generate test cases with negative input values to `forPositiveInt`. Thus, no test case reaches the problematic operation in the tested method that occurs after the call to `forPositiveInt`. Precondition annotations help Check 'n' Crash target its test cases better to reach the location of interest. This increases the chance of confirming ESC/Java warnings.

Integrating Daikon addresses both of these problems. The greatest impact is with respect to the first problem: DSD-Crasher can be more focused than its predecessor Check 'n' Crash and issue many fewer false bug warnings because of the Daikon-inferred preconditions.

6.6 Metrics for evaluating dynamic-static hybrid tools

An interesting question is how to evaluate hybrid dynamic-static tools. We next discuss several simple metrics and how they are often inappropriate for such evaluation. This section serves two purposes. First, we argue that the best way to evaluate DSD-Crasher is by measuring the end-to-end efficiency of the tool in automatically discovering bugs (which are confirmed by human inspection), as we do in subsequent

sections. Second, we differentiate DSD-Crasher from the Daikon-ESC/Java combination [86].

The main issues in evaluating hybrid tools have to do with the way the dynamic and static aspects get combined. Dynamic analysis excels in narrowing the domain under examination. In contrast, static analysis is best at exploring every corner of the domain without testing, effectively generalizing to all useful cases within the domain boundaries. Thus it is hard to evaluate the integration in pieces: when dynamic analysis is used to steer the static analysis (such as when Daikon produces annotations for Check 'n' Crash), then the accuracy or efficiency of the static analysis may be biased because it operates on too narrow a domain. Similarly, when the static analysis is used to create dynamic inputs (as in Check 'n' Crash) the inputs may be too geared towards some cases because the static analysis has eliminated others (e.g., large parts of the code may not be exercised at all).

We discuss three examples of metrics that we have found to be inappropriate for evaluating DSD-Crasher.

6.6.1 Formal specifications and non-standard test-suites

DSD-Crasher aims at finding bugs in current, medium-sized, third-party software. These testees consist of thousands of lines of code and come with the original developers' test-suites. They have been developed and are used by people other than us. The open-source programs we are aware of do not contain formal specifications. So for classifying bugs we are mainly relying on our subjective judgement, source code comments, and some external prose. This approach is explicitly dissimilar from previous evaluations such as the ones performed on Eclat [88], which mainly use text book examples, student homeworks, or libraries for which formal specifications were written or already existed. Some of these testees seem to have large non-standard test suites, e.g., geared towards finding programming errors in student homework

submissions. In contrast, typical third-party software is not formally specified and often comes with small test suites.

6.6.2 Coverage

Coverage metrics (e.g., statement or branch coverage in the code) are often used to evaluate the efficiency of analysis and testing tools. Nevertheless, coverage metrics may not be appropriate when using test suites automatically generated after static analysis of the code. Although some static analysis tools, such as Blast [9] and SLAM [5], have been adapted to generate tests to achieve coverage, static analysis tools generally exhaustively explore statements and branches but only report those that may cause errors. ESC/Java falls in this class of tools. The only reported conditions are those that may cause an error, although all possibilities are statically examined. Several statements and paths may not be exercised at all under the conditions in an ESC/Java report, as long as they do not cause an exception.

Consider test cases generated by Check 'n' Crash compared to test cases generated by its predecessor tool, JCrasher. JCrasher will create many more test cases with random input values. As a result, a JCrasher-generated test suite will usually achieve higher coverage than a Check 'n' Crash-generated one. Nevertheless, this is a misleading metric. If Check 'n' Crash did not generate a test case that JCrasher would have, it is potentially because the ESC/Java analysis did not find a possible program crash with these input values. Thus, it is the role of static analysis to intelligently detect which circumstances can reveal an error, and only produce a test case for those circumstances. The result is that parts of the code will not be exercised by the test suite, but these parts are unlikely to contain any of the errors that the static analysis is designed to detect.

6.6.3 Invariant precision and recall

Nimmer and Ernst have performed some of the research closest to ours in combining Daikon and ESC/Java. Reference [87] evaluates how well Daikon (and Houdini) can automatically infer program invariants to annotate a testee before checking it with ESC/Java. Reference [86] also evaluates a Daikon-ESC/Java integration, concentrating more on automatically computed metrics.

In contrast to our path precision and recall metrics of Section 2.3, the main metrics used by Nimmer and Ernst are the *precision* and *recall* of invariants. These are computed as follows. First, Daikon is used to produce a set of proposed invariants for a program. Then, the set of invariants is hand-edited until (a) the invariants are sufficient for proving that the program will not throw unexpected exceptions and (b) the invariants themselves are provable ("verifiable") by ESC/Java. Then "precision" is defined as the proportion of verifiable invariants among all invariants produced by Daikon. "Recall" is the proportion of verifiable invariants produced by Daikon among all invariants in the final verifiable set. Nimmer and Ernst measured scores higher than 90% on both precision and recall when Daikon was applied to their set of testees.

We believe that these metrics are perfectly appropriate for human-controlled environments (as in the Nimmer and Ernst study) but inappropriate for fully automatic evaluation of third-party applications. Both metrics mean little without the implicit assumption that the final "verifiable" set of annotations is near the ideal set of invariants for the program. To see this, consider what really happens when ESC/Java "verifies" annotations. As discussed earlier, the Daikon-inferred invariants are used by ESC/Java as both *requirements* (statements that need proof) and *assumptions* (statements assumed to hold). Thus, the assumptions limit the space of possibilities and may result in a certain false property being proven. ESC/Java will not look outside the preconditions. Essentially, a set of annotations "verified" by ESC/Java means that it is internally consistent: the postconditions only need to hold for inputs

that satisfy the preconditions.

This means that it is trivial to get perfect "precision" and "recall" by just doing a very *bad* job in invariant inference! Intuitively, if we narrow the domain to only the observations we know hold, they will always be verifiable under the conditions that enable them. For instance, assume we have a method `meth(int x)` and a test suite that calls it with values 1, 2, 3, and 10. Imagine that Daikon were to do a bad job at invariant inference. Then a possible output would be the precondition `x=1 or x=2 or x=3 or x=10` (satisfied by all inputs) and some similar postcondition based on all observed results of the executions. These conditions are immediately verifiable by ESC/Java, as it will restrict its reasoning to executions that Daikon has already observed. The result is 100% precision and 100% recall.

In short, the metrics of precision and recall are only meaningful under the assumption that there is a known ideal set of annotations that we are trying to reach, and the ideal annotations are the only ones that we accept as verifiable. Thus, precision and recall will not work as automatable metrics that can be quantified for reasonably-sized programs.

6.6.4 Goals

We want to explore two questions.

1. Can DSD-Crasher eliminate false bug warnings produced by Check 'n' Crash? Reducing false bug warnings with respect to a static-dynamic tool such as Check 'n' Crash was the main goal of DSD-Crasher.

2. Does DSD-Crasher find deeper bugs than similar approaches that use a lightweight bug search?

This evaluation will not establish that DSD-Crasher is generally better than its competition (in all dimensions). DSD-Crasher trades improvements along the above dimensions with disadvantages on other dimensions, such as the number of bugs found

or execution time. Instead, we would like to find evidence that a dynamic-static-dynamic approach such as DSD-Crasher can provide improved results in some scenarios. Our goal is to provide motivation to use DSD-Crasher as part of a multi-tool approach to automated bug-finding. To investigate the first question, we are looking for cases in which Daikon-inferred invariants help DSD-Crasher rule out cases that likely violate implicit user assumptions. To investigate the second question, we are looking for bugs DSD-Crasher finds that elude a lightweight static analysis such as a mostly random bug search.

6.7 Experience

JBoss JMS is the JMS module of the JBoss open-source J2EE application server (http://www.jboss.org/). It is an implementation of Sun's Java Message Service API [53]. We used version 4.0 RC1, which consists of some five thousand non-comment source statements (NCSS).

Groovy is an open-source scripting language that compiles to Java bytecode. We used the Groovy 1.0 beta 1 version, whose application classes contain some eleven thousand NCSS. We excluded low-level AST Groovy classes from the experiments. The resulting set of testees consisted of 34 classes with a total of some 2 thousand NCSS. We used 603 of the unit test cases that came with the tested Groovy version, from which Daikon produced a 1.5 MB file of compressed invariants. (The source code of the testee and its unit tests are available from http://groovy.codehaus.org/)

We believe that Groovy is a very representative test application for our kind of analysis: it is a medium-size, third-party application. Importantly, its test suite was developed completely independently of our evaluation by the application developers, for regression testing and not for the purpose of yielding good Daikon invariants. JBoss JMS is a good example of a third-party application, especially appropriate for comparisons with Check 'n' Crash as it was a part of Check 'n' Crash's past

evaluation [24]. Nevertheless, the existing test suite supplied by the original authors was insufficient and we had to supplement it ourselves to increase coverage for selected examples.

All experiments were conducted on a 1.2 GHz Pentium III-M with 512 MB of RAM. We excluded those source files from the experiments which any of the tested tools could not handle due to engineering shortcomings.

6.7.1 More precise than the static-dynamic Check 'n' Crash

The first benefit of DSD-Crasher is that it produces fewer false bug warnings than the static-dynamic Check 'n' Crash tool.

6.7.1.1 JBoss JMS

Check 'n' Crash reported five cases, which include the errors reported earlier [24]. Two reports are false bug warnings. We use one of them as an example on how DSD-Crasher suppresses false bug warnings. The `setBytes` method of the testee class `org.jboss.jms.util.JMSMap` uses the potentially negative parameter `length` as the length in creating a new array. Calling `setBytes` with a negative `length` parameter causes a `NegativeArraySizeException`.

```
public void setBytes(String name, byte[] value, int offset, int length)
 throws JMSException
{
  byte[] bytes = new byte[length];
  //..
}
```

We used unit tests that (correctly) call `setBytes` three times with consistent parameter values. DSD-Crasher's initial dynamic step infers a precondition that includes `requires length == daikon.Quant.size(value)`. This implies that the

Table 5: Groovy results: The dynamic-static-dynamic DSD-Crasher vs. the static-dynamic Check 'n' Crash.

	Runtime [min:s]	Exception reports	NullPointer reports
Check 'n' Crash classic	10:43	4	0
Check 'n' Crash relaxed	10:43	19	15
DSD-Crasher	30:32	11	9

`length` parameter cannot be negative. So DSD-Crasher's static step does not warn about a potential `NegativeArraySizeException` and DSD-Crasher does not produce this false bug warning.

6.7.1.2 Groovy

As discussed and motivated earlier, Check 'n' Crash by default suppresses most `NullPointerException`s because of the high number of false bug warnings for actual code. Most Java methods fail if a `null` reference is passed instead of a real object, yet this rarely indicates a bug, but rather an implicit precondition. With Daikon, the precondition is inferred, resulting in the elimination of the false bug warnings.

Table 5 shows these results, as well as the runtime of the tools (confirming that DSD-Crasher has a realistic runtime). All of the tools involved are based on the current Check 'n' Crash implementation, which in addition to the published description [24] only reports exceptions thrown by a method directly called by the generated test case. This restricts Check 'n' Crash's reports to the cases investigated by ESC/Java and removes accidental crashes inside other methods called before reaching the location of the ESC/Java warning. Check 'n' Crash classic is the current Check 'n' Crash implementation. It suppresses all `NullPointerException`s, `IllegalArgumentException`s, etc. thrown by the method under test. DSD-Crasher is our integrated tool and reports any exception for a method that has a Daikon-inferred precondition. Check 'n' Crash relaxed is Check 'n' Crash classic but uses the

same exception reporting as DSD-Crasher.

Check 'n' Crash relaxed reports the 11 DSD-Crasher exceptions plus 8 others. (These are 15 `NullPointerException`s plus the four other exceptions reported by Check 'n' Crash classic.) In 7 of the 8 additional exceptions, DSD-Crasher's ESC/Java step could statically rule out the warning with the help of the Daikon-derived invariants. In the remaining case, ESC/Java emitted the same warning, but the more complicated constraints threw off our prototype constraint solver. In this case, `(-1 - fromIndex) == size` has an expression on the left side, which is not yet supported by our solver. The elimination of the 7 false error reports confirms the benefits of the Daikon integration. Without it, Check 'n' Crash has no choice but to either ignore potential `NullPointerException`-causing bugs or to report them, resulting in a high false bug warning rate.

6.7.2 More efficient than the dynamic-dynamic Eclat

We compare DSD-Crasher with Eclat [88], since it is the most closely related tool available to us. Specifically, Eclat also uses Daikon to observe existing correct executions and employs random test case generation to confirm testee behavior. This is not a perfect comparison, however: Eclat has a broader scope than DSD-Crasher (Section 6.4). So our comparison is limited to only one aspect of Eclat.

6.7.2.1 ClassCastExceptions in JBoss JMS

For the JBoss JMS experiment, the main difference we observed between DSD-Crasher and the dynamic-dynamic Eclat was in the reporting of potential dynamic type errors (`ClassCastExceptions`). The bugs reported [24] were `ClassCastException`s. (Most of the other reports concern `NullPointerException`s. Eclat produces 47 of them, with the vast majority being false bug warnings. DSD-Crasher produces 29 reports, largely overlapping the Eclat ones.)

Table 6 compares the `ClassCastExceptions` found by DSD-Crasher and Eclat.

Table 6: JBoss JMS results: `ClassCastException` (CCE) reports by the dynamic-static-dynamic DSD-Crasher and the dynamic-dynamic Eclat. This table omits all other exception reports as well as all of Eclat's non-exception reports.

	CCE reports	Runtime [min:s]
Eclat-default	0	1:20
Eclat-hybrid, 4 rounds	0	2:37
Eclat-hybrid, 5 rounds	0	3:34
Eclat-hybrid, 10 rounds	0	16:39
Eclat-exhaustive, 500 s timeout	0	13:39
Eclat-exhaustive, 1000 s timeout	0	28:29
Eclat-exhaustive, 1500 s timeout	0	44:29
Eclat-exhaustive, 1750 s timeout	0	1:25:44
DSD-Crasher	3	1:59

As in the other tables, every report corresponds to a unique combination of exception type and throwing source line. We tried several Eclat configurations, also used in our Groovy case study later. Eclat-default is Eclat's default configuration, which uses random input generation. Eclat-exhaustive uses exhaustive input generation up to a given time limit. This is one way to force Eclat to test every method. Otherwise a method that can only be called with a few different input values, such as `static m(boolean)` is easily overlooked by Eclat. Eclat-hybrid uses exhaustive generation if the number of all possible combinations is below a certain threshold; otherwise, it resorts to the default technique (random).

We tried several settings trying to cause Eclat to reproduce any of the cases in which DSD-Crasher has observed a `ClassCastException`. With running times ranging from eighty seconds to over an hour, Eclat was not able to do so. (In general, Eclat does try to detect dynamic type errors: for instance, it finds a potential `ClassCastException` in our Groovy case study. In fairness, however, Eclat is not a tool tuned to find crashes but to generate a range of tests.)

DSD-Crasher produces three distinct `ClassCastException` reports, which include

Table 7: Groovy results: The dynamic-static-dynamic DSD-Crasher vs. the dynamic-dynamic Eclat. This table omits all of Eclat's non-exception reports.

