COMBINING STATIC AND DYNAMIC ANALYSIS FOR BUG DETECTION AND PROGRAM UNDERSTANDING

A Dissertation Presented

by

KAITUO LI

Submitted to the Graduate School of the
University of Massachusetts Amherst in partial fulfillment
of the requirements for the degree of

DOCTOR OF PHILOSOPHY

August 31 2016

School of Computer Science

COMBINING STATIC AND DYNAMIC ANALYSIS FOR BUG DETECTION AND PROGRAM UNDERSTANDING

A Dissertation Presented

by

KAITUO LI

Approved as to style and content by:

Yannis Smaragdakis, Chair

George Avrunin, Member

Yuriy Brun, Member

Lori Clarke, Member

James Allan, Department Chair
School of Computer Science

ABSTRACT

COMBINING STATIC AND DYNAMIC ANALYSIS FOR BUG DETECTION AND PROGRAM UNDERSTANDING

AUGUST 31 2016

KAITUO LI

B.Sc., JILIN UNIVERSITY

M.Sc., ZHEJIANG UNIVERSITY

Ph.D., UNIVERSITY OF MASSACHUSETTS AMHERST

Directed by: Professor Yannis Smaragdakis

This work proposes new combinations of static and dynamic analysis for bug detection and program understanding. There are 3 related but largely independent directions: a) In the area of dynamic invariant inference, we improve the consistency of dynamically discovered invariants by taking into account second-order constraints that encode knowledge about invariants; the second-order constraints are either supplied by the programmer or vetted by the programmer (among candidate constraints suggested automatically); b) In the area of testing dataflow (esp. map-reduce) programs, our tool, SEDGE, achieves higher testing coverage by leveraging existing input data and generalizing them using a symbolic reasoning engine (a powerful SMT solver); c) In the area of bug detection, we identify and present the concept of residual investigation: a dynamic analysis that serves as the runtime agent of a static analysis. Residual investigation identifies with higher certainty whether an error reported by the static analysis is likely true.

CHAPTER 1

INTRODUCTION

1.1 My thesis

We can improve the precision and availability of existing program understanding and bug detection techniques by combining static and dynamic analysis. Specifically, we combine static and dynamic analysis in 3 different directions: predictive and precise dynamic analysis with static analysis in bug detection, second-order constraints (static constraints over invariants) in dynamic invariant inference, and dynamic symbolic execution in dataflow program testing.

1.2 Static and dynamic analysis

Software practitioners have traditionally faced many problems with testing, documenting and maintaining software systems. These problems are often the result of poor understanding of program properties, lack of reliable methods for validating software results, over-reliance on specific bug detecting tools, and so on. Various static and dynamic program analysis techniques can be used to remedy these problems. Each flavor of analysis has its own advantages. Dynamic analysis excels in precision and efficiency: it operates by executing programs with test cases, and subsequently is able to compute 100% accurate results with relatively little cost; static analysis is good at generality: we are able to generalize the static analysis results to future executions no matter what the user input is or what environment the system is in [35]. This has led researchers to devise combinations of static and dynamic analyses (e.g., [18,25,26,46,63,103,105]) to achieve the best of both worlds.

Dynamic symbolic execution [92] is such an example. Symbolic execution is a static analysis technique. In a function or method of modern programming languages, a concrete execution comprises input data received as arguments, an execution following a particular path, and output data if this method or function returns one or more values. Instead of using concrete values, symbolic execution represents input and output values by symbolic names. It then computes the path condition for each path using these symbolic names and sends them to a constraint solver. Path conditions are logical formulas that the input values need to satisfy in order to steer the execution into a certain path. Constraint solvers are able to pick values for those symbolic names such that the logical formulas can be made true. With symbolic execution, generating test cases for a path amounts to mapping branch conditions of a path to low-level formulas.

However, symbolic execution cannot reason about native method calls because the source code or the object code of native methods are not available. Also, the ability of symbolic execution to generate test cases heavily depends on the power of constraint solver. Constraint solving is practically always limited in the face of complex constraints such as certain non-linear constraints and certain constraints that use quantifiers. Dynamic symbolic execution resolves these problems by arranging for the symbolic execution and the actual execution of a program to proceed in tandem. Calls to native methods or symbolic names in complex constraints are replaced with concrete values from the actual program execution.

Dynamic symbolic execution is effective at exposing bugs. It can systematically generates test cases from paths of control flow graph of the system under test. If bugs are found, users are able to reproduce all of the bugs with the generated test cases, provided that they are working on a sequential program or a multi-threaded program whose thread interleaving are deterministic. Due to its effectiveness, dynamic symbolic execution has received much attention of late. There has been a lot of dynamic symbolic execution engines in industry and academia, notably DART [46], CUTE [94], jCUTE [93], KLEE [17], Java

PathFinder [108], SAGE [47], PEX [103], BitBlaze [99]. The popularity of dynamic symbolic execution is sharply illustrated by the adoption of SAGE into Microsoft's Windows 7 development: SAGE found one-third of all bugs during the development of Windows 7 [47].