	Exception reports	Runtime [min:s]
Eclat-default	0	7:01
Eclat-hybrid, 4 rounds	0	8:24
Eclat-exhaustive, 2 rounds	2	10:02
Eclat-exhaustive, 500 s timeout	2	16:42
Eclat-exhaustive, 1200 s timeout	2	33:17
DSD-Crasher	4	30:32

the two cases presented in the past [24]. In the third case, class `JMSTypeConversions` throws a `ClassCastException` when the following method `getBytes` is called with a parameter of type `Byte[]` (note that the cast is to a "`byte[]`", with a lower-case "b").

```
public static byte[] getBytes(Object value)
 throws MessageFormatException
{
  if (value == null) return null;
  else if (value instanceof Byte[])
  {
    return (byte[]) value;
  } //..
}
```

6.7.2.2 Groovy

Table 7 compares DSD-Crasher with Eclat on Groovy. DSD-Crasher finds both of the Eclat reports. Both tools report several other cases, which we filtered manually to make the comparison feasible. Namely, we remove Eclat's reports of invariant violations, reports in which the exception-throwing method does not belong to the

testees under test specified by the user, etc.

One of the above reports provides a representative example of why DSD-Crasher explores the test parameter space more deeply (due to the ESC/Java analysis). The exception reported can only be reproduced for a certain non-null array. ESC/Java derives the right precondition and Check 'n' Crash generates a satisfying test case, whereas Eclat misses it. The constraints are: `arrayLength(sources) == 1`, as well as `sources:141.46[i] == null` and `i == 0`. Check 'n' Crash generates the input value `new CharStream[]{null}` that satisfies the conditions, while Eclat just performs random testing and tries the value `null`.

6.7.3 Summary of benefits

The main question of our evaluation is whether DSD-Crasher is an improvement over using Check 'n' Crash alone. The answer from our experiments is positive, as long as there is a regression test suite sufficient for exercising large parts of the application functionality. We found that the simple invariants produced by Daikon were fairly accurate, which significantly aided the ESC/Java reasoning. The reduction in false bug warnings enables DSD-Crasher (as opposed to Check 'n' Crash) to produce reasonable reports about `NullPointerException`s. Furthermore, we never observed cases in our experiments where false Daikon invariants over-constrained a method input domain. This would have caused DSD-Crasher to miss a bug found by Check 'n' Crash. Instead, the invariants inferred by Daikon are a sufficient generalization of observed input values, so that the search domain for ESC/Java is large enough to locate potential erroneous inputs.

Of course, inferred invariants are no substitute for human-supplied invariants. One should keep in mind that we focused on simple invariants produced by Daikon and eliminated more "ambitious" kinds of inferred invariants (e.g., ordering constraints on arrays), as discussed in Section 6.4. Even such simple invariants are sufficient

for limiting the false bug warnings that Check 'n' Crash produces without any other context information.

6.8 Applicability and limitations

Our experience with DSD-Crasher yielded interesting lessons with respect to its applicability and limitations. Generally, we believe that the approach is sound and has significant promise, yet at this point it has not reached sufficient maturity for broad practical use. This may seem to contradict our previously presented experiments, which showcased benefits from the use of DSD-Crasher. It is, however, important to note that those were performed in a strictly controlled, narrow-range environment, designed to bring out the promise of DSD-Crasher under near-ideal conditions. The environment indirectly reveals DSD-Crasher's limitations.

6.8.1 Test suite

An extensive test suite is required to produce reliable Daikon invariants. The user may need to supply detailed test cases with high coverage both of program paths and of the value domain. We searched open-source repositories for software with detailed regression test suites, and used Groovy partly because its suite was one of the largest. A literature review reveals no instance of using Daikon on non-trivial, third-party open-source software to infer useful invariants *with the original test suite* that the software's developers supply.

6.8.2 Scalability

The practical scalability of DSD-Crasher is less than ideal. The applications we examined were of medium size, mainly because scaling to large applications is not easily possible. For instance, Daikon can quickly exhaust the heap when executed on a large application. Furthermore, the inferred invariants slow down the ESC/Java analysis and may make it infeasible within reasonable time bounds.

These shortcomings should be largely a matter of engineering. Daikon's dynamic invariant inference approach is inherently parallelizable, for instance. This is a good property for future architectures and an easy way to eliminate scalability problems due to memory exhaustion. By examining the invariants of a small number of methods only, memory requirements should be low, at the expense of some loss in efficiency, which can be offset by parallelism.

6.8.3 Kinds of bugs caught

As discussed earlier, DSD-Crasher is a tool that aims for high degrees of automation. If we were to introduce explicit specifications, the tool could target any type of error, since it would be a violation of an explicit specification. Explicit specifications require significant human effort, however. Therefore, the intended usage mode of the tool limits its attention to violations of implicit preconditions of language-level operations, which cause run-time exceptions, as described in Section 6.4. Thus, semantic errors that do not result in a program crash but produce incorrect results stay undetected. Furthermore, the thorough (due to the static analysis) but relatively local nature of DSD-Crasher means that it is much better for detecting violations of boundary conditions, than it is for detecting "deep" errors involving complex state and multiple methods.

To illustrate this, we analyzed different versions of a subject (the Apache Xml Security module) from the software-artifact infrastructure repository (SIR), which is maintained by Do et al. [35]. The repository contains several versions of a few medium-sized applications together with their respective test suites and seeded bugs. Several other research groups have used subjects from this repository to evaluate bug-finding techniques. Our results are summarized in Table 8. We found that most of the seeded bugs are too deep for DSD-Crasher to catch. Indeed, about half of the seeded bugs do not even affect ESC/Java's internal reasoning, independently of

Table 8: Experience with SIR subjects. SIR contains three bug-seeded versions of the Apache Xml Security distribution. NCSS are non-commented source statements. For all analyzed subject versions, ESC/Java (with the usual DSD-Crasher settings) produces the same warnings for the unseeded and seeded classes. (For the last version we excluded the one seeded fault labelled "conflicting" from our analysis.) Seeded methods are testee methods that contain at least one SIR seed. Note that this includes cases where the ESC/Java warning occurs before a seeded change, so the seeded bug may not necessarily influence the ESC/Java warning site. "ESC/Java wp" stands for ESC/Java's internal weakest precondition computation when running within DSD-Crasher. The last column gives the number of seeded bugs that change the local backward slice of an ESC/Java warning.

Analyzed version of Apache Xml Security	Size [kNCSS]	ESC/Java warnings		Seeded bugs		
		total	in seeded methods	total	affecting ESC/Java wp	in slice of ESC/Java warning
1.0.4	12.4	111	2	20	10	1
1.0.5 D2	12.8	104	3	19	10	1
1.0.71	10.3	120	5	13	7	2

whether this reasoning leads to a bug warning or not. For instance, for version 1.0.4 of our subject, only 10 of the 20 seeded bugs affect *at all* the logical conditions computed during ESC/Java's analysis. The eventual ESC/Java warnings produced very rarely have any relevance to the seeded bug, even with a liberal "relevance" condition (local backward slice). DSD-Crasher does not manage to produce test cases for any of these warnings.

It is worth examining some of these bugs in more detail, for exposition purposes. An example seeded bug that cannot be detected consists of changing the initial value of a class field. The bug introduces the code

```
boolean _includeComments = true;
```

when the correct value of the field is false. However, this does not affect ESC/Java's reasoning, since ESC/Java generally assumes that a field may contain any value. ESC/Java maps the unseeded and the seeded versions of this field to the same abstract value. Hence the result of ESC/Java's internal reasoning will not differ for the

unseeded and the seeded versions.

As another example, one of the seeded bugs consists of removing the call

```
super(doc);
```

from a constructor. Omitting a call to a method or constructor without specifications does not influence ESC/Java's weakest precondition computation, since current ESC/Java versions assume purity for methods without specifications.

The next case is interesting because the bug is within the scope of DSD-Crasher, yet the changed code influences the weakest precondition produced by ESC/Java only superficially. In the following code, the seeded bug consists of comparing the node type to `Node.ELEMENT_NODE` instead of the correct `Node.TEXT_NODE`.

```
for (int i = 0; i < iMax; i++) {
    Node curr = children.item(i);

    if (curr.getNodeType() == Node.ELEMENT_NODE) {
        sb.append(((Text) curr).getData());
    } ...
```

The ESC/Java analysis of the call to (specification-free) method `getNodeType` results in a fresh unconstrained local variable. This local variable will not be used outside this `if` test. Hence, in both the original and the seeded version, we can simplify the equality tests between an unspecified value and a constant to the same abstract unspecified value. The weakest precondition does not change due to this seeded bug. Nevertheless, the test lies on an intraprocedural path to a warning. ESC/Java warns about a potential class cast exception in the statement under the `if`. Despite the warning, the original method is correct: the path to the cast exception is infeasible. For the erroneous version, DSD-Crasher does not manage to reproduce the error, since it involves values produced by several other methods.

CHAPTER VII

ADDING BEHAVIORAL SUBTYPING TO USAGE-OBSERVING PROGRAM ANALYSIS

When implementing DSD-Crasher in Chapter 6, we realize a problem in the first analysis step, which dynamically detects program invariants from existing executions. This problem is caused by behavioral subtyping and object-oriented programming and is exhibited by current versions of the popular Daikon invariant detection system. This chapter describes the problem and proposes a novel algorithm [26] to address the problem.

7.1 Overview

Several recent tools dynamically infer program invariants from existing executions. Ernst et al. have pioneered the field with Daikon [40], which has prompted follow-up work such as DIDUCE [52] or our own DySy tool [28]. Such tools attempt to monitor a large number of program executions and heuristically infer abstract logical properties of the program. Empirically, the invariant detection approach has proven effective for program understanding tasks. Nevertheless, the greatest value of program specifications is in automating program reasoning tasks. Indeed, Daikon produces specifications in several formal specification languages (e.g., in JML [71] for Java) and the resulting annotations have been used to automatically guide tools such as test case generators [113, 88].

Using inferred invariants automatically in other tools places a much heavier burden on the invariant inference engine. Treating inferred invariants, which are heuristics, as true invariants means that they need to be internally consistent. Otherwise a single

contradiction is sufficient to throw off any automatic reasoning engine (be it a theorem prover, a constraint solver, a model checker, or other) that uses the invariants. Here, we discuss how an invariant detection tool can produce consistent invariants in a language that allows indirection in the calling of code. Object-oriented languages are good representatives, as they allow dynamically determining called code through the mechanism of method overriding. The problem has two facets:

- When a method is called on an object with a different static and dynamic type, should inferred invariants be attributed to the static type, the dynamic type, or a combination?
- How can inferred invariants be consistent under the rule of *behavioral subtyping*, which states that the overriding method should keep or weaken the precondition and keep or strengthen the postcondition of each method it overrides.

We discuss these issues in the context of Java, the JML specification language, and the Daikon invariant inference tool. Similar observations apply to different contexts. We describe a solution and our in-progress implementation of a dynamic invariant inference tool that supports it.

7.2 The Java modeling language JML and behavioral subtying

For realistic applications, JML preconditions exist very rarely. JCrasher does not even consider JML preconditions. The tool's ideal area of application is in code currently being developed, which is highly unlikely to have preconditions specified.

JML enforces the principle of *behavioral inheritance* or *behavioral subtyping* [71] for overriding methods. Informally, behavioral subtyping is the requirement that the overriding method should be usable everywhere the method it overrides can be used. This is a common concept, employed also in program analyzers (e.g., ESC/Java2 [20]) and design methodologies (e.g., "subcontracting" in Design by Contract [79]). To

see behavioral subtyping more formally, consider the following Java code with JML annotations:

```
public class Super {
  //@ requires P;
  //@ ensures Q;
  public void m() {...} }
```

JML requires that a subclass's preconditions and postconditions be specified with the `also` keyword, to denote behavioral subtyping:

```
public class C extends Super {
  //@ also
  //@ requires R;
  //@ ensures S;
  public void m() {...} }
```

`R` and `S` are *not* the precondition and postcondition of method `C.m`, however. Instead, JML derives the preconditions and postconditions from the `also` clauses and the behavior of the overridden method `Super.m`:

- `C.m`'s precondition is `P | R` (read "P or R")
- `C.m`'s postcondition is `(P ==> Q) & (R ==> S)` (read "if P holds as a precondition, Q holds as a postcondition and if R holds as a precondition, S holds as a postcondition").

This is exactly the formal embodiment of behavioral subtyping: the precondition of the subtype method allows all the preconditions of the methods it overrides, plus possibly some more. The postcondition of the subtype method is at least that of the overridden method if the precondition falls inside the original domain, and may also have more constraints.

7.3 *Problem 4: Need to support behavioral subtyping in dynamic invariant detection*

The first issue is whether a precondition or postcondition is really a property of the static or the dynamic type of an object. Daikon associates any method execution with the executed method body, not with the method definition of the call's static receiver. It then infers invariants from the execution trace, maintaining the association with the executed method. According to this behavior, Daikon never infers pre- or postconditions of methods defined by a Java interface, since interface methods do not have a body. Yet, this behavior is counter-intuitive: even though postconditions are a property of the called code, preconditions are established by the calling environment. When these are inferred by actual program behavior, they should also be associated with the type known to the caller, regardless of the actual type of an object.

In the following example, the `Client` class was written against interface `I`. Method `Client.foo` calls `I.m`, so `foo` has to honor all preconditions of `I.m`. (These might be specified informally in the Javadoc comments of `I.m` or elsewhere.) The conditions that hold when method `m` gets called reflect the preconditions of the abstract method `I.m`, and not just those of the called method `Impl.m`.

```
public interface I {
  public void m(int arg); }


class Client {
  void foo(I i) {  //called with i = new Impl()
    i.m(...); } }


public class Impl implements I {
  public void m(int arg) {...} }
```

The second issue with invariant inferencing and overriding is thornier. Since invariant inference is heuristic, it is easy to derive invariants that do not respect behavioral subtyping and, thus, lead to a contradiction. Consider a method `m` defined in a class `Super` and overridden in a subclass `C`. Assume a scenario where, under the observed program behavior, `Super.m` is consistently called with an input value of 1 and always returns (in the observed executions) the output value 1. It is reasonable to infer precondition `i == 1` and postcondition `\result == 1` for `Super.m`. At the same time, if `C.m` is also consistently called with input value 1 and always returns (in the observed executions) the output value 0, then it is reasonable to infer precondition `i == 1` and postcondition `\result == 0` for `C.m`. Daikon just outputs the invariants for both methods, with the `also` clause used for the invariants of `C.m`, as dictated by JML for overriding methods:

```
public class Super {
  //@ requires i == 1;
  //@ ensures \result == 1;
  public int m(int i) {...} }

public class C extends Super {
  //@ also
  //@ requires i == 1;
  //@ ensures \result == 0;
  public int m(int i) {...} }
```

Then, according to the JML rules discussed earlier, the complete invariants for `C.m` become:

- Precondition: `i == 1`
- Postcondition:

`((i==1) ==> (\result==1)) &`

`((i==1) ==> (\result==0))`.

We can simplify this to:

`((i==1) ==> (\result==1) & (\result==0))`,

which is equivalent to: `i != 1`.

The derived precondition means that calling `C.m(1)` is legal. It is impossible for the method to reach its postcondition, though, since the method cannot change the state (`i == 1`) that existed before its own execution. (Any method parameter appearing in a postcondition is evaluated to its value before method execution—it is implicitly enclosed by `\old`). So the method cannot terminate normally without violating its postcondition. But it cannot go into an infinite loop or terminate abnormally by throwing an exception either, since such behavior is ruled out in JML when left unspecified. Thus, every possible implementation of method `C.m` (including the actual implementation observed by Daikon) violates the derived specification.

This contradictory postcondition is very undesirable for any automatic use of the derived specifications. The problem is that *the behavior of the overriding method,* `C.m`*, indirectly reveals that there is more behavior of the overridden method,* `Super.m`*, than seen during the execution of the test suite.*[1] Nevertheless, there is no easy way to take this into account during the inference of the invariants for method `Super.m`. It is tempting to think that there may be a different set of conditions that can be output for `C.m` so that no contradiction occurs. While we could explicitly manipulate the `C.m` invariants to narrow the precondition (in this case to `false`) to address the contradiction, this would also render the invariants of `C.m` useless. The problem is fundamentally with the invariants of `Super.m` and not `C.m`.

[1] The implicit assumption for every dynamic invariant detection tool is that the derived invariants have to be consistent with the observed behavior. So a future execution of the observed behavior should pass any invariant checks compiled from the derived invariants.

7.4 Solution design

To solve both problems described earlier, we design a general algorithm that invariant detection tools can follow. The algorithm is oblivious to the actual strategy of the tool for deriving invariants from executions, and instead concentrates on what method observes which execution (i.e., which input and output values). The algorithm can be described informally as follows: values at input are used for computing the precondition of the method executed (dynamic receiver) and all methods it overrides up to and including the static receiver. Values at output are used to compute the postcondition of the method executed and all methods it overrides as long as the values satisfy the methods' preconditions.

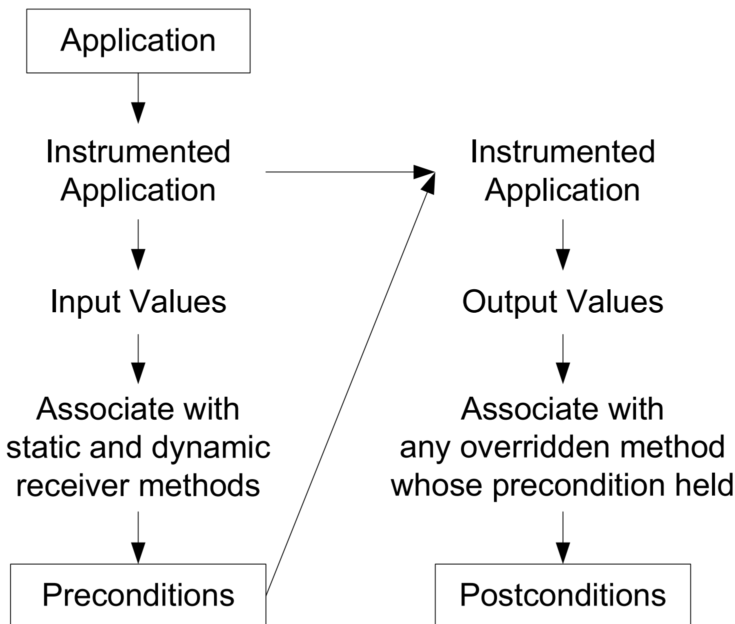


Figure 10: Overview of the proposed two-stage algorithm for dynamic invariant inference. Phase A (left) and Phase B (right).

Figure 10 illustrates the algorithm, which has the following two phases:

- Phase A: the test suite is run and values at the beginning of each call are registered for the dynamic receiver of a method call and for all methods it overrides up to and including the static receiver. Preconditions are inferred from

these values, in the same way as they would be otherwise. No postconditions are inferred. The phase completes with all preconditions computed.

- Phase B: the test suite is run again (or a trace is replayed) and for a call to method `C.m` with inputs `i1..iN` we find all methods (including `C.m` itself) that `C.m` overrides. For each such method, `S.m`, if the inputs `i1..iN` satisfy the inferred precondition of `S.m` then the execution of the method is used in the computation of the postcondition of `S.m`.

This approach solves both problems identified earlier. First, preconditions are computed for both the static and the dynamic receiver of a method call. Second, an execution of an overriding method is also used to compute postconditions for all its overridden methods, as long as the inputs do not fall outside the domain of the overridden method. Since that domain is fixed (from Phase A), we are guaranteed that no contradictory postconditions can be computed (because the overriding and overridden methods have seen the same behavior for all inputs in their common domain).

Note that there are several other ways to remove the symptoms of the second problem (i.e., the derivation of a contradiction). Generally, we can strengthen (i.e., narrow) the inferred precondition of the overriding method or weaken its postcondition until the contradiction disappears. Nevertheless, all such approaches result in artificial overapproximations. In contrast, our above algorithm solves the problem by ensuring that we take into account all relevant behavior for every method when computing its pre- and postcondition.

7.5 Solution implementation

We are in the process of implementing a variant of Daikon using the above algorithm to guide the invariant inference logic. There are several implementation complications in adopting our approach. First, we need to efficiently carry information about the

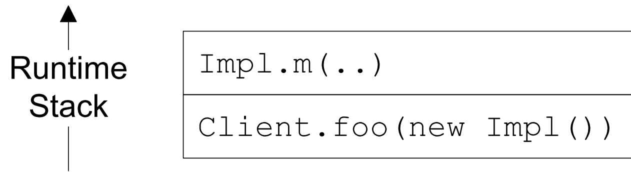


Figure 11: Static receiver problem: The static receiver type `I` is not readily available during the execution of the dynamic receiver `Impl`.

static type used to call a method during method execution. Then, we need to compile inferred JML preconditions into actual runtime checks that we will perform during Phase B of our algorithm to determine which methods' postconditions may be affected by a given execution.

7.5.1 Keeping track of static receivers

For the first issue, consider a method `Impl.m` dynamically dispatched through an interface `I.m`. The problem is that the executed method (`Impl.m`) does not have direct access to the static receiver type (`I`) against which the method was called (Figure 11).

We could rewrite the call site to observe the entry values and possibly update the static receiver's preconditions. Nevertheless, this is awkward, since a similar task also needs to be performed inside the body of the dynamic receiver. Instead, we want to pass the static receiver type information to the dynamic receiver body to use its existing invariant inference routines. To avoid synchronization problems with centralized data stores we transform methods to pass the static receiver type information with the method call (Figure 12). That is, a call to method `m` in class `Impl` will be transformed to calling the jump-through method `m___I` for all its calls through a reference with static type `I`:

```
public interface I {
  public void m(int arg);
  public void m___I(int arg); }
```

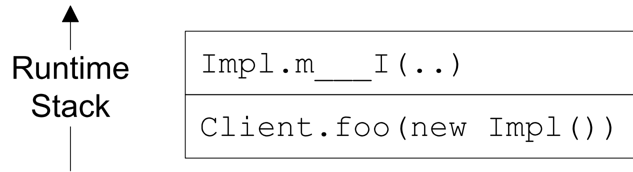


Figure 12: Static receiver solution: Encode the static receiver type `I` in the method name, for example: `m___I`

```
class Client {
  void foo(I i) {  //called with i = new Impl()
    i.m___I(...); } }

public class Impl implements I {
  public void m___I(int arg) {
    trace.add(I);
    return m(arg);
  }
  public void m___Impl(int arg) {
    return m(arg);
  }
  public void m(int arg) {...} }
```

We use BCEL [1] for the program transformation at the bytecode level.

7.5.2 Checking invariants during execution

After Phase A we add the derived preconditions as special checks to the application. At the beginning of each method body we add instructions that check if the precondition of any overridden method holds as well. We associate the traced values with every so determined method.

Up front we determine all method definitions that are overridden by a given

method `D.m`. Note that `m` may override methods in more than one direct super-types (e.g., in interfaces) and in transitive super-types. If `m` has a method body (is not abstract), we compile the precondition derived for every overridden method into a separate runtime check and add it to the beginning of `m`. If a runtime check succeeds then the current invocation also satisfies the precondition of the overridden method and we associate the invocation with this method definition. In the following example, an invocation of method `D.m` may contribute to the postconditions of `C.m`, `B.m`, and `A.m`, in addition to `D.m`.

```
public interface A {
  public void m(int i); }

public interface B extends A {
  public void m(int i); }

public class C {
  public void m(int i) {...} }

public class D extends C implements B {
  public void m(int i) {
    if A.m.pre(i) {trace.add(A);}
    if B.m.pre(i) {trace.add(B);}
    if C.m.pre(i) {trace.add(C);} ... } }
```

CHAPTER VIII

RESETTING CLASS STATE BETWEEN TEST EXECUTIONS

When implementing our first automated testing tool, JCrasher, in Chapter 4, we realize the following problem. JCrasher can generate and execute a huge number of test cases, which may all operate on the same small set of classes, reading and updating fields as implemented in the tested methods and the code called by those methods. This can lead to un-expected dependencies between test cases. While this problem is most prominent for a mass-producer of test cases such as JCrasher, it still applies to any other test case generation tool, such as our Check 'n' Crash and DSD-Crasher tools. But since this problem is most pressing for JCrasher, we investigate it in the context of JCrasher.

8.1 Overview

Most of the JCrasher tests are very short in practice—they consist of a method call with arguments that are returned by other method calls, etc., but typically only up to a depth of three to five. More complex tests are not too meaningful due to the enormous size of the test space for all but the simplest classes.

To enable different test cases to execute without interference in a single virtual machine instance, we explore two main ideas: using a different class loader for each test and modifying the code under test at the bytecode level to expose static re-initialization routines, which get called after the end of each test case. Both techniques aim at re-initializing all static data in the classes under test. That is, our techniques reset the in-memory, user-level state introduced by previous tests. A limitation is

that external state—files or data stored in a database, for example—or static state in system classes cannot be reset. Nevertheless, methods that are sensitive to external or system state are typically too complex to benefit by a random testing approach.

The problem of resetting static state is interesting for applications very different from testing. Although, as we argue later, the case of JCrasher is unusual in that it does not require full correctness: a minor relaxation of correctness only risks introducing some false positives and may be desirable if it yields much better performance. For instance, Reference [34] proposes a Java virtual machine design that allows fast re-initialization without process creation overhead for server applications. In our setting, working with an unmodified, general purpose virtual machine is a huge advantage for the ease of deployment of JCrasher. To our knowledge, the comparative merits of the approaches described here have not been discussed before. Resetting static state for testing scalability is one of the novelties of JCrasher.

8.2 Tool background: JUnit

We first describe the JUnit machinery very briefly. Both of our approaches to resetting static state require minor modifications to JUnit. This is somewhat undesirable since it means that a patched version of JUnit is needed during JCrasher testing. Nevertheless, the changes are very small—a few lines patched—and the main benefits of using JUnit as the JCrasher back-end are preserved: generated test cases remain in JUnit format and if they prove to be interesting for regression testing they can be used with an unmodified version of JUnit.

JUnit automates the execution of test cases and facilitates the collection of test case results, but does not offer any assistance in implementing test cases. The JUnit user provides a test class—a subclass of `junit.framework.TestCase`—with test methods that exercise the functionality of different classes. Figure 13 illustrates how JUnit works. In this example, JUnit represents a test case through a method `test1`

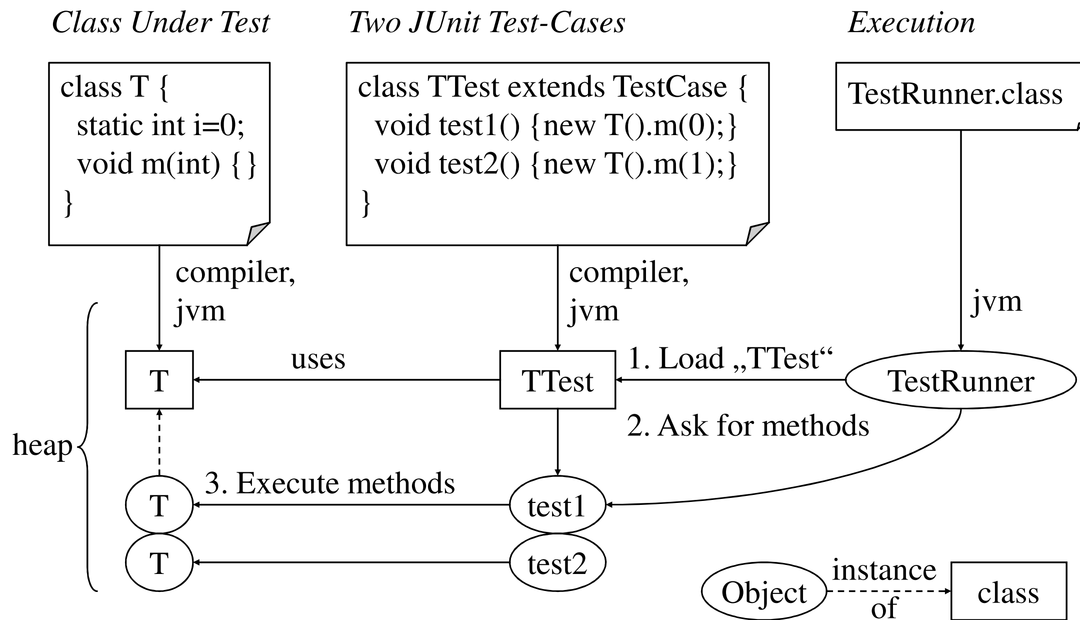


Figure 13: Overview of JUnit's internal data-structures at runtime, when testing testee class `T` with two test cases `test1`, `test2` of a JUnit test-class `TTest`. JUnit loads `TTest` by name via a class loader and retrieves and executes its test cases via reflection. Class `T` is loaded on test case execution as instances of `T` are created. JVM is a standard Java virtual machine.

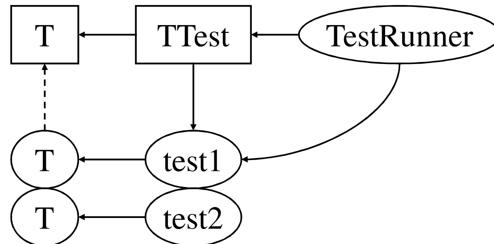
of a test-class `TTest`. The predefined JUnit code uses reflection to find methods whose name begins with "test" and executes them. JCrasher test cases correspond to individual "test" methods as described in Section 4.3.1.

8.3 Problem 5: Need to reset static state during long-running test executions

Since we test small units of functionality for relatively "shallow" errors, it makes sense to perform all its tests in a single instance of the Java Virtual Machine, thus avoiding the expensive tasks of process creation, loading of Java virtual machine classes, etc. Nevertheless, this raises the problem of dependencies among test cases. Without special care, a test case will end up setting the static state on which subsequent test cases will execute. This interference is undesirable: ideally, we would like to know whether a method fails in a well-defined initial state. Otherwise it would be hard to

JUnit

All test-cases test1(), test2() share the same environment, i.e. static state of T



Modified JUnit – junitMultiCL

Each test-case test1(), test2() has its own environment, i.e. class T

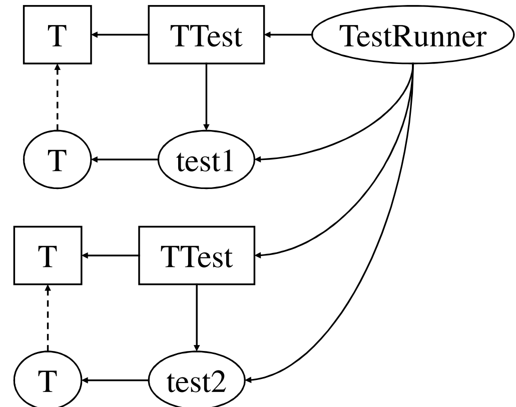


Figure 14: Using multiple class loaders to undo class state change. Left: Original JUnit. Right: Modified JUnit that uses multiple class loaders.

detect the real cause for the violation of a program's invariants.