1.3 Overview and contribution

We propose new combinations of static and dynamic analysis for bug detection and program understanding. More specifically, this dissertation makes the following contributions:

1. To reduce false or low-value warnings for static tools, we introduce the concept of residual investigation for program analysis. Residual investigation provides a general recipe for designing predictive and precise dynamic analyses. A residual investigation is a dynamic check installed as a result of running a static analysis that reports a possible program error. The purpose is to observe conditions that indicate whether the statically predicted program fault is likely to be realizable and relevant. The key feature of a residual investigation is that it has to be much more precise (i.e., with fewer false warnings) than the static analysis alone, yet significantly more general (i.e., reporting more errors) than the dynamic tests in the program's test suite pertinent to the statically reported error.

2. We present a technique to enhance the dynamic invariant detection approach. Our work identifies and describes the idea of second-order constraints that can help improve the consistency and relevance of dynamically inferred invariants. We call these constraints second-order because they are constraints over constraints: they relate classes of invariants (first-order constraints). For instance, even though the invariants describing the behavior of two functions f1 and f2 may be unknown, we may know

that any valid input for f1 is also valid for f2, i.e., the precondition of f1 implies that of f2.

3. Our SEDGE system addresses the dataflow program test case generation completeness problem: for every dataflow operator, we produce data aiming to cover all cases that arise in the dataflow program (e.g., both passing and failing a filter). SEDGE relies on transforming the program into symbolic constraints, and solving the constraints using a symbolic reasoning engine (a powerful SMT solver), while using input data as concrete aids in the solution process. The approach resembles dynamic-symbolic (a.k.a. concolic) execution in a conventional programming language, adapted to the unique features of the dataflow domain. SEDGE is the first dynamic symbolic execution engine for dataflow programs.

CHAPTER 2

RESIDUAL INVESTIGATION: PREDICTIVE AND PRECISE BUG DETECTION

A tension exists between static analysis and testing. Static analysis has the problem of issuing false positives: namely, it often warns about bugs that are not real bugs. They are just there because of over-generalizing observations. However, testing does not suffer from false positives. When it reports an error, it is a true error. Conversely, testing suffers from false negatives: it misses many errors. It only catches errors that are dynamically exercised by the test suite, whereas static analysis generalizes the test suite, or generalizes what the application would normally do. Static analysis does not even need a test suite. Therefore static analysis has fewer false negatives. That is a traditional tension between static analysis and testing. Ideally, what we would like is to have something in the middle. Dynamic analysis is a concept between testing and static analysis. Although the term is not always well defined, it can stand for techniques that generalize executions relative to mere testing, yet do not perform a fully over-approximate (and, hence, imprecise) static analysis.

This chapter investigates predictive and precise dynamic analysis. The thrust of this dynamic analysis is to have very few false positives and fewer negatives than testing. Predictive and precise dynamic analysis can be combined with static analysis to improve the precision and recall of bug detection. We have confirmed the efficacy of this combination by implementing it inside FindBugs, a static bug detection tool.

2.1 Introduction and Motivation

False error reports are the bane of automatic bug detection—this experience is perhaps the most often-reported in the program analysis research literature [3, 79, 90, 110, 115].

Programmers are quickly frustrated and much less likely to trust an automatic tool if they observe that reported errors are often not real errors, or are largely irrelevant in the given context. This is in contrast to error detection at early stages of program development, where guarantees of detecting all errors of a certain class (e.g., type soundness guarantees) are desirable. Programmers typically welcome conservative sanity checking while the code is actively being developed, but prefer later warnings (which have a high cost of investigation) to be issued only when there is high confidence that the error is real, even at the expense of possibly missing errors.

The need to reduce false or low-value warnings raises difficulties especially for static tools, which, by nature, overapproximate program behavior. This has led researchers to devise combinations of static analyses and dynamic observation of faults (e.g., [18, 25, 26, 46, 63, 103, 105]) in order to achieve higher certainty than purely static approaches.

In this chapter we identify and present a new kind of combination of static and dynamic analyses that we term *residual investigation*. A residual investigation is a dynamic analysis that serves as the run-time agent of a static analysis. Its purpose is to determine with higher certainty whether the error identified by the static analysis is likely true. In other words, one can see the dynamic analysis as the "residual" of the static analysis at a subsequent stage: that of program execution. The distinguishing feature of a residual investigation, compared to past static-dynamic combinations, is that the residual investigation does not intend to report the error only if it actually occurs, but to identify *general* conditions that reinforce the statically detected error. That is, a residual investigation is a *predictive* dynamic analysis, predicting errors in executions not actually observed. The predictive nature of residual investigation is a significant advantage in practice: high-covering test inputs are hard to produce for complex programs.

Consider as an example the "equal objects must have equal hashcodes" analysis (codenamed HE) in the FindBugs static error detector for Java [3, 57, 59]. The HE analysis emits a warning whenever a class overrides the method `equals(Object)` (originally defined in the `Object` class, the ancestor of all Java classes) without overriding the `hashCode()` method (or vice versa). The idea of the analysis is that the hash code value of an object should serve as an equality signature, so a class should not give a new meaning to equality without updating the meaning of `hashCode()`. An actual fault may occur if, e.g., two objects with distinct hash code values are equal as far as the `equals` method is concerned, and are used in the same hash table. The programmer may validly object to the error warning, however: objects of this particular class may never be used in hash tables in the current program. Our residual investigation consists of determining whether (during the execution of the usual test suite of the program) objects of the suspect class are ever used inside a hash table data structure, or otherwise have their `hashCode` method ever invoked. (The former is a strong indication of an error, the latter a slightly weaker one.) Note that this will likely not cause a failure of the current test execution: all objects inserted in the hash table may have distinct hash code values, or object identity in the hash table may not matter for the end-to-end program correctness. Yet, the fact that objects of a suspect type are used in a suspicious way is a very strong indication that the program will likely exhibit a fault for different inputs. In this way the residual investigation is a predictive dynamic analysis: it is both more general than mere testing and more precise than static analysis.