8.4 First solution: Multiple class loaders

The first option offered by JCrasher for resetting static state consists of loading each test case with a different class loader. Using multiple class loaders is a Java technique for integrating the same code multiple times, with different copies of its static state. In essence, the same class bytecode loaded twice with different class loaders results in two entirely different classes inside a Java VM.

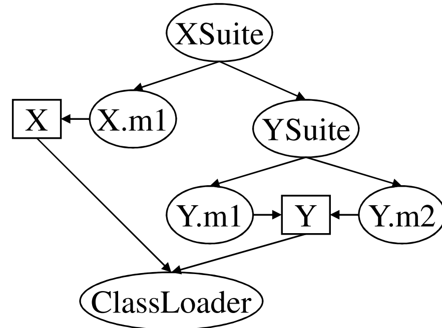
Figure 14 illustrates the concept. It would seem that the modifications to JUnit for integrating the multiple class loaders idea are very minimal and localized. Indeed, our first JUnit patch with early versions of JCrasher consisted of about 10 lines of source code. Nevertheless, we quickly found out that this version was not scalable: due to the JUnit representation of test cases and their interdependencies, older classes never became unreferenced and thus could not be garbage collected. This limited the scalability of JCrasher to a few hundreds of test cases, well below our goal of

JUnit

1. One classloader loads all test-cases and all used classes
2. One suite-hierarchy is executed

Modified JUnit – junitMultiCL

1. Each test-case is loaded by a new classloader
2. The hierarchy is split up into a set of suites
3. The garbage collector can free the heap from executed suites



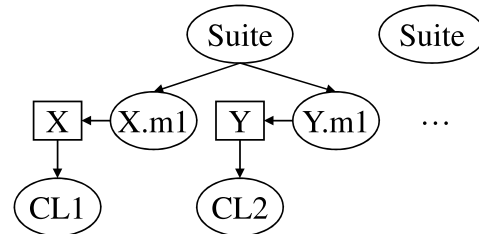


Figure 15: Adapted execution model for using multiple class loaders.

many hundreds of thousands of test cases per Java virtual machine process. Of course, multiple virtual machine instances could be used in sequence, in order to cover all test cases, but this would require more complex coordination outside the JUnit framework—an executable program or script that starts virtual machines, for example—which is an undesirable solution for deployment/portability purposes.

Our final solution changes the JUnit execution model more substantially, as shown in Figure 15. Instead of having a single "Test Suite" object that recursively points to all the tests to be executed, our modified version of JUnit has independent suites, each supporting a small fixed number of test cases—we have been using a single test case per suite. Once a suite gets executed, all memory it occupies—together with loaded classes—can be garbage collected. The only exception is for test cases that actually resulted in errors. In that case, JUnit stores a reference to the offending test case in its error report and the garbage collector cannot reclaim that memory.

The multiple class loader approach is a reasonable compromise between conciseness and efficiency. Nevertheless, it is still rather inefficient since all user classes need to be re-loaded every time just to execute a new test method on a clean slate.

8.5 Second solution: Load-time class re-initialization

A second alternative for resetting static state is to imitate the class initialization algorithm of the Java virtual machine in user space. This will allow re-initializing the same set of classes, existing from previous test cases. This gives much better performance as it saves the time needed for loading and unloading classes. Furthermore, it yields better scalability in terms of memory requirements in the case of test cases resulting in errors. Recall that with multiple loaders, multiple copies of the same classes have to be kept for executions that did cause an error.

A significant insight for the rest of this section is that our solution does not need to be fully correct with respect to the semantics of Java VM initialization, as long as it is correct for the vast majority of test cases. Testing with JCrasher is an optimistic, "nothing-to-lose" approach. The system picks some test inputs randomly and attempts to discover robustness errors. As long as the vast majority of the test cases execute with correct Java semantics, it should not matter if a few false positives—failed tests that do not correspond to errors—are introduced. False negatives matter even less as they are inherent in the random input approach anyway. Thus, it is often preferable to have a very fast but not fully accurate solution. Indeed, the current JCrasher implementation does not perform fully correct static re-initialization although it works correctly for the vast majority of programs. At the end of this section we will discuss how its correctness can be improved.

The key idea is to implement a load-time class initialization algorithm—similar to the one performed by the Java virtual machine [75, Section 2.17.5]—in the JCrasher runtime. Executing this procedure before each test case execution re-initializes the static state of all previously loaded classes each time to the same value. The required elements of such a load-time initialization algorithm are the following. First, a list of the classes in the order in which they should be initialized. Second, the ability to reset the values of the static fields of each of these classes to the default before

each test case execution. All zero bits represents the default value: `null`, zero, or `false`. Finally, the ability to execute the variable initializer of each static field before each test case execution. The variable initializer of a static field is compiled into the `<clinit>()` method of the class declaring the field. The `<clinit>()` method cannot be called by Java code as its name is not a valid method name.

In order to implement class re-initialization, JCrasher again makes some modifications to JUnit. The JUnit class loader is replaced by a special class loader that performs modifications of the class bytecode before loading it—we use BCEL [1] for bytecode engineering. Classes belonging to the Java platform, JUnit, or JCrasher are excluded from the following treatment by the class loader.

- If the class has a `<clinit>()` method, this is copied to methods `_clreinit()` and `_clinit()`. The former method is the re-initializer, while the latter will be used as the original initializer. If no static initializer exists, empty `_clreinit()` and `_clinit()` methods are added to the class.

- The `_clreinit()` method is modified to differ from the original static initializer to avoid attempting to reset final fields.

- A static initializer, `<clinit>()`, method is added to the class. This static initializer calls the original static initialization code, `_clinit()`. On return from that code, the static initializer registers the fact that static initialization ended for this class in a JCrasher-maintained data structure. The ending order of initializations will be the same as the order of re-initializations before future tests.

The re-initialization occurs after the end of each test case execution. JUnit calls via `TestCase.setup()` a JCrasher-runtime function. This function first sets the static fields of the registered classes to all zero bits using Java reflection. Then the `_clreinit()` method of each class is executed. The classes are re-initialized in the

order their original initializations finished in previous tests. This is not correct Java initialization, for two reasons: first because of the possibility of cyclic dependencies between classes and second because this re-initialization is eager—it happens up front.

In the case that static data among classes have cyclic dependencies, the order of re-initialization could be incorrect. In some cases, the loading order of classes could affect the result of static initialization. Thus different test cases could need to be initialized with different static data although they use exactly the same classes—but the code is such that they are loaded in different order. For a standard cyclic dependency example [67], consider the following classes:

```
class A {static int a = B.b + 1;}
class B {static int b = A.a + 1;}
```

In a regular execution, if the initialization of A is started before the initialization of B, then B is initialized before A and a=2, b=1; else a=1, b=2. So if B has been loaded before A and the JCrasher-runtime would run the B-initializer, then the A-initializer: a=1, b=2. Code with cyclic static dependencies is *extremely rare*—the study of Kozen and Stillerman [67] examined a few libraries and found no examples, and such code should be avoided anyway.

The eager character of the JCrasher re-initialization represents a more serious departure from the Java initialization semantics. The Java semantics enforces the lazy loading of classes. Static data get initialized as classes get loaded. Furthermore, static data can be initialized to values dependent on dynamic data computed by the code of previously loaded classes. Thus, eager re-initialization—before other computation has taken place—may yield different values. Nevertheless, most Java programs do not rely on the timing of class loading, so we expect such errors to be rare.

For correct execution semantics, we could simulate the JVM class initialization procedure exactly. This would require extensive bytecode modification, which will

introduce significant overhead. For instance, the event that will trigger the initialization of a class may be the creation of an instance, meaning that all object creations throughout the program under test need to be rewritten to first check whether the instantiated class has been re-initialized. We believe that this approach would be overkill in terms of implementation complexity and overhead, for very little practical benefit. As discussed earlier, the lack of full support for the Java semantics is not a big problem in practice for an optimistic testing approach such as the one of JCrasher. The performance advantages of the re-initialization approach far outweigh the small danger of false positives in the testing process. Furthermore, in practice, it is unlikely that such false positives will be introduced during regular development.

8.6 Experience

We have conducted preliminary experiments on the performance of the different static state resetting techniques. Roughly speaking, the multiple class loading approach is more than twice as fast as re-starting the entire Java VM if only a couple of classes are being reloaded. In contrast, the less safe re-initialization approach of Section 8.5 is over 20 times faster than multiple loading, making the resetting time be negligible with respect to other overheads.

Of course, whether the resetting overhead matters depends on the time that each test takes. Most tests executed when testing an actual application are very brief—each lasts on average a few milliseconds with some being significantly shorter and a few happening to "go deep" in application functionality and take much longer. Of course, the user can select to test the top-level classes of an application, in which case even a single test may take arbitrarily long to complete.

Additionally, the current implementation of our tools is relatively heavyweight. We output all test cases as files on the disk, JUnit has to load each test file and execute it, etc. For efficiency, we produce test files that contain no more than 500

test cases each. One can easily imagine a more specialized, tuned to batch execution runtime, where test cases will be executed with less overhead. Nevertheless, *even in the current, heavyweight execution model, the relative overhead of re-initializing classes can be significant.* This relative overhead would only be more significant with a faster runtime.

More specifically, in our testing environment—a 1.2 GHz Intel mobile Pentium 3 processor with 512 MB RAM running Windows XP and a 12 ms avg., 100 MB/s hard disk—the average time taken to execute a test with JUnit is a little below 5 ms. Starting a Java VM that will execute a trivial class takes 170 ms. Starting a VM and executing a trivial JUnit test takes 270 ms—that is, loading the JUnit classes, the test class and one application class under test and running the JUnit testing code. With the multiple loading approach, using a new loader to re-load a single application class takes 120 ms. That is, the multiple class loaders approach saves the 270 ms of Java virtual machine startup time and JUnit re-loading time. Nevertheless, even going to disk and reloading a single class file introduces enough overhead to reduce the overall benefit to about 150 ms. Still, the multiple class loading approach is more than twice as fast as restarting the Java virtual machine.

The re-initialization approach through calling the static initialization code, discussed in Section 8.5, is significantly faster. The time taken by the JCrasher runtime machinery for re-initializing a class with 10 static fields is 0.06 ms, which is negligible. Although this number depends on the complexity of static initializers, the approach is clearly one of minimal overhead and the time savings are dramatic with respect to the 5 ms it takes JUnit to execute a simple test. Overall, the re-initialization approach enabled the latest version of JCrasher (which generates the most test cases of all our tools) to execute tests an order of magnitude faster than previous versions.

CHAPTER IX

LESSONS LEARNT AND FUTURE WORK

We now take a step back and reflect upon the work described in the previous chapters. First, we highlight some of the trade-offs encountered during the design of the DSD-Crasher program analysis pipeline. Then we look beyond the particular technical context explored by this work. Chapter 1 and Chapter 2 hint at a broader applicability of the core ideas of this work. We outline how these ideas can be generalized to related contexts, that is, programming languages, static analyses, and program properties of interest. Finally, we analyze the different evaluations performed throughout this work. We make several observations that lead us to suggesting future directions in automatic bug finding.

9.1 Design trade-offs

DSD-Crasher is our implementation of the proposed dynamic-static-dynamic program analysis pipeline. Figure 16 gives a high-level overview of the major components of DSD-Crasher. This figure is similar to the earlier overview in Figure 9, but also shows the internals of the static analysis component. This detailed view will serve as illustration for the remainder of this section, where we discuss a few of the design trade-offs that are characteristic of DSD-Crasher.

9.1.1 Low-level interfaces between analysis stages

Figure 16 illustrates that DSD-Crasher consists of a sequence of analysis stages. Each analysis stage takes some data as input, performs its analysis, produces some new data, and passes it on to the following stage. What is characteristic of DSD-Crasher is that it does not pass its data in its internal, high-level representation. Instead,

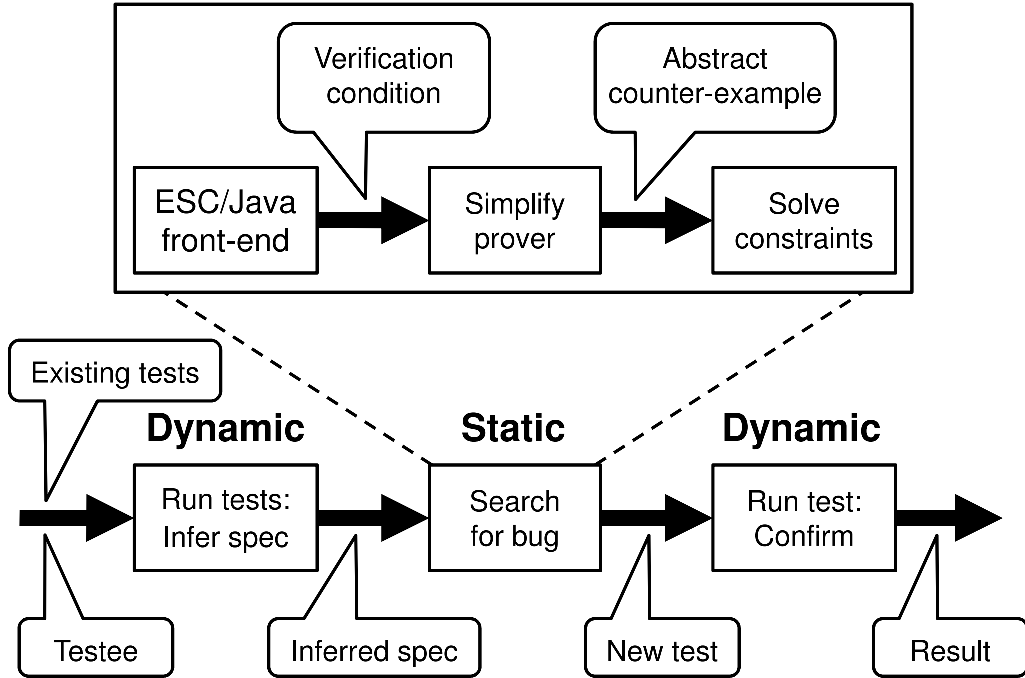


Figure 16: DSD-Crasher design overview: Each bubble represents data, and each box stands for a stage of the analysis pipeline. The static analysis stage is broken out to show its component stages.

each analysis stage in DSD-Crasher uses additional resources to map its data into a low-level format before passing it on to the subsequent analysis stage. The subsequent stage uses a reverse mapping to reconstruct a high-level representation of the data. This is a classic design trade-off between speed and program decomposition. By using separate, low-level data representations for communication, we get a cleaner interface between analysis stages. This makes it easier for us to inspect the data that is passed between the different analysis stages, which eases our development and debugging efforts. It also makes it easier for us to share our intermediate data with third-party analyses.

The interface between the static analysis and the final dynamic analysis is a good example of our low-level interfaces between analysis stages. In the static analysis, DSD-Crasher builds complex in-memory graphs, as described in Section 4.2, and

derives its test cases from these graphs. Now we could just pass these in-memory data-structures to the subsequent dynamic analysis, whose job is to execute and filter the derived test cases. This would have the advantage of fast, in-memory communication between these two analysis stages. Tomb et al. built a related two-stage analysis pipeline for Java programs that uses such a fast, high-level communication between their static and dynamic analysis stages [101]. In contrast, the static analysis stage of DSD-Crasher exports all derived test cases to Java source files that contain JUnit test cases. In the following dynamic stage, DSD-Crasher reads the test cases from the files and rebuilds many of the data structures to execute the test cases. There is no other, more direct, high-level communication between the static and the following dynamic analysis stage in DSD-Crasher. One could argue that the extra effort DSD-Crasher spends on exporting and later re-importing the test cases is wasteful overhead. But we found this hard separation between analysis stages to be very helpful in testing and debugging. It helped us to clearly separate the static analysis stage from the dynamic execution stage. I.e., bugs in the dynamic execution of generated test cases could not affect the static analysis. Tomb et al. [101] arrive at a similar conclusion:

> "The process just described occurs within the same virtual machine as the analysis, and test cases are generated and executed using reflection. On the other hand, JCrasher, for instance, creates external files containing JUnit test cases. In retrospect, JCrasher's approach seems more robust, and we plan to adopt it in the future."

9.1.2 Generating test cases inside the Simplify automated theorem prover

The previous Section 9.1.1 argues for low-level interfaces between the different stages of a program analysis pipeline. It is not immediately clear, though, how to decompose a complex program analysis pipeline into the right component analyses. For example, we refer to DSD-Crasher as a three-stage program analysis pipeline. But the current

DSD-Crasher implementation could also be described as a five-stage analysis pipeline. Figure 16 shows both the three main stages and the three sub-stages of the static analysis stage. So at the implementation level, we may also talk of a dynamic-static-static-static-dynamic program analysis pipeline, DSSSD for short.

The current DSD-Crasher design applies the low-level interface strategy described in the previous Section 9.1.1 not only to the three main stages, but also to the three sub-stages of the static analysis pipeline. I.e., the Simplify theorem prover generates counter-examples to the verification conditions generated by the ESC/Java front-end. Our subsequent test case generation sub-stage parses these counter-examples. Here we encounter a similar design trade-off between communication speed and separation of analysis sub-stages. An alternative strategy would be to integrate the test case generation logic directly into the Simplify theorem prover. Our main reason for not extending the Simplify theorem prover was that Simplify is implemented in Modula-3 [84]. Modula-3 is a good programming language, but the community has largely abandoned it. This means that there is little support for writing Modula-3 code, both in available tools and expertise. Beyond these implementation issues, it would be an interesting direction for future work to extend a traditional automated theorem prover such as Simplify, by adding a full test case generator for an object-oriented programming language such as Java.

9.1.3 Integration into an integrated development environment

Integration into the Eclipse integrated development environment (IDE) was an early goal of the JCrasher random testing system. Early versions of JCrasher provided extensions that enabled this integration. Figure 17 shows a screenshot of the JCrasher integration into the Eclipse IDE. But during the development of JCrasher and the remaining stages of DSD-Crasher, the costs of maintaining this integration into a complex and evolving IDE outweighed the perceived benefits of easier usage. This

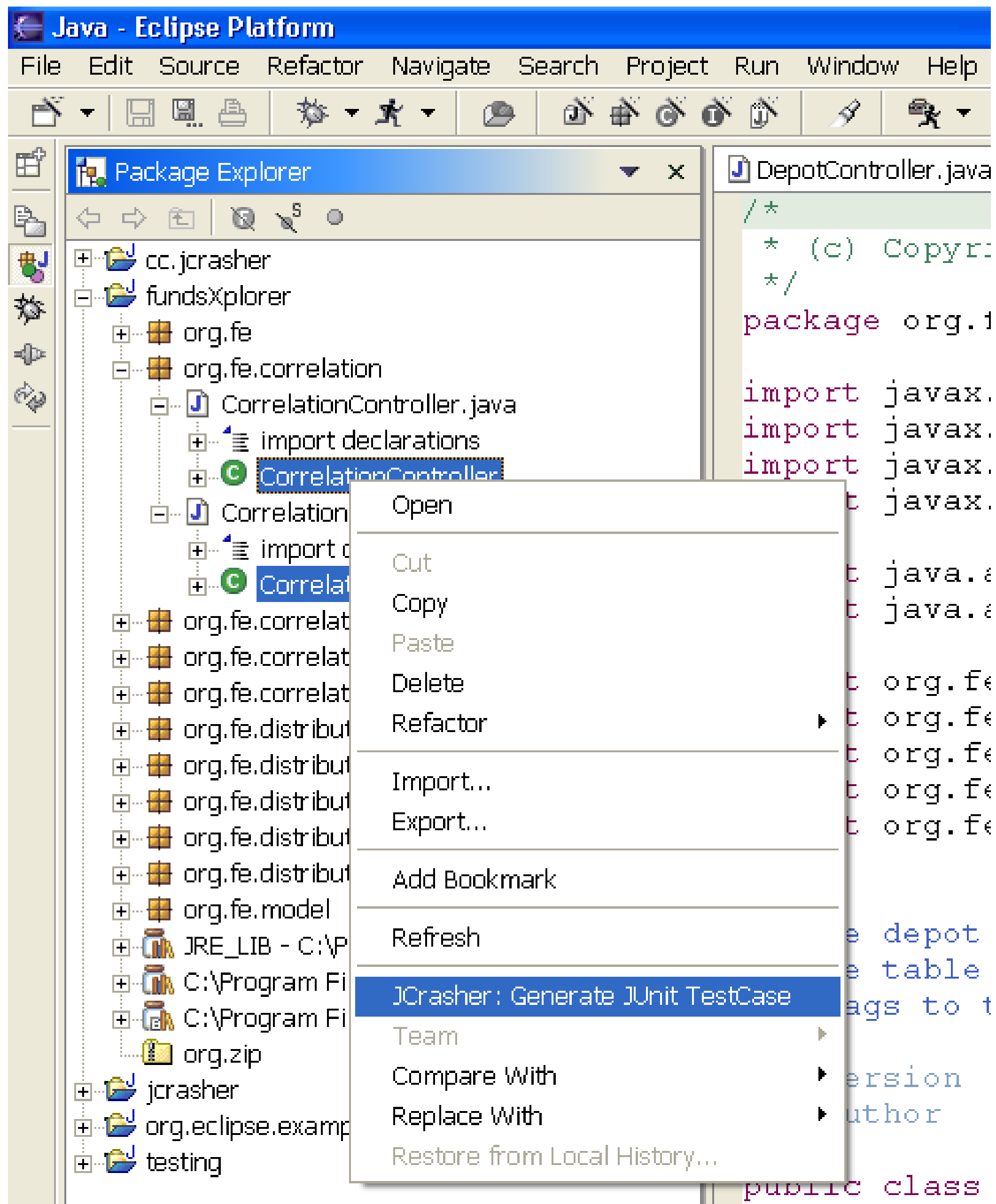


Figure 17: Integration into the Eclipse IDE

view seems to be supported by our users. I.e., several research groups use JCrasher [112, 114, 88, 115, 111, 89, 76], but we are not aware of any researchers using the integration into the Eclipse IDE. Therefore we discontinued the integration into the Eclipse IDE and now consider it a failed experiment. This conclusion should be seen in our context of focusing on usage by researchers, though. We leave it to future work to determine the usefulness of integrating an automated bug finding tool into an integrated development environment for a more representative set of end-users.

9.2 Applying this work to related contexts

Section 1.4 describes the environment of this work, a triple consisting of the Java object-oriented programming language, the ESC/Java static program analysis, and searching for runtime exception bugs. Now we argue that this environment is representative of a wide range of related environments. We briefly describe such related environments and discuss how to generalize our techniques to these environments. We treat each dimension of our environment in turn, first the program properties of interest, then the static program analysis used, and, finally, the programming language analyzed.

9.2.1 Program properties of interest beyond runtime exceptions

The current DSD-Crasher implementation focuses on finding abnormal terminations caused by runtime exceptions. The following describes how we can generalize from finding runtime exceptions to discovering other kinds of problematic behavior.

At a high level, abnormal termination by throwing a runtime exception is just a special case of a program path that violates some correctness condition. For example, dereferencing null violates the correctness condition "variable $\neq$ null" at the time of the dereference. Internally, a program analysis tool such as ESC/Java can model this check for a violation as an assertion that has to hold before the dereference happens. Indeed, ESC/Java translates a pointer dereference to an assertion statement such

as “assert (variable $\neq$ null)” followed by the actual dereference. Similarly, ESC/Java maps the other sources of runtime exceptions in Java to assertion statements, followed by a statement that models the normal behavior of the operation. Besides runtime exceptions, the program analysis may map additional correctness conditions to assertion statements, for example the violation of user-defined correctness conditions.

ESC/Java implements more checks than we are using in the current implementation of DSD-Crasher. For example, ESC/Java can search for potential violations of general, user-defined correctness conditions [42]. The user can annotate the program text with invariants and pre- and post-conditions on the methods and constructors of the analyzed program. These annotations may be expressed in the Java Modeling Language (JML) [71]. Given the imprecise reasoning of ESC/Java, ESC/Java may produce many false warnings on the violation of invariants, pre-conditions, and post-conditions. Now we could use the same machinery of DSD-Crasher to make this search for specification violations more precise. I.e., we could transform the user specification into executable assertions, using JML's specification to assertion translator [16], convert the ESC/Java warnings to test cases using DSD-Crasher, and use the remaining parts of the DSD-Crasher pipeline to execute the generated test cases, to distinguish actual assertion violations from cases that did not violate the user's program specification.

9.2.2 Program analysis: From ESC/Java to FindBugs and Java PathFinder

DSD-Crasher uses the ESC/Java static program analysis tool. But ESC/Java is not the only option to implement a three-stage dynamic-static-dynamic program analysis pipeline such as DSD-Crasher. Many other static program analysis tools exist. Some of them are similar to ESC/Java in that they use weakest precondition calculations at their core. Other tools use different program analysis techniques. In this section we outline how to transfer our approach from a weakest precondition

based technique such as ESC/Java to different techniques, based on dataflow frameworks or model checking approaches. For illustration, we pick representative tools that implement these alternative techniques and outline how we could use them in a three-stage dynamic-static-dynamic program analysis pipeline such as DSD-Crasher. For dataflow based tools we use the FindBugs [57] bug pattern detector, and for model checking techniques we pick Java PathFinder [54, 103, 64, 104, 13].

FindBugs is a bug detection system for Java programs [57]. At its core, FindBugs provides a library of several standard program analyses, including control flow and dataflow analyses. In addition to these basic facilities, FindBugs includes many clients of these program analyses. Each client implements a detector of a specific bug pattern. One example is a detector that checks if a method just calls itself. A call to such a method at runtime will result in an infinite recursive loop, ending in a stack overflow and a program crash. Other detectors look for null pointer dereferences, deadlocks, violations of Java-specific coding standards, and other problems.

The focus of FindBugs is to provide a useful tool for real-world programs. To achieve this goal, Findbugs uses several heuristics to minimize the number of false bug warnings it produces. From our experience, it seems that FindBugs is indeed much better at suppressing false bug warnings than ESC/Java. But, still, FindBugs is not a sound tool and does not guarantee the absence of false bug warnings. To address the problem of false warnings in FindBugs, we can adopt the ideas of this work. I.e., by combining the static analysis of FindBugs with dynamic analyses, we can reduce the number of false bug warnings produced by FindBugs.

We have a prototype, named FB-Crasher, which implements a combination of the static analysis of FindBugs with the dynamic analysis of JCrasher, to implement a two-stage static-dynamic analysis pipeline. FB-Crasher is most closely related to our two-stage analysis pipeline Check 'n' Crash, which is described in Chapter 5. FB-Crasher, like Check 'n' Crash, takes each warning produced by the static analysis

and generates test cases, using JCrasher, to test if the warning of the static analysis can be confirmed with an actual test execution. The current implementation of FB-Crasher differs from Check 'n' Crash in that it is not as tightly integrated with the static analysis stage as Check 'n' Crash is. I.e., FB-Crasher does not get a constraint system from the FindBugs static analysis that would describe the suspected bug in detail. ESC/Java does provide Check 'n' Crash with such constraint systems, which helps Check 'n' Crash to generate test cases that are better targeted than those of FB-Crasher. Still, in very preliminary, non-representative experiments with FindBugs 1.0.0 and FB-Crasher 0.0.1, FB-Crasher was able to reproduce some of the null pointer exception warnings produced by FindBugs. These experiments were conducted on a subset of the regression test suite of FindBugs 1.0.0, which triggered 21 FindBugs warnings. The FB-Crasher generated test cases reproduced 17 of these 21 warnings. FB-Crasher misses two of the 21 warnings, as they require very specific inputs to be reproduced. The remaining two FindBugs warnings, on manual review, proved to be false warnings, warning about null pointer dereferences in dead code. This means that in this small experiment we traded in a relatively small portion of the bug recall of the FindBugs results. For more realistic subjects, we expect that the loss of recall will be higher. But this small experiment also confirms that we can improve the language-level precision of a dataflow based program analysis tool such as FindBugs to 100 percent.

Java PathFinder (JPF) is a well-known program analysis tool for Java [54, 103, 64, 104, 13]. It uses model checking techniques to explore different execution paths of the analyzed program. At the core, JPF uses a static analysis, since it explores a model of the analyzed program, without actually executing the program on a full Java virtual machine, thus omitting native calls and other low-level execution semantics. Given this inaccurate modelling of the Java execution semantics, JPF suffers from similar imprecision problems as ESC/Java. Khurshid et al. showed how to create

test cases from the results of the JPF model checker [64, 104]. This is very similar to the last part of our three-stage pipeline, as implemented in Check 'n' Crash. They take into account preconditions on the programs they analyze. Thus, it seems like a straightforward approach, to extend their current two-stage static-dynamic analysis to a three-stage dynamic-static-dynamic analysis, by adding a usage-oberving dynamic analysis that produces preconditions, as implemented in DSD-Crasher and described in Chapter 6.

9.2.3 Programming language: From Java to C# and C++

C# and C++ are currently the other major object-oriented programming languages, with wide use in practice and research. We argue that the main ideas of this work apply to both C# and C++ and that we expect similar benefits when applied to these languages.

C# [36, 55] and Java are very similar. Both compile to similar bytecode for similar virtual machines [74, 75, 37, 81, 38]. For C#, the term Common Intermediate Language (CIL) is used as a synonym for bytecode, and the term Common Language Runtime (CLR) is used as a synonym for the respective virtual machine implementation by Microsoft. The CLR is an implementation of the Common Language Infrastructure (CLI) virtual machine specification [37, 81, 38]. The virtual machines for Java and C# provide similar services such as enforcing a similar set of type rules and automatic garbage collection. There are no big conceptual differences between program analyses for Java and C#, and our techniques developed in the context of Java should directly apply to C# and similar languages.

C++ [98, 99] is a predecessor of both Java and C#. It is generally considered much more complex than most other languages, including Java and C#. For example, C++ programs are notoriously hard to even parse correctly [107]. A more

fundamental source of additional complexity is that C++ gives the programmer complete freedom to manipulate arbitrary memory locations. But arbitrary changes to memory can circumvent any safety rules programmers are used to from strongly typed languages such as Java and C#. An automatic program analysis that wants to over-approximate program behavior will need to make vast over-approximations in order to remain sound. This results in less precise program analyses. Our techniques make existing program analyses more precise, so we expect an even bigger demand for these techniques for C++ and similar unsafe languages.