We have designed and implemented residual investigations for several of the static analyses/bug patterns in the FindBugs system. The result is a practical static-dynamic analysis prototype tool, RFBI (for *Residual FindBugs Investigator*). Our implementation uses standard techniques for dynamic introspection and code interposition, such as bytecode rewriting and customized AspectJ aspects [65]. The addition of extra analyses is typically hindered only by engineering (i.e., implementation) overheads. Designing the residual investigation to complement a specific static pattern requires some thought, but it is typically quite feasible, by following the residual investigation guidelines outlined earlier: the analysis should be significantly more general than mere testing while also offering a strong indication that the statically predicted fault may indeed occur.

We believe that the ability to easily define such analyses is testament to the value of the concept of residual investigation. Predictive dynamic analyses are usually hard to invent. To our knowledge, there are only a small number of predictive dynamic analyses that have appeared in the research literature. (A standard example of a predictive dynamic analysis is the Eraser race detection algorithm [91]: its analysis predicts races based on inconsistent locking, even when no races have appeared in the observed execution.) In contrast, we defined several predictive analyses in a brief period of time, by merely examining the FindBugs list of bug patterns under the lens of residual investigation.

In summary, the main contributions of this work are:

- We introduce residual investigation as a general concept and illustrate its principles.
- We implement residual investigations for several of the most common analyses in the FindBugs system, such as "cloneable not implemented correctly", "dropped exception", "read return should be checked", and several more. This yields a concrete result of our work, in the form of the Residual FindBugs Investigator (RFBI) tool.
- We validate our expectation that the resulting analyses are useful by applying them to 9 open-source applications (including large systems, such as JBoss, Tomcat, NetBeans, and more) using their native test suites. We find that residual investigation produces numerous (31) warnings that do not correspond to test suite failures and are overwhelmingly bugs.
- We discuss the applicability of residual investigation to several other static analyses in the literature. These include race detection, SQL injection analyses, and other pattern-based code analyses in FindBugs. We also discuss how the quality of the test suite affects the applicability of residual investigation.

2.2 Residual Investigation

Residual investigation is a simple concept—it is a vehicle that facilitates communication rather than a technical construction with a strict definition. We next discuss its features and present the example analyses we have defined.

2.2.1 Background and General Concept

We consider a dynamic check that is tied to a static analysis to be a residual investigation if it satisfies the informal conditions outlined in Section 2.1:

- The check has to identify with very high confidence that the statically predicted behavior (typically a fault[1]) is valid and relevant for actual program executions. A residual investigation should substantially reduce the number of false (or low-value) error reports of the static analysis.

- The analysis has to be predictive: it should be generalizing significantly over the observed execution. A residual investigation should recognize highly suspicious behaviors, not just executions with faults. This is a key requirement, since it is easier to have a test case to expose suspicious behaviors than to have a test case to actually cause faults.

A bit more systematically, we can define the following predicates over a program p:

- $B_p(b)$, for "p has bug b", i.e., the program text contains an error, b, of a kind we are concerned with (e.g., class overrides `equals` but not `hashcode`) and there is some execution e_p of p for which this error leads to a fault.

[1] The computing literature is remarkably inconsistent in the use of the terms "error", "fault", "failure", etc. In plain English "error" and "fault" are dictionary synonyms. Mainstream Software Engineering books offer contradicting definitions (some treat an "error" as the cause and a "fault" as the observed symptom, most do the opposite). Standard systems parlance refers indiscriminately to "bus errors" and "segmentation faults", both of which are quite similar program failures. In this chapter we try to consistently treat "error" (as well as "bug" and "defect") as the cause (in the program text) of unexpected state deviation, and "fault" as the dynamic occurrence that exhibits the consequences of an error. That is we think of *programming* errors, and *execution* faults. It should also be possible for the reader to treat the two terms as synonyms.

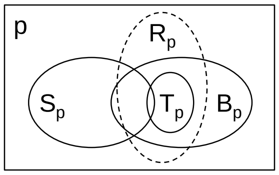

Figure 2.1: The goal of residual investigation (R_p) of a program p is to provide a filter for the static bug warnings (S_p), such that R_p and S_p combined (i.e., the intersection of R_p and S_p) better approximates the program's set of true bugs (B_p) than static analysis alone.

- $S_p(b)$, for "p induces a static error report on bug b", i.e., the program text contains a possible error that the static analysis reports.
- $T_p(b)$, for "p causes a test case fault due to bug b", when executed with p's test suite.
- $R_p(b)$, for "p triggers the residual investigation (dynamic) check" (associated with the static report for bug b), when executed with p's test suite.