A complex feature that may make dynamic invariant detection significantly harder is the C++ template system, which is Turing complete. It is not clear how to map runtime observations back to the original source code in the presence of complex templates.

Implementing our techniques for C++ will also be more challenging because C++ programs are usually compiled to binaries. Binaries are significantly harder to analyze than Java bytecode, since they contain less type information than bytecode does. Implementing our techniques for C++ would therefore require additional techniques to reconstruct type information from binaries [4, 17]. An alternative approach would be for our different analysis steps to work on the source code, as opposed to our current Java implementation, which works mostly on bytecode.

But beyond any such complicating factors, the basic premise remains unchanged. An existing static analysis that over-approximates program behavior will benefit from our technique of combining it with usage-observing and under-approximating ones.

9.3 Critical review of the performed evaluation

A significant portion of this work is dedicated to evaluating the proposed analysis pipeline. Different components of the pipeline are evaluated separately, they are compared to other subsets of the pipeline, and the entire analysis pipeline is compared

Table 9: Overview of the performed evaluations. Every row represents a set of experiments, evaluating one or more tools. **Analysis** refers to our tools, D = JCrasher, S-D = Check 'n' Crash, and D-S-D = DSD-Crasher. **Compare** is the tool we are comparing against, if any. This column uses the same abbreviations as the Analysis column, except here D-S-D = Eclat. **Subjects** distinguishes small from medium-sized subject programs. **Spec** distinguishes exceptional, good, cases where we have a fairly complete, but still informal, specification from the usual case of sparse informal specifications. **Section** and **Table** refer to the respective sections and tables in this document.

Analysis	Compare	Subjects	Spec	Section	Table
D		small	good	4.4.2	1
S-D	D	small	good	5.6	3
S-D		medium		5.7	4
D-S-D	S-D	medium		6.7.1	5
D-S-D	D-S-D	medium		6.7.2	6,7
D-S-D		medium	good	6.8.3	8

to the most closely related work. Table 9 summarizes the performed evaluations. This highlights a few interesting properties of the performed evaluation, which we discuss in the remainder of this section.

9.3.1 Evaluation on large subjects

Table 9 lists several evaluations on small and medium-sized subjects. But we do not have an evaluation on a large subject. The main reason for excluding large subjects from our evaluations are engineering shortcomings in the different stages of the current implementation of the DSD-Crasher analysis pipeline. Section 6.8.2 lists some of the existing engineering shortcomings in DSD-Crasher and ways to address them in future work.

9.3.2 Definite results for medium-sized or large subjects

Table 9 shows only one set of experiments with non-toy subjects that have good specifications. The core problem here is that we are not aware of many realistic, medium-sized or large Java applications that have easily accessible and definite specifications

or bug lists. A notable exception is the software-artifact infrastructure repository (SIR) [35], and we draw from it for one set of experiments. SIR has many good properties, such as good availability and documentation of definite versions of several open-source programs together with seeded bugs. In our case, many of these seeded bugs are unfortunately too subtle for the current DSD-Crasher implementation, as discussed in Section 6.8.3.

The remaining evaluations on non-toy subjects in Table 9 draw subject programs directly from the open source community. Open source software development provides us with good access to source code, which is a great improvement over proprietary software development. But some problems remain. I.e., early versions are often not exposed publically, many interesting bugs may be omitted from public forums because they are handled informally, and specifications are often left unclear and sparse. For these experiments we rely on our own judgement and mainly appeal to intuition to distinguish bugs from other behavior. Making these judgements on demand is one of the most time-consuming aspects of the entire work, since it requires the experimenters to understand a third-party application with all its source code and informal specifications.

It would be beneficial for both current and future experimenters to have access to realistic subjects with definite specifications or sets of bugs. These subjects should include many different versions of a program, including early versions, containing clearly marked bugs of a wide range of complexity. I.e., besides subtle and hard to find bugs they should contain a variety of bugs that are easy to identify for both human consumers and automated program analyses.

9.3.3 Standard benchmark

One interesting property of the evaluations summarized in Table 9 that is not obvious from the table is that several of the experiments reuse subjects from earlier

experiments. I.e., some of the subjects of the JCrasher experiments have been used in related work before and the later Check 'n' Crash and DSD-Crasher experiments reuse subjects from previous evaluations. Using the same subjects for many experiments has the big advantage that it is easier to relate the experiments with each other.

Related research communities, such as the programming language implementation community, have developed a sequence of standard benchmark suites, for example the DaCapo benchmarks by Blackburn et al. [10]. The testing research community, on the other hand, unfortunately, seems less eager to agree upon and later evolve standard benchmarks. The software-artifact infrastructure repository (SIR) [35] could evolve into a standard benchmark for the testing community. It is not yet clear, though, if the testing community will develop and evolve standard benchmark suites with agreed-upon, definite specifications or bug lists, following the pioneering work of SIR.

An existing suite of subjects could be complemented by an agreed-upon random program generator, similar to the ones proposed for evaluating just-in-time (JIT) compilers [46, 117]. Randomly generated programs could be used to evaluate tools that search for runtime exceptions, under the assumption that any runtime exception of a certain type constitutes a bug. This is a common assumption in systems programming (for example, when searching for bugs in the Linux kernel). In our case, we have argued that the user is often only interested in the subset of runtime exceptions that violate some additional, user-defined, correctness condition. Therefore, such randomly generated subject programs should only complement a manually selected and specified corpus of subject programs.

9.3.4 Bug finding competition

A standard benchmark as discussed in Section 9.3.3 would allow the testing community to hold testing competitions, much like other research communities do [83,

29, 7, 6, 3]. The testing community could hold a few, tightly specified competitions, possibly co-located with an existing international conference on testing and program analysis.

The verification community seems to have great success with their annual competitions, e.g., the annual Satisfiability Modulo Theories Competition (SMT-COMP) [7, 6]. They provide a large body of benchmark formulas, in a standard notation, together with a comprehensive set of rules that specify the execution environment and how the results will be interpreted and counted. Several different teams compete each year and their respective tools run on a selected subset of the standard benchmark. The rules imply the results, which are summarized in a quantitative ranking of the contestants.

The cryptography community has used a related form of competition to rank a set of algorithms in the AES competition [83, 29]. The contestants were judged manually by a group of experts, but the competition still had a precise set of rules to guide the evaluation. The programming language research community is currently pursuing a similar competition in the PoplMark challenge [3].

A bug finding competition could focus the community and give us more insight into the comparative strengths of different approaches. Most likely different approaches will have different strengths and weaknesses. Having multiple competition categories could highlight such trade-offs between tools. For example, a given tool may be superior for programs that use complex data, where another tool may be superior for programs that mainly use simple data. For each competition category, the rules should be as specific as possible, defining programming languages, the body of subjects the competition will use, the way results will be interpreted, how to count precision, recall, time and space usage. The rules of the related competitions [83, 29, 7, 6, 3] could serve as a good starting point in developing such a competition.

CHAPTER X

RELATED WORK

There is an enormous amount of work on automated bug-finding tools. We discuss representative recent work below. We deliberately include approaches from multiple research communities in our discussion. Particularly, we compare DSD-Crasher with tools from the testing, program analysis, and verification communities. We believe this is a valuable approach, since many tools produced by these closely related communities have overlapping goals, i.e., to find bugs. We also discuss our choice of component analyses from which we constructed DSD-Crasher. This should highlight that other analysis combinations are possible and may provide superior properties than our concrete instantiation in the DSD-Crasher tool. For the subsequently encountered problems of behavioral subtyping and dynamic invariant detection (Chapter 7) and efficient state resetting (Chapter 8) we have already mentioned the little related work we are aware of in the respective chapters.

10.1 Improving path precision at the language level

Tools in this category use an overapproximating search to find as many bugs as possible. Additionally, they use some technique to reduce the number of false bug warnings, focussing on language-level unsound bug warnings. Our representative of this category is Check 'n' Crash [24].

Tomb et al. [101] present a direct improvement over Check 'n' Crash, by making their overapproximating bug search interprocedural (up to a user-defined call depth). The tool otherwise closely follows the Check 'n' Crash approach by generating test cases to confirm the warnings of their static analysis. On the other hand, their tool neither seems to incorporate pre-existing specifications (which provides an alternative

source of interproceduralty) nor address user-level unsound bug warnings.

Kiniry et al. [66] motivate their recent extensions of ESC/Java2 similarly: "User awareness of the soundness and completeness of the tool is vitally important in the verification process, and lack of information about such is one of the most requested features from ESC/Java2 users, and a primary complaint from ESC/Java2 critics." They list several sources of unsoundness for correctness and incorrectness in ESC/Java2 including less known problems such as Simplify silently converting arithmetic overflows to incorrect results. They propose a static analysis that emits warnings about potential shortcomings of the ESC/Java2 output, namely potentially missing bug reports and potentially unsound bug reports. On the bug detection side their analysis is only concerned with language-level soundness and does not worry about soundness with regard to user-level (and potentially informal) specifications as DSD-Crasher does. DSD-Crasher also provides a more extreme solution for language-level unsound bug reports as it only reports cases that are guaranteed to be language-level sound. We believe our approach is more suitable for automated bug finding since it provides the user with concrete test cases that prove the existence of offending behavior. On the other hand, DSD-Crasher only addresses the unsoundness of ESC/Java2 bug reports. On the sound-for-correctness side, DSD-Crasher would greatly benefit from such static analysis to reduce the possibility of missing real errors. DSD-Crasher needs such analysis even more than ESC/Java2 does, as it may miss sound bug reports of ESC/Java2 due to its limited constraint solving.

Several dynamic tools generate candidate test cases and execute them to filter out false error reports. For example, Xie and Notkin [113] present an iterative process for augmenting an existing test suite with complementary test cases. They use Daikon to infer a specification of the testee when executed on a given test suite. Each iteration consists of a static and a dynamic analysis, using Jtest and Daikon. In the static phase, Jtest generates more test cases, based on the existing specification. In the

dynamic phase, Daikon analyzes the execution of these additional test cases to select those which violate the existing specification—this represents previously uncovered behavior. For the following round the extended specification is used. Thus, the Xie and Notkin approach is also a DSD hybrid, but Jtest's static analysis is rather limited (and certainly provided as a black box, allowing no meaningful interaction with the rest of the tool). Therefore this approach is more useful for a less directed augmentation of an existing test suite aiming at high testee coverage—as opposed to our more directed search for fault-revealing test cases.

Concolic execution [45, 93, 14, 44], uses concrete execution to overcome some of the limitations of symbolic execution [65, 19]. This makes it potentially more powerful than static-dynamic sequences such as Check 'n' Crash. But unlike DSD-Crasher, concolic execution alone does not observe existing test cases and therefore does not address user-level soundness.

SMART, systematic modular automated random testing [44], makes concolic execution more efficient by exploring each method in isolation. SMART summarizes the exploration of a method in pre- and postconditions and uses this summary information when exploring a method that calls a previously summarized method. DSD-Crasher also summarizes methods during the first dynamic analysis step in the form of invariants, which ESC/Java later uses for modular static analysis. DSD-Crasher would benefit from SMART-inferred method summaries for methods that were not covered by our initial dynamic analysis. SMART seems like a natural replacement for the SD-part (ESC/Java and JCrasher) of DSD-Crasher. Designing such a dynamic-concolic tool ("DC-Crasher") and comparing it with DSD-Crasher is part of our future work.

The commercial tool Jtest [90] has an automatic white-box testing mode that generates test cases. Jtest generates chains of values, constructors, and methods in an effort to cause runtime exceptions, just like our approach. The maximal supported

depth of chaining seems to be three, though. Since there is little technical documentation, it is not clear to us how Jtest deals with issues of representing and managing the parameter-space, classifying exceptions as errors or invalid tests, etc. Jtest does, however, seem to have a test planning approach, employing static analysis to identify what kinds of test inputs are likely to cause problems.

10.2 Improving path precision at the language level and the user level

Tools in this category are most similar to DSD-Crasher in that they attack false bug warnings at the language level and at the user level. The common implementation techniques are to infer program specifications from existing test executions and to generate test cases to produce warnings only for language level sound bugs.

Symclat [30] is a closely related tool that, like DSD-Crasher, uses the Daikon invariant detector to infer a model of the testee from existing test cases. Symclat uses Java PathFinder for its symbolic reasoning, which has different tradeoffs than ESC/Java, e.g., Java PathFinder does not incorporate existing JML specifications into its reasoning. Unlike our tools, Symclat has a broader goal of discovering general invariant violations. It appears to be less tuned towards finding uncaught exceptions than our tools since it does not seem to try to cover all control flow paths implicit in primitive Java operations.

Palulu [2] derives method call sequence graphs from existing test cases. It then generates random test cases that follow the call rules encoded in the derived call graphs. Such method call graphs capture implicit API rules (e.g., first create a network session object, then send some initialization message, and only then call the testee method), which are essential in generating meaningful test cases. It would be interesting to integrate deriving such API rules into our first dynamic analysis step.

10.3 *Component analyses*

DSD-Crasher integrates the dynamic Daikon, the static ESC/Java, and the dynamic JCrasher component analyses. But these are certainly not the only component analyses suitable for an automated bug-finding tool like DSD-Crasher. Future variants of DSD-Crasher could be constructed from different components. The following motivates our choice of component analyses and compares them with competing ones.

10.3.1 Inferring specifications to improve user-level path precision

Daikon is not the only tool for invariant inference from test case execution, although it has pioneered the area and has seen the widest use in practice. For instance, the DIDUCE invariant inference tool [52] is optimized for efficiency and can possibly allow bigger testees and longer-running test suites than Daikon. Agitar Agitator [11], a commercial tool, also uses Daikon-like inference techniques to infer likely invariants (termed "observations") from test executions and suggests these observations to developers so that they can manually and selectively promote observations to assertions. Then Agitator further generates test cases to confirm or violate these assertions. Agitator requires manual effort in promoting observations to assertions in order to avoid false warnings of observation violations, whereas our tools concentrate on automated use. Our DySy tool [28] represents a new approach to dynamic invariant detection, which we plan to incorporate in future work. DySy uses the Pex concolic exploration system to track all execution decisions in full detail, via a symbolic shadow state. This enables much more precise invariant inference than current template-based tools can provide.

Inferring method call sequence rules is another valuable approach for capturing implicit user assumptions. For example, [106] presents static and dynamic analyses that automatically infer over- and underapproximating finite state machines of method call sequences. [2] used such finite state machines to generate test cases. [56]

automatically infers *algebraic specifications* from program executions, which additionally include the state resulting from method call sequences. Algebraic specifications express relations between nested method calls, such as *pop(push(obj)) == obj*, which makes them well-suited for specifying container classes. It is unclear, though, how this technique scales beyond container classes. Yet it would be very interesting to design an automated bug-finding tool that is able to process algebraic specifications and compare it with DSD-Crasher.

Taghdiri et al. [100] describe a recent representative of purely static approaches that summarize methods into specifications. Such method summaries would help DSD-Crasher perform deeper interprocedural analysis in its overapproximating bug search component. Summarization approaches typically aim at inferring total specifications, though. So they do not help us in distinguishing between intended and faulty usage scenarios, which is key for a bug-finding tool as DSD-Crasher. Related work by Kremenek et al. [68] infers partial program specifications via a combination of static analysis and expert knowledge. The static analysis is based on the assumption that the existing implementation is correct most of the time. The thereby inferred specifications helped them to correct and extend the specifications used by the commercial bug finding tool Coverity Prevent [22]. Expert knowledge is probably the most important source of good specifications, but also the most expensive one, because it requires manual effort. An ideal bug finding tool should combine as many specification sources as possible, including automated static and dynamic analyses.

10.3.2 The core bug search component: Overapproximating analysis for bug-finding

The Check 'n' Crash and DSD-Crasher approach is explicitly dissimilar to a common class of static analysis tools that have received significant attention in the recent research literature. We call these tools collectively "bug pattern matchers". They are tools that statically analyze programs to detect specific bugs by pattern matching the

program structure to well-known error patterns [51, 116, 57]. Such tools can be quite effective in uncovering a large number of suspicious code patterns and actual bugs in important domains. But the approach requires domain-specific knowledge of what constitutes a bug. In addition, bug pattern matchers often use a lightweight static analysis, which makes it harder to integrate with automatic test case generators. For example, FindBugs does not produce rich constraint systems (like ESC/Java does) that encode the exact cause of a potential bug.

Model-checking techniques offer an alternative approach to overapproximating program exploration and therefore bug searching. Recent model-checkers directly analyze Java bytecode, which makes them comparable to our overapproximating bug search component ESC/Java. Well-known examples are Bogor/Kiasan [31] and Java PathFinder with symbolic extensions [64]. Building on model-checking techniques is an interesting direction for bug-finding and is being explored in the context of JML-like specification languages [32].

Verification tools are powerful ways to discover deep program errors [9, 69]. Nevertheless, such tools are often limited in usability or the language features they support. Related tools enable automatic checking of complex user-defined specifications [60, 102]. Counterexamples are presented to the user in the formal specification language. Their method addresses bug finding for linked data structures, as opposed to numeric properties, object casting, array indexing, etc., as in our approach.

10.3.3 Finding feasible executions

AutoTest [80] is a closely related automatic bug finding tool. It targets the Eiffel programming language, which supports invariants at the language level in the form of contracts [79]. AutoTest generates random test cases, like JCrasher, but uses more sophisticated test selection heuristics and makes sure that generated test cases satisfy given testee invariants. It can also use the given invariants as its test oracle. Our tools

do not assume existing invariants since, unlike Eiffel programmers, Java programmers usually do not annotate their code with formal specifications.

Korat [12] generates all (up to a small bound) non-isomorphic method parameter values that satisfy a method's explicit precondition. Korat executes a candidate and monitors which part of the testee state it accesses to decide whether it satisfies the precondition and to guide the generation of the next candidate. The primary domain of application for Korat is that of complex linked data structures. Given explicit preconditions, Korat will generate deep random tests very efficiently. Thus, Korat will be better than DSD-Crasher for the cases when our constraint solving does not manage to produce values for the abstract constraints output by ESC/Java and we resort to random testing. In fact, the Korat approach is orthogonal to DSD-Crasher and could be used as our random test generator for reference constraints that we cannot solve. Nevertheless, when DSD-Crasher produces actual solutions to constraints, these are much more directed than Korat. ESC/Java analyzes the method to determine which path we want to execute in order to throw a runtime exception. Then we infer the appropriate constraints in order to force execution along this specific path (taking into account the meaning of standard Java language constructs) instead of just trying to cover all paths.

CHAPTER XI

CONCLUSIONS

We have attacked the well-known problem of path-imprecision in static problem analysis. Our starting point was an existing static program analysis that over-approximates the execution paths of the analyzed program. We have made this over-approximating program analysis more precise for automatic testing in an object-oriented programming language. We have achieved this by combining the over-approximating program analysis with usage-observing and under-approximating analyses. More specifically, this dissertation has made the following contributions.

1. We have presented a technique to eliminate language-level unsound bug warnings produced by an execution-path-over-approximating analysis for object-oriented programs that is based on the weakest precondition calculus. Our technique post-processes the results of the over-approximating analysis by solving the produced constraint systems and generating and executing concrete test-cases that satisfy the given constraint systems. Only test-cases that confirm the results of the over-approximating static analysis are presented to the user.

2. Our technique of converting constraint systems to concrete test-cases has the important side-benefit of making the results of a weakest-precondition based static analysis easier to understand for human consumers. We have shown examples from our experiments that visually demonstrate the difference between hundreds of complicated constraints and a simple corresponding JUnit test-case.

3. Besides eliminating language-level unsound bug warnings, we have presented

an additional technique that also addresses user-level unsound bug warnings. This technique pre-processes the testee with a dynamic analysis that takes advantage of actual user data. It annotates the testee with the knowledge obtained from this pre-processing step and thereby provides guidance for the over-approximating analysis.

4. We have presented an improvement to dynamic invariant detection for object-oriented programming languages. Previous approaches do not take behavioral subtyping into account and therefore may produce inconsistent results, which can throw off automated analyses such as the ones we are performing for bug-finding.

5. We have addressed the problem of unwanted dependencies between test-cases caused by global state. We have presented two techniques for efficiently re-initializing global state between test-case executions and have discussed their trade-offs.

6. We have implemented the above techniques in the JCrasher, Check 'n' Crash, and DSD-Crasher tools and have presented initial experience in using them for automated bug finding in real-world Java programs.

APPENDIX A

EXAMPLE OUTPUT OF ESC/JAVA

For the swapArrays example shown in Section 5.3, ESC/Java produces the following list (conjunction) of over 100 constraints, which we have minimally formatted for readability.

```
(arrayLength(firstArray:294.43) <= intLast)
(firstArray:294.43 < longFirst)
(tmp0!new!double[]:297.27 < longFirst)
(secondArray:295.43 < longFirst)
(longFirst < intFirst)
(vAllocTime(tmp0!new!double[]:297.27) < alloc<1>)

(0 < arrayLength(firstArray:294.43))

(null <= max(LS))
(vAllocTime(firstArray:294.43) < alloc)
(alloc <= vAllocTime(tmp0!new!double[]:297.27))
(eClosedTime(elems@loopold) < alloc)
(vAllocTime(out:6..) < alloc)
(vAllocTime(secondArray:295.43) < alloc)
(intLast < longLast)
(1000001 <= intLast)
((intFirst + 1000001) <= 0)
(out:6.. < longFirst)
arrayLength(tmp0!new!double[]:297.27) ==
  arrayLength(firstArray:294.43)
```

```
typeof(0) <: T_int

null.LS == @true

typeof(firstArray:294.43[0]) <: T_double

isNewArray(tmp0!new!double[]:297.27) == @true

typeof(arrayLength(firstArray:294.43)) <: T_int

T_double[] <: arrayType

typeof(firstArray:294.43) == T_double[]

T_double[] <: T_double[]

elemtype(T_double[]) == T_double

T_double <: T_double

typeof(secondArray:295.43) <: T_double[]

typeof(secondArray:295.43) == T_double[]

arrayFresh(tmp0!new!double[]:297.27 alloc alloc<1>
  elems@loopold arrayShapeOne(arrayLength(firstArray:294.43))
  T_double[] F_0.0)

typeof(firstArray:294.43) <: T_double[]

typeof(tmp0!new!double[]:297.27) == T_double[]

(null <= max(LS))

tmp0!new!double[]:297.27[0] == firstArray:294.43[0]

arrayLength(secondArray:295.43) == 0

elems@loopold == elems

elems@pre == elems

state@pre == state

state@loopold == state

m@loopold:299.12 == 0

out@pre:6.. == out:6..

EC@loopold == EC

alloc@pre == alloc
```

```
!typeof(out:6..) <: T_float

!typeof(out:6..) <: T_byte

!typeof(secondArray:295.43) <: T_boolean

!typeof(firstArray:294.43) <: T_double

!typeof(secondArray:295.43) <: T_short

!typeof(tmp0!new!double[]:297.27) <: T_boolean

!typeof(tmp0!new!double[]:297.27) <: T_short

!typeof(firstArray:294.43) <: T_boolean

!typeof(out:6..) <: T_char

!typeof(secondArray:295.43) <: T_double

!typeof(firstArray:294.43) <: T_float

!typeof(out:6..) <: T_int

!isAllocated(tmp0!new!double[]:297.27 alloc)

!typeof(secondArray:295.43) <: T_long

!typeof(firstArray:294.43) <: T_char

!typeof(tmp0!new!double[]:297.27) <: T_double

!typeof(tmp0!new!double[]:297.27) <: T_long

T_double[] != T_boolean

T_double[] != T_char

T_double[] != T_byte

T_double[] != T_long

T_double[] != T_short

T_double[] != T_int

T_double[] != T_float

!typeof(firstArray:294.43) <: T_byte

!typeof(out:6..) <: T_boolean

!typeof(secondArray:295.43) <: T_float

!typeof(out:6..) <: T_short

!typeof(secondArray:295.43) <: T_byte
```

```
!typeof(tmp0!new!double[]:297.27) <: T_float

!typeof(firstArray:294.43) <: T_long

!typeof(tmp0!new!double[]:297.27) <: T_byte

!typeof(out:6..) <: T_double

!typeof(firstArray:294.43) <: T_short

!typeof(out:6..) <: T_long

typeof(out:6..) != T_boolean

typeof(out:6..) != T_char

typeof(out:6..) != T_byte

typeof(out:6..) != T_long

typeof(out:6..) != T_short

typeof(out:6..) != T_int

typeof(out:6..) != T_float

!typeof(secondArray:295.43) <: T_char

!typeof(secondArray:295.43) <: T_int

!typeof(tmp0!new!double[]:297.27) <: T_char

!typeof(firstArray:294.43) <: T_int

!typeof(tmp0!new!double[]:297.27) <: T_int

bool$false != @true

secondArray:295.43 != null

firstArray:294.43 != null

T_double != typeof(out:6..)

T_double != T_double[]

tmp0!new!double[]:297.27 != null
```

REFERENCES

[1] Apache Software Foundation, "Bytecode engineering library (BCEL)." http://jakarta.apache.org/bcel/, Apr. 2003. Accessed June 2008.

[2] Artzi, S., Ernst, M. D., Kieżun, A., Pacheco, C., and Perkins, J. H., "Finding the needles in the haystack: Generating legal test inputs for object-oriented programs," in *Proc. 1st International Workshop on Model-Based Testing and Object-Oriented Systems (M-TOOS)*, Oct. 2006.

[3] Aydemir, B. E., Bohannon, A., Fairbairn, M., Foster, J. N., Pierce, B. C., Sewell, P., Vytiniotis, D., Washburn, G., Weirich, S., and Zdancewic, S., "Mechanized metatheory for the masses: The PoplMark challenge," in *Proc. 18th International Conference on Theorem Proving in Higher Order Logics (TPHOLs)*, pp. 50–65, Springer, Aug. 2005.

[4] Balakrishnan, G. and Reps, T., "Analyzing memory accesses in x86 executables," in *Proc. 13th International Conference on Compiler Construction (CC)*, pp. 5–23, Springer, Feb. 2004.

[5] Ball, T., "Abstraction-guided test generation: A case study," Tech. Rep. MSR-TR-2003-86, Microsoft Research, Nov. 2003.

[6] Barrett, C. W., de Moura, L. M., and Stump, A., "Design and results of the first satisfiability modulo theories competition (SMT-COMP 2005)," *Journal of Automated Reasoning*, vol. 35, pp. 373–390, Nov. 2005.

[7] Barrett, C. W., de Moura, L. M., and Stump, A., "SMT-COMP: Satisfiability modulo theories competition," in *Proc. 17th International Conference on Computer Aided Verification (CAV)*, pp. 20–23, Springer, July 2005.

[8] Beck, K. and Gamma, E., "Test infected: Programmers love writing tests," *Java Report*, vol. 3, pp. 37–50, July 1998.

[9] Beyer, D., Chlipala, A. J., Henzinger, T. A., Jhala, R., and Majumdar, R., "Generating tests from counterexamples," in *Proc. 26th ACM/IEEE International Conference on Software Engineering (ICSE)*, pp. 326–335, IEEE, May 2004.

[10] Blackburn, S. M., Garner, R., Hoffmann, C., Khang, A. M., McKinley, K. S., Bentzur, R., Diwan, A., Feinberg, D., Frampton, D., Guyer, S. Z., Hirzel, M., Hosking, A., Jump, M., Lee, H., Moss, J. E. B., Moss, B., Phansalkar, A., Stefanović, D., VanDrunen, T., von Dincklage, D., and Wiedermann, B., "The DaCapo benchmarks:

Java benchmarking development and analysis," in *Proc. 21st ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pp. 169–190, ACM, Oct. 2006.

[11] Boshernitsan, M., Doong, R., and Savoia, A., "From Daikon to Agitator: Lessons and challenges in building a commercial tool for developer testing," in *Proc. ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, pp. 169–180, ACM, July 2006.

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

[13] Brat, G., Drusinsky, D., Giannakopoulou, D., Goldberg, A., Havelund, K., Lowry, M., Pasareanu, C., Venet, A., Visser, W., and Washington, R., "Experimental evaluation of verification and validation tools on Martian Rover software," *Formal Methods in System Design*, vol. 25, pp. 167–198, Sept. 2004.

[14] Cadar, C., Ganesh, V., Pawlowski, P. M., Dill, D. L., and Engler, D. R., "EXE: Automatically generating inputs of death," in *Proc. 13th ACM Conference on Computer and Communications Security (CCS)*, pp. 322–335, ACM, Oct. 2006.

[15] Chan, P., Lee, R., , and Kramer, D., *The Java Class Libraries*, vol. 1. Addison-Wesley, second ed., 1998.

[16] Cheon, Y. and Leavens, G. T., "A simple and practical approach to unit testing: The JML and JUnit way," in *Proc. 16th European Conference on Object-Oriented Programming (ECOOP)*, pp. 231–255, Springer, June 2002.

[17] Christodorescu, M., Kidd, N., and Goh, W.-H., "String analysis for x86 binaries," in *Proc. 6th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE)*, pp. 88–95, ACM, Sept. 2005.

[18] Claessen, K. and Hughes, J., "QuickCheck: A lightweight tool for random testing of Haskell programs," in *Proc. 5th ACM SIGPLAN International Conference on Functional Programming (ICFP)*, pp. 268–279, ACM, Sept. 2000.

[19] Clarke, L. A., "A system to generate test data and symbolically execute programs," *IEEE Transactions on Software Engineering (TSE)*, vol. 2, no. 3, pp. 215–222, 1976.

[20] Cok, D. R. and Kiniry, J. R., "ESC/Java2: Uniting ESC/Java and JML: Progress and issues in building and using ESC/Java2," Tech. Rep. NIII-R0413, Nijmegen Institute for Computing and Information Science, May 2004.

[21] Cousot, P. and Cousot, R., "Systematic design of program analysis frameworks," in *Proc. 6th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pp. 269–282, ACM, Jan. 1979.

[22] Coverity Inc., "Coverity Prevent." http://www.coverity.com/, 2003. Accessed June 2008.

[23] Csallner, C. and Smaragdakis, Y., "JCrasher: An automatic robustness tester for Java," *Software—Practice & Experience*, vol. 34, pp. 1025–1050, Sept. 2004.

[24] Csallner, C. and Smaragdakis, Y., "Check 'n' Crash: Combining static checking and testing," in *Proc. 27th ACM/IEEE International Conference on Software Engineering (ICSE)*, pp. 422–431, ACM, May 2005.