Although we typically use the term "residual investigation" for the dynamic analysis, the error reporting process includes the static analysis. That is, a residual investigation issues a reinforced warning in case the static analysis predicted an error and the dynamic analysis confirms it, i.e., in case $S_p(b) \wedge R_p(b)$.

We assume that the dynamic testing is sound for the execution it examines.[2] Thus, we have:

$$\forall p, b : T_p(b) \Rightarrow B_p(b)$$

The requirements for having a valid and useful residual investigation then become:

1. The static analysis is unsound (i.e., some warnings are false):

$$\exists p, b : S_p(b) \wedge \neg B_p(b)$$

[2] We view all analyses as bug detectors, not as correctness provers. Therefore soundness means that warning about an error implies it is a true error, and completeness means that having an error implies it will be reported. For a correctness prover the two notions would be exactly inverse.

(As discussed in the introduction, this assumption is true for most static analysis systems, as they produce false warnings.)

2. The dynamic testing (of a program's test suite) is incomplete (i.e., bugs are missed by testing):

$$\exists p, b : B_p(b) \wedge \neg T_p(b)$$

(Again, the undecidability of non-trivial program properties combined with the soundness of testing implies testing is incomplete.)

3. The residual investigation should be an appropriate bridge for the gap between the static analysis and the bug (see also Figure 2.1):

$$\forall p, b : B_p(b) \text{ approximately iff } S_p(b) \wedge R_p(b)$$

This is the only informal notion in the above. It is practically impossible to have exact equivalence for realistic programs and error conditions, since $R_p(b)$ examines a finite number of program executions. Note that (approximate) equivalence means both that $S_p(b) \wedge R_p(b)$ (likely) implies a bug b exists and that, if there is a bug b, $S_p(b) \wedge R_p(b)$ will (likely) be true. In practice, we place a much greater weight on the former direction of the implication. That is, we are happy to give up on completeness (which is largely unattainable anyway) to achieve (near-)soundness of error warnings.

The question then becomes: how does one identify a good residual investigation? We have used some standard steps:

- Start with the static analysis and identify under what conditions it is *inaccurate* (produces false positives) or *irrelevant* (produces true but low-value positives).

- Estimate how likely these conditions can be. In other words, is this static analysis likely to yield error reports that the programmer will object to, seeing them as false or of low-value?

- If so, is there a concise set of dynamic information (other than a simple fault) that can invalidate the programmer's objections? That is, can we determine based on observable dynamic data if the likely concerns of a programmer to the static warnings do not apply?

Recognizing such "likely objections of the programmer" has been the key part in our design. With this approach we proceeded to identify residual investigations for seven static analyses in the FindBugs system, including some of the analyses that issue the most common FindBugs warnings.

2.2.2 Catalog of Analyses

We next present the residual investigations defined in our RFBI (Residual FindBugs Investigator) tool, each tuned to a static analysis in the FindBugs system. We list each analysis (uniquely described by the corresponding FindBugs identifier) together with the likely user *objections* we identified and a description of *clues* that dynamic analysis can give us to counter such objections. To simplify the presentation, we detail our implementation at the same time.

2.2.2.1 Bad Covariant Definition of Equals (Eq)

The `equals(Object)` method is defined in the `Object` class (`java.lang.Object`) and can be overridden by any Java class to supply a user-defined version of object value equality. A common mistake is that programmers write `equals` methods that accept a parameter of type other than `Object`. The typical case is that of a covariant re-definition of `equals`, where the parameter is a subtype of `Object`, as in the example class `Pixel`:

```
class Pixel {
  int x;
  int y;
  int intensity;

  boolean equals(Pixel p2)
  { return x==p2.x && y==p2.y; }
}
```

This `equals` method does not override the `equals` method in `Object` but instead overloads it for arguments of the appropriate, more specific, static type. As a result, unexpected behavior will occur at runtime, especially when an object of the class type is entered in a Collections-based data structure (e.g., `Set`, `List`). For example, if one of the instances of `Pixel` is put into an instance of a class implementing interface `Container`, then when the `equals` method is needed, `Object.equals()` will get invoked at runtime, not the version defined in `Pixel`. One of the common instances of this scenario involves invoking the `Container.contains(Object)` method. A common skeleton for `Container.contains(Object)` is:

```
boolean contains(Object newObj) {
  for (Object obj : this) {
    if (obj.equals(newObj))
      return true;
  }
  return false;
}
```

Here, `contains(Object)` will use `Object.equals`, which does not perform an appropriate comparison: it compares references, not values. Therefore, objects of type `Pixel` are not compared in the way that was likely intended.

Possible programmer objections to static warnings. FindBugs issues an error report for each occurrence of a covariant definition of `equals`. Although the covariant definition of `equals` is very likely an error, it is also possible that no error will ever arise in the program. This may be an accidental artifact of the program structure, or even a result of the programmer's calculation that for objects of the suspect class the dynamic type will always be equal to the static type, for every invocation of `equals`. For instance, the redefined `equals(Pixel)` method may be used only inside class `Pixel`, with arguments that are always instances of subtypes of `Pixel`, and the programmer may have chosen the covariant definition because it is more appropriate and convenient (e.g., obviates the need for casts).

Dynamic clues that reinforce static warnings. Our residual investigation consists of simply checking whether the ancestral `equals` method, `Object.equals(Object)`, is called on an instance of a class that has a covariant definition of `equals`. The implementation first enters suspect classes into a blacklist and then instruments all call sites of `Object.equals(Object)` to check whether the dynamic type of the receiver object is in the blacklist.

Implementation. We transform the application bytecode, using the ASM Java bytecode engineering library. Generally, for all our analyses, we instrument incrementally (i.e., when classes are loaded), except in applications that perform their own bytecode rewriting which may conflict with load-time instrumentation. In the latter case, we pre-instrument the entire code base in advance (build time).

2.2.2.2 Cloneable Not Implemented Correctly (CN)

Java is a language without direct memory access, hence generic object copying is done only via the convention of supplying a `clone()` method and implementing the `Cloneable` interface. Additionally, the `clone()` method has to return an object of the right dynamic type: the dynamic type of the returned object should be the same as the dynamic type of the receiver of the `clone` method and *not* the (super)class in which the executed method `clone` happened to be defined. This is supported via a user convention: any definition of `clone()` in a class `S` has to call `super.clone()` (i.e., the corresponding method in `S`'s superclass). The end result is that the (special) `clone()` method in the `java.lang.Object` class is called, which always produces an object of the right dynamic type.

Possible programmer objections to static warnings. FindBugs statically detects violations of the above convention and reports an error whenever a class implements the `Cloneable` interface, but does not directly invoke `super.clone()` in its `clone` method (typically because it merely creates a new object by calling a constructor). Although this condition may at first appear to be quite accurate, in practice it often results in false error reports because the static analysis is not inter-procedural. The `clone` method may actually call `super.clone()` by means of invoking a different intermediate method that calls `super.clone()` and returns the resulting object.

Dynamic clues that reinforce static warnings. A dynamic check that determines whether a `clone` method definition is correct consists of calling `clone` on a subclass of the suspect class `S` and checking the return type (e.g., by casting and possibly receiving a `ClassCastException`). Our residual investigation introduces a fresh subclass `C` of `S` defined and used (in a minimal test case) via the general pattern:

```
class C extends S {
  public Object clone()
  { return (C) super.clone(); }
}
... ((new C()).clone()) // Exception
```

If `S` does not have a no-argument constructor, we statically replicate in `C` all constructors with arguments and dynamically propagate the actual values of arguments used for construction of `S` objects, as observed at runtime.

If the test case results in a `ClassCastException` then the definition of `clone` in class `S` is indeed violating convention. Conversely, if `S` implements the `clone` convention correctly (i.e., indirectly calls `super.clone()`) no exception is thrown. This test code is executed the first time an object of class `S` is instantiated. In this way, if class `S` does not get used at all in the current test suite, no error is reported.

The above residual investigation provides a very strong indication of a problem that will appear in an actual execution of the program, without needing to observe the problem itself. Indeed, the current version of the program may not even have any subclasses of `S`, but a serious error is lurking for future extensions.

Implementation. Our implementation of this analysis uses AspectJ to introduce the extra class and code. In the case of complex constructors, we retrieve those with Java reflection and use AspectJ's constructor joinpoints instead of generating customized calls. A subtle point is that if the superclass, `S`, is declared `final` or only has private constructors, the residual investigation does not apply. This is appropriate, since the absence of any externally visible constructor suggests this class is not to be subclassed. Similarly, the generated code needs to be in the same package as the original class `S`, in order to be able to access package-protected constructors.

2.2.2.3 Dropped Exception (DE)

Java has *checked exceptions*: any exception that may be thrown by a method needs to either be caught or declared to be thrown in the method's signature, so that the same obligation is transferred to method callers. To circumvent this static check, programmers may catch an exception and "drop it on the floor", i.e., leave empty the `catch` part in a `try-catch` block. FindBugs statically detects dropped exceptions and reports them.

Possible programmer objections to static warnings. Detecting all dropped exceptions may be a good practice, but is also likely to frustrate the programmer or to be considered a low-priority error report. After all, the type system has already performed a check for exceptions and the programmer has explicitly disabled that check by dropping the exception. The programmer may be legitimately certain that the exception will never be thrown in the given setting (a common case—especially for I/O classes—is that of a general method that may indeed throw an exception being overridden by an implementation that never does).

Dynamic clues that reinforce static warnings. Our residual investigation consists of examining which methods end up being dynamically invoked in the suspect code block and watching whether the same methods ever throw the dropped exception when called from *anywhere* in the program. For instance, the following code snippet shows a method `meth1` whose `catch` block is empty. In the `try` block of `meth1`, first `foo1` is executed, then `foo2` (possibly called from `foo1`), then `foo3`, and so on:

```
void meth1() {
  try {
    foo1();
    //Call-graph foo1()->foo2()->...->fooN()
  } catch(XException e) { }  //empty
}
```

The residual investigation will report an error if there is any other method, `methX`, calling some `foo`*i* where `foo`*i* throws an `XException` during that invocation (regardless of whether that exception is handled or not):

```
void methX {
  try { ...
    //Call-graph ...->fooN()->...
  } catch(XException e) {
    ...  // handled
  }
}
```

In this case the user should be made aware of the possible threat. If `foo`*i* can indeed throw an exception, it is likely to throw it in any calling context. By locating the offending instance, we prove to programmers that the exception can occur. Although the warning may still be invalid, this is a much less likely case than in the purely static analysis.