[25] Csallner, C. and Smaragdakis, Y., "DSD-Crasher: A hybrid analysis tool for bug finding," in *Proc. ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, pp. 245–254, ACM, July 2006.

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

[27] Csallner, C., Smaragdakis, Y., and Xie, T., "DSD-Crasher: A hybrid analysis tool for bug finding," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 17, pp. 1–37, Apr. 2008.

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

[29] Daemen, J. and Rijmen, V., *The Design of Rijndael: AES - The Advanced Encryption Standard*. Springer, first ed., Mar. 2002.

[30] d'Amorim, M., Pacheco, C., Xie, T., Marinov, D., and Ernst, M. D., "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)*, pp. 59–68, IEEE, Sept. 2006.

[31] Deng, X., Lee, J., and Robby, "Bogor/Kiasan: A k-bounded symbolic execution for checking strong heap properties of open systems," in *Proc. 21st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 157–166, IEEE, Sept. 2006.

[32] Deng, X., Robby, and Hatcliff, J., "Kiasan/KUnit: Automatic test case generation and analysis feedback for open object-oriented systems," in *Proc. Testing: Academia and Industry Conference - Practice And Research Techniques (TAIC PART)*, pp. 3–12, IEEE, Sept. 2007.

[33] Detlefs, D., Nelson, G., and Saxe, J. B., "Simplify: A theorem prover for program checking," Tech. Rep. HPL-2003-148, Hewlett-Packard Systems Research Center, July 2003.

[34] Dillenberger, D., Bordawekar, R., Clark, C. W., Durand, D., Emmes, D., Gohda, O., Howard, S., Oliver, M. F., Samuel, F., and John, R. W. S., "Building a java virtual machine for server applications: The jvm on os/390," *IBM Systems Journal*, vol. 39, pp. 194–210, Jan. 2000.

[35] Do, H., Elbaum, S. G., and Rothermel, G., "Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact," *Empirical Software Engineering*, vol. 10, pp. 405–435, Oct. 2005.

[36] Ecma International, *Standard ECMA-334: C# Language Specification*, first ed., Dec. 2001.

[37] Ecma International, *Standard ECMA-335: Common Language Infrastructure (CLI)*, first ed., Dec. 2001.

[38] Ecma International, *Standard ECMA-335: Common Language Infrastructure (CLI)*, fourth ed., June 2006.

[39] Ernst, M. D., "Static and dynamic analysis: Synergy and duality," in *Proc. ICSE Workshop on Dynamic Analysis (WODA)*, pp. 24–27, May 2003.

[40] Ernst, M. D., Cockrell, J., Griswold, W. G., and Notkin, D., "Dynamically discovering likely program invariants to support program evolution," *IEEE Transactions on Software Engineering (TSE)*, vol. 27, pp. 99–123, Feb. 2001.

[41] Flanagan, C. and Leino, K. R. M., "Houdini, an annotation assistant for ESC/Java," in *Proc. International Symposium of Formal Methods Europe (FME)*, pp. 500–517, Springer, Mar. 2001.

[42] Flanagan, C., Leino, K. R. M., Lillibridge, M., Nelson, G., Saxe, J. B., and Stata, R., "Extended static checking for Java," in *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pp. 234–245, ACM, June 2002.

[43] Forrester, J. E. and Miller, B. P., "An empirical study of the robustness of Windows NT applications using random testing," in *Proc. 4th USENIX Windows Systems Symposium*, pp. 59–68, Aug. 2000.

[44] Godefroid, P., "Compositional dynamic test generation," in *Proc. 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pp. 47–54, ACM, Jan. 2007.

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

[46] Goldreich, O., Goldwasser, S., and Micali, S., "How to construct random functions," *Journal of the ACM*, vol. 33, pp. 792–807, Oct. 1986.

[47] Gosling, J., Joy, B., and Steele, G. L., *The Java Language Specification.* Addison-Wesley, first ed., Sept. 1996.

[48] Gosling, J., Joy, B., Steele, G. L., and Bracha, G., *The Java Language Specification.* Prentice Hall, third ed., June 2005.

[49] Gulwani, S., *Program Analysis Using Random Interpretation.* PhD thesis, UC-Berkeley, 2005.

[50] Gulwani, S. and Jojic, N., "Program verification as probabilistic inference," in *Proc. 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pp. 277–289, ACM, Jan. 2007.

[51] Hallem, S., Chelf, B., Xie, Y., and Engler, D., "A system and language for building system-specific, static analyses," in *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pp. 69–82, ACM, June 2002.

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

[53] Hapner, M., Burridge, R., Sharma, R., and Fialli, J., *Java Message Service: Version 1.1.* Sun Microsystems, Inc., Apr. 2002.

[54] Havelund, K. and Pressburger, T., "Model checking Java programs using Java PathFinder," *International Journal on Software Tools for Technology Transfer*, vol. 2, pp. 366–381, Mar. 2000.

[55] Hejlsberg, A., Wiltamuth, S., and Golde, P., *The C# Programming Language.* Addison-Wesley, second ed., June 2006.

[56] Henkel, J., Reichenbach, C., and Diwan, A., "Discovering documentation for Java container classes," *IEEE Transactions on Software Engineering (TSE)*, vol. 33, pp. 526–543, Aug. 2007.

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

[58] Howe, A. E., von Mayrhauser, A., and Mraz, R. T., "Test case generation as an AI planning problem," *Automated Software Engineering*, vol. 4, pp. 77–106, Jan. 1997.

[59] Jackson, D. and Rinard, M., "Software analysis: A roadmap," in *Proc. Conference on The Future of Software Engineering*, pp. 133–145, ACM, June 2000.

[60] Jackson, D. and Vaziri, M., "Finding bugs with a constraint solver," in *Proc. ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, pp. 14–25, ACM, Aug. 2000.

[61] Jensen, K. and Wirth, N., *Pascal: User Manual and Report.* Springer, fourth ed., Sept. 1991.

[62] Kernighan, B. W. and Ritchie, D. M., *The C Programming Language.* Prentice Hall, first ed., Feb. 1978.

[63] Kernighan, B. W. and Ritchie, D. M., *The C Programming Language.* Prentice Hall, second ed., Apr. 1988.

[64] Khurshid, S., Pasareanu, C. S., and Visser, W., "Generalized symbolic execution for model checking and testing," in *Proc. 9th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pp. 553–568, Springer, Apr. 2003.

[65] King, J. C., "Symbolic execution and program testing," *Communications of the ACM*, vol. 19, no. 7, pp. 385–394, 1976.

[66] Kiniry, J. R., Morkan, A. E., and Denby, B., "Soundness and completeness warnings in ESC/Java2," in *Proc. 5th International Workshop on Specification and Verification of Component-Based Systems (SAVCBS)*, pp. 19–24, ACM, Nov. 2006.

[67] Kozen, D. and Stillerman, M., "Eager class initialization for Java," in *Proc. 7th International Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems (FTRTFT)*, pp. 71–80, Springer, Sept. 2002.

[68] Kremenek, T., Twohey, P., Back, G., Ng, A., and Engler, D., "From uncertainty to belief: Inferring the specification within," in *Proc. 7th USENIX Symposium on Operating Systems Design and Implemetation (OSDI)*, pp. 161–176, USENIX, Nov. 2006.

[69] Kroening, D., Groce, A., and Clarke, E. M., "Counterexample guided abstraction refinement via program execution," in *Proc. 6th International Conference on Formal Engineering Methods (ICFEM)*, pp. 224–238, Springer, Nov. 2004.

[70] Kropp, N. P., Koopman, P. J., and Siewiorek, D. P., "Automated robustness testing of off-the-shelf software components," in *Proc. 28th International Symposium on Fault-Tolerant Computing (FTCS)*, pp. 230–239, IEEE, June 1998.

[71] Leavens, G. T., Baker, A. L., and Ruby, C., "Preliminary design of JML: A behavioral interface specification language for Java," Tech. Rep. TR98-06y, Department of Computer Science, Iowa State University, June 1998.

[72] Leino, K. R. M., "Efficient weakest preconditions," Tech. Rep. MSR-TR-2004-34, Microsoft Research, Apr. 2004.

[73] Leino, K. R. M., Nelson, G., and Saxe, J. B., "ESC/Java user's manual," Tech. Rep. 2000-002, Compaq Computer Corporation Systems Research Center, Oct. 2000.

[74] Lindholm, T. and Yellin, F., *The Java Virtual Machine Specification*. Addison-Wesley, first ed., Sept. 1996.

[75] Lindholm, T. and Yellin, F., *The Java Virtual Machine Specification*. Addison-Wesley, second ed., Apr. 1999.

[76] Martin, E., Basu, S., and Xie, T., "Automated testing and response analysis of web services," in *Proc. IEEE International Conference on Web Services (ICWS), Application Services and Industry Track*, pp. 647–654, IEEE, July 2007.

[77] McConnell, S., *Code Complete*. Microsoft Press, second ed., July 2004.

[78] Memon, A. M., Pollack, M. E., and Soffa, M. L., "Hierarchical GUI test case generation using automated planning," *IEEE Transactions on Software Engineering (TSE)*, vol. 27, pp. 144–155, Feb. 2001.

[79] Meyer, B., *Object-Oriented Software Construction*. Prentice Hall, second ed., Apr. 1997.

[80] Meyer, B., Ciupa, I., Leitner, A., and Liu, L., "Automatic testing of object-oriented software," in *Proc. 33rd International Conference on Current Trends in Theory and Practice of Computer Science (SOFSEM)*, pp. 114–129, Springer, Jan. 2007.

[81] Miller, J. S. and Ragsdale, S., *The Common Language Infrastructure Annotated Standard*. Addison-Wesley, first ed., Nov. 2003.

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

[83] Nechvatal, J., Barker, E., Bassham, L., Burr, W., Dworkin, M., Foti, J., and Roback, E., "Report on the development of the advanced encryption standard (AES)," *Journal of Research of the National Institute of Standards and Technology*, vol. 106, pp. 511–577, May 2001.

[84] Nelson, G., *Systems Programming with Modula-3*. Prentice Hall, July 1991.

[85] Nielson, F., Nielson, H. R., and Hankin, C., *Principles of Program Analysis*. Springer, first ed., Nov. 1999.

[86] Nimmer, J. W. and Ernst, M. D., "Automatic generation of program specifications," in *Proc. ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, pp. 229–239, ACM, July 2002.

[87] Nimmer, J. W. and Ernst, M. D., "Invariant inference for static checking: An empirical evaluation," in *Proc. 10th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*, pp. 11–20, ACM, Nov. 2002.

[88] Pacheco, C. and Ernst, M. D., "Eclat: Automatic generation and classification of test inputs," in *Proc. 19th European Conference on Object-Oriented Programming (ECOOP)*, pp. 504–527, Springer, July 2005.

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

[90] Parasoft Inc., "Jtest." http://www.parasoft.com/, Oct. 2002. Accessed June 2008.

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

[92] Schlenker, H. and Ringwelski, G., "POOC: A platform for object-oriented constraint programming," in *Proc. Joint ERCIM/CologNet International Workshop on Constraint Solving and Constraint Logic Programming*, pp. 159–170, Springer, June 2002.

[93] Sen, K., Marinov, D., and Agha, G., "CUTE: A concolic unit testing engine for C," in *Proc. 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*, pp. 263–272, ACM, Sept. 2005.

[94] SilverMark Inc., "Enhanced JUnit version 3.7 user reference." http://www.silvermark.com/Product/java/enhancedjunit/documentation/enhancedjunitmanual.pdf, June 2002.

[95] Smaragdakis, Y. and Csallner, C., "Combining static and dynamic reasoning for bug detection," in *Proc. 1st International Conference on Tests And Proofs (TAP)*, pp. 1–16, Springer, Feb. 2007.

[96] Steensgaard, B., "Points-to analysis in almost linear time," in *Proc. 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pp. 32–41, ACM, Jan. 1996.

[97] Stotts, D., Lindsey, M., and Antley, A., "An informal formal method for systematic JUnit test case generation," in *Proc. 2nd XP Universe and 1st Agile Universe Conference (XP/Agile Universe)*, pp. 131–143, Springer, Aug. 2002.

[98] Stroustrup, B., *The C++ Programming Language.* Addison-Wesley, first ed., Jan. 1986.

[99] Stroustrup, B., *The C++ Programming Language.* Addison-Wesley, special third ed., Feb. 2000.

[100] Taghdiri, M., Seater, R., and Jackson, D., "Lightweight extraction of syntactic specifications," in *Proc. 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*, pp. 276–286, ACM, Nov. 2006.

[101] Tomb, A., Brat, G. P., and Visser, W., "Variably interprocedural program analysis for runtime error detection," in *Proc. ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, pp. 97–107, ACM, July 2007.

[102] Vaziri, M. and Jackson, D., "Checking properties of heap-manipulating procedures with a constraint solver," in *Proc. 9th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pp. 505–520, Springer, Apr. 2003.

[103] Visser, W., Havelund, K., Brat, G. P., Park, S., and Lerda, F., "Model checking programs," *Automated Software Engineering*, vol. 10, pp. 203–232, Apr. 2003.

[104] Visser, W., Păsăreanu, C. S., and Khurshid, S., "Test input generation with Java PathFinder," in *Proc. ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, pp. 97–107, ACM, July 2004.

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

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

[107] Willink, E. D., *Meta-Compilation for C++*. PhD thesis, University of Surrey, June 2001.

[108] Wirth, N., "The programming language Pascal," *Acta Informatica*, vol. 1, pp. 35–63, May 1971.

[109] Wirth, N., *Programming in Modula-2*. Springer, first ed., May 1983.

[110] Wirth, N., *Programming in Modula-2*. Springer, fourth ed., Jan. 1989.

[111] Xie, T., "Augmenting automatically generated unit-test suites with regression oracle checking," in *Proc. 20th European Conference on Object-Oriented Programming (ECOOP)*, pp. 380–403, Springer, July 2006.

[112] Xie, T., Marinov, D., and Notkin, D., "Rostra: A framework for detecting redundant object-oriented unit tests," in *Proc. 19th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 196–205, IEEE, Sept. 2004.

[113] Xie, T. and Notkin, D., "Tool-assisted unit test selection based on operational violations," in *Proc. 18th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 40–48, IEEE, Oct. 2003.

[114] Xie, T. and Notkin, D., "Automatic extraction of object-oriented observer abstractions from unit-test executions," in *Proc. 6th International Conference on Formal Engineering Methods (ICFEM)*, pp. 290–305, Springer, Nov. 2004.

[115] Xie, T. and Notkin, D., "Automatically identifying special and common unit tests for object-oriented programs," in *Proc. 16th IEEE International Symposium on Software Reliability Engineering (ISSRE)*, pp. 277–287, IEEE, Nov. 2005.

[116] Xie, Y. and Engler, D., "Using redundancies to find errors," *IEEE Transactions on Software Engineering (TSE)*, vol. 29, pp. 915–928, Oct. 2003.

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

[118] Young, M., "Symbiosis of static analysis and program testing," in *Proc. 6th International Conference on Fundamental Approaches to Software Engineering (FASE)*, pp. 1–5, Springer, Apr. 2003.

[119] Zitser, M., Lippmann, R., and Leek, T., "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)*, pp. 97–106, ACM, Oct. 2004.