Implementation. The implementation of this residual investigation uses both the ASM library for bytecode transformation and AspectJ, for ease of manipulation.

We execute the program's test suite twice. During the first pass, we instrument the beginning and end of each empty `try-catch` block with ASM, then apply an AspectJ aspect to find all methods executed in the dynamic scope of the `try-catch` block (i.e., transitively called in the block) that may throw the exception being caught.[3] (We also

[3] To apply the combination of ASM and AspectJ at load time, we had to make two one-line changes to the source code of AspectJ. The first allows aspects to apply to ASM-transformed code, while the second allows AspectJ-instrumented code to be re-transformed.

check that there is no intermediate method that handles the exception, by analyzing the signatures of parent methods on the call stack.) In the first pass we collect all such methods and generate custom AspectJ aspects for the second pass. During the second pass, we then track the execution of all methods we identified in the first pass and identify thrown exceptions of the right type. For any such exception we issue an RFBI error report.

2.2.2.4 Equals Method Overrides Equals in Superclass and May Not Be Symmetric *(EQ_OVERRIDING_EQUALS_NOT_SYMMETRIC)*

Part of the conventions of comparing for value equality (via the `equals` method) in Java is that the method has to be symmetric: the truth value of `o1.equals(o2)` has to be the same as that of `o2.equals(o1)` for every `o1` and `o2`. FindBugs has a bug pattern "equals method overrides equals in super class and may not be symmetric", which emits a warning if both the overriding equals method in the subclass and the overridden equals method in the superclass use `instanceof` in the determination of whether two objects are equal. The rationale is that it is common for programmers to begin equality checks with a check of type equality for the argument and the receiver object. If, however, both the overridden and the overriding `equals` methods use this format the result will likely be asymmetric because, in the case of a superclass, `S`, of a class `C`, the `instanceof S` check will be true for a `C` object but the `instanceof C` check will be false for an `S` object.

Possible programmer objections to static warnings. The above static check is a blunt instrument. The programmer may be well aware of the convention and might be using `instanceof` quite legitimately, instead of merely in the naive pattern that the FindBugs analysis assumes. For instance, the code of the JBoss system has some such correct `equals` methods that happen to use `instanceof` and are erroneously flagged by FindBugs.

Dynamic clues that reinforce static warnings. Our residual investigation tries to establish some confidence before it reports the potential error. We checked this pattern dynamically by calling both `equals` methods whenever we observe a comparison involving a contentious object and test if the results match (this double-calling is safe as long as there are no relevant side effects). If the two `equals` methods ever disagree (i.e., one test is true, one is false) we emit an error report.

Implementation. We implemented this residual investigation using AspectJ to intercept calls to the `equals` method and perform the dual check in addition to the original.

2.2.2.5 Equal Objects Must Have Equal Hashcodes (HE)

As mentioned in the Introduction, FindBugs reports an error when a class overrides the `equals(Object)` method but not the `hashCode()` method, or vice versa. All Java objects support these two methods, since they are defined at the root of the Java class hierarchy, class `java.lang.Object`. Overriding only one of these methods violates the standard library conventions: an object's hash code should serve as an identity signature, hence it needs to be consistent with the notion of object value-equality.

Possible programmer objections to static warnings. This warning can easily be low-priority or irrelevant in a given context. Developers may think that objects of the suspect type are never stored in hashed data structures or otherwise have their hash code used for equality comparisons in the course of application execution. Furthermore, the warning may be cryptic for programmers who may not see how exactly this invariant affects their program or what the real problem is.

Dynamic clues that reinforce static warnings. Our Residual FindBugs Investigator installs dynamic checks for the following cases:

- `Object.hashCode()` is called on an object of a class that redefines `equals(Object)` and inherits the implementation of `hashCode()`.

- `Object.equals(Object)` is called on a class that redefines `hashCode()` and inherits the implementation of `equals(Object)`.

Meeting either of these conditions is a strong indication that the inconsistent overriding is likely to matter in actual program executions. Of course, the error may not trigger a fault in the current (or any other) execution.

Implementation. Our detector is implemented using the ASM Java bytecode engineering library. First, we create a blacklist containing the classes that only redefine one of `Object.equals(Object)` and `Object.hashCode()` in a coordinated manner. Then we introduce our own implementations of the missing methods in the blacklisted classes. The result is to intercept every call to either `Object.equals(Object)` or `Object.hashCode()` in instances of blacklisted classes.

2.2.2.6 Non-Short-Circuit Boolean Operator (NS)

Programmers may mistakenly use the non-short-circuiting binary operators `&` or `|` where they intend to use the short-circuiting boolean operators `&&` or `||`. This could introduce bugs if the first argument suffices to determine the value of the expression and the second argument contains side-effects (e.g., may throw exceptions for situations like a null-pointer dereference or division by zero). Therefore, FindBugs issues warnings for uses of `&` and `|` inside the conditions of an `if` statement.

Possible programmer objections to static warnings. Such warnings can clearly be invalid or irrelevant, e.g. if the programmer used the operators intentionally or if they do not affect program behavior. FindBugs can sometimes identify the latter case through static analysis, but such analysis must be conservative (e.g., FindBugs considers any method call on the right hand side of an `&&` or `||` to be side-effecting). Therefore the error reports are often false.

Dynamic clues that reinforce static warnings. Using a residual investigation we can check for actual side-effects on the right-hand side of a non-short-circuiting boolean operator. It is expensive to perform a full dynamic check for side-effects, therefore we check instead for several common cases. These include dynamically thrown exceptions (directly or in transitively called methods, as long as they propagate to the current method), writes to any field of the current class, writes to local variables of the current method, and calls to well-known (library) I/O methods. Since the residual investigation can miss some side-effects, it can also miss actual bugs. Additionally, the residual investigation will often fail to generalize: there are common patterns for which it will report an error only if the error actually occurs in the current execution. For instance, in the following example an exception is thrown only when the left-hand side of the boolean expression should have short-circuited:

```
  if (ref == null | ref.isEmpty()) ...
```

Still, the residual investigation generally avoids the too-conservative approach of FindBugs, while reporting dynamic behavior that would normally go unnoticed by plain testing.

Implementation. The implementation of this residual investigation is one of the most complex (and costly) in the Residual FindBugs Investigator arsenal. We rewrite boolean conditions with the ASM bytecode rewriting framework to mark a region of code (the right-hand side of the operator) for an AspectJ aspect to apply, using a "conditional check pointcut". The aspect then identifies side-effects that occur in this code region, by instrumenting field writes, installing an exception handler, and detecting method calls to I/O methods over files, network streams, the GUI, etc. Additionally, we use ASM to detect local variable writes (in the current method only) in the right-hand side of a boolean condition.

2.2.2.7 Read Return Should Be Checked (RR)

The `java.io.InputStream` class in the Java standard library provides two `read` methods that return the number of bytes actually read from the stream object (or an end-of-file condition). It is common for programmers to ignore this return value. FindBugs reports an error in this case. At first glance this check looks to be foolproof: the code should always check the stream status and received number of bytes against the requested number of bytes. If the return value from `read` is ignored, we may read uninitialized/stale elements of the result array or end up at an unexpected position in the input stream.

Possible programmer objections to static warnings. Perhaps surprisingly, this FindBugs check is the source of many false positives. Although the original `java.io.InputStream` class can indeed read fewer bytes than requested, the class is not `final` and can be extended. Its subclasses have to maintain the same method signature (i.e., return a number) when overriding either of the two `read` methods, yet may guarantee to always return as many bytes as requested (Notably, the Eclipse system defines such a subclass and suffers from several spurious FindBugs error reports.)

Dynamic clues that reinforce static warnings. Our residual investigation first examines whether the read method is called on a subclass of `java.io.InputStream` that overrides the `read` method. If so, we wait until we see a `read` method on an object of the suspect subclass return fewer bytes than requested (even for a call that *does* check the return value). Only then we report *all* use sites that do not check the return value of `read`, as long as they are reached in the current execution and the receiver object of `read` has the suspect dynamic type.

Implementation. The implementation of this analysis involves two computations, performed in the same execution. For the first computation, we instrument all `read` methods that override the one in `InputStream` (using AspectJ) to observe which ones return fewer bytes than requested. We collapse this information by dynamic object type, resulting in a list of all types implementing a `read` method that may return fewer bytes than requested; we always include `java.io.InputStream` in that list. For the second computation, we instrument all suspected `read` call sites with ASM, to determine the dynamic type of the receiver object. These dynamic types are the output of the second computation. At the end of the test suite execution, we cross-check the output of both passes. We then report any use of `read` without a result check on an object with a dynamic type for which we know that `read` may return fewer bytes than the maximum requested.

2.2.3 Discussion

The main purpose of a residual investigation is to provide increased soundness for bug reports: an error report should be likely valid and important. In this sense, a residual investigation is not in competition with its underlying static analysis, but instead complements it. In the case of RFBI, the point is not to obscure the output of FindBugs: since the static analysis is performed anyway, its results are available to the user for inspection. Instead, RFBI serves as a classifier and reinforcer of FindBugs reports: the RFBI error reports are classified as higher-certainty than a plain FindBugs report.

This confidence filtering applies both positively and negatively. RFBI may confirm some FindBugs error reports, fail to confirm many because of a lack of pertinent dynamic observations, but also fail to confirm some *despite* numerous pertinent dynamic observations. To see this, consider an analysis such as “dropped exception” (DE). RFBI will issue no error report if it never observes an exception thrown by a method dynamically called from a suspicious `try-catch` block. Nevertheless, it could be the case that the program’s test suite never results in exceptions or that there are exceptions yet the suspicious `try-catch` block was never exercised, and hence the methods under its dynamic scope are unknown. In this case, RFBI has failed to confirm an error report due to lack of observations and not due to the observations not supporting the error. This difference is important for the interpretation of results. It is an interesting future work question how to report effectively to the user the two different kinds of negative outcomes (i.e., unexercised code vs. exercised code yet failure to confirm the static warning). Our experimental evaluation describes this distinction via a metric of the number of dynamic opportunities to confirm an error report that the residual investigation had, as we discuss in the next section.

2.2.4 Implementation Complexity

Residual investigation requires effort in capturing dynamic evidence related to static bug reports, by using instrumentation infrastructure such as ASM and AspectJ. The degree of effort varies with the complexity of runtime evidence that we are trying to collect. In simple cases, residual investigations are easy to develop. For example, for the bug pattern "Equal Objects Must Have Equal Hashcodes", we only have to insert a method for each class in a blacklist as the class is being loaded. But in complex cases that require interaction of different instrumentation tools or a variety of dynamic information, residual investigation needs non-trivial development effort. A typical example is the implementation of "Non-Short-Circuit Boolean Operator", which needs to make AspectJ communicate with ASM on the code blocks for which field writes, exception handlers, network or I/O operations, etc. need to be instrumented.

In total, our tool contains 3292 lines of Java code, including 298 lines for utility classes, such as logging facilities and Java agents, and 2994 lines for 7 bug patterns altogether. Note that this does not include AspectJ code generated dynamically. Figure 2.2 quantifies the implementation complexity via source lines of code[4] for each bug pattern. The figure also lists the set of tools used to develop the residual investigation.

Bug Pattern	**Implementation Tool**	**Lines of Code**
Bad Covariant Definition of Equals	ASM	40
Cloneable Not Implemented Correctly	AspectJ	512
Dropped Exception	ASM, AspectJ	769
Equals Method May Not Be Symmetric	AspectJ	465
Equal Objects Must Have Eq. Hashcode	ASM	40
Non-Short-Circuit Boolean Operator	ASM, AspectJ	811
Read Return Should Be Checked	ASM, AspectJ	357

Figure 2.2: Size of implementation and implementation tools in residual investigations.

2.3 Evaluation

We evaluate RFBI in two complementary ways. First, we do a targeted evaluation using two medium-size systems with which we were already familiar, and, hence, can directly

[4] The counting is done using SLOCCount—`http://www.dwheeler.com/sloccount/`

assess the quality of bug reports. Second, we apply RFBI on several well-known large open-source systems. For such systems, judging the quality of a bug report is harder and can only be done via sampling and best-effort estimates.

In order to make the important distinction between "relevant code not exercised" (undetermined bugs) and "relevant code exercised, yet no confirmation of the bug found" (rejected bugs), we use a *dynamic potential* metric. Generally, for each static error report, we automatically pick a method whose execution is part of our analysis (chosen according to the kind of bug reported) and measure how many times this method gets executed by the test suite. The resultant count gives us an upper bound on the number of reports we can expect from residual investigation. We found this metric to be invaluable. Generally, when multiple conditions need to be satisfied for a residual investigation, we choose one of them arbitrarily. For instance, for the "equal objects must have equal hashcodes" analysis, we measure the number of times the overridden `equals` is called on objects of the suspect class that redefines `equals` and not `hashCode` (and vice versa). This is not directly tied to the opportunities to find the bug (which, as we described earlier, is reported if `hashCode` is ever called on such a suspect object) but it is a good indication of how much this part of the code is exercised.

2.3.1 Targeted Evaluation

The targeted evaluation consists of applying RFBI to programs whose behavior we know well, so that we can determine whether RFBI's warnings are correct or incorrect by examining the systems' source code. We evaluated RFBI on two medium-sized systems, Daikon [36] (Revision a1364b3888e0) and Apache Jackrabbit (Revision 1435197), using the systems' standard test suites. Since the size of Apache Jackrabbit is still large for our purpose of targeted evaluation, we used only the core component of Apache Jackrabbit (which has 21 components overall).

2.3.1.1 Analysis of reports

FindBugs reports 14 bugs for Daikon and Jackrabbit. The standard system test suites exercise 8 of these bugs, i.e., there are 8 reports for which the DP metric is non-zero. This ratio between RFBI and FindBugs reports is high, compared to our later experiments with other large third-party systems.

Of the 8 RFBI reports, 3 are reinforced and 5 are rejected. By manual inspection, we found that RFBI correctly confirms 2 of the 3 reports, incorrectly reinforces a false positive, and correctly rejects 5 reports. (This yields a precision of 66.6% and a recall of 100% over the 8 exercised FindBugs reports, but clearly the absolute numbers of reports are low for statistical accuracy.)

- Findbugs reports two "Cloneable Not Implemented Correctly" warnings for Jackrabbit. RFBI observes one dynamically exercised instance and correctly confirms the instance: rather than calling `super.clone()` in its superclass, `clone` in the instance directly constructs a new object.

- Findbugs reports four "Dropped Exception" warnings for Daikon and two for Jackrabbit. RFBI does not observe any dynamically exercised instance for Daikon, but does observe two for Jackrabbit. Of the two dynamically exercised instances, RFBI incorrectly confirms one instance, and correctly rejects another one. In the incorrectly confirmed instance, exception throwing is used as a means for handling control flow within a large method body. The method in question tries several alternatives for an action, in a fixed order. It encodes each alternative as a single block of code wrapped into its own (empty) exception handler. If an alternative succeeds, the block returns from the method to signal overall success. If the alternative fails, the block raises an exception and thereby continues to the next block. Thus, the dropped exception here needs no exception handler. The correctly rejected bug report ignores a statically reported exception that supposedly can be raised while shutting down a virtual file system. In practice, this exception can only arise if initialization failed.

However, if the virtual file system failed to initialize correctly, some other exceptions would already have been thrown at creation time, and the related catch blocks would have handled or reported these exceptions. Thus, it is not necessary to handle or report the same exception again when the virtual file system is closed.

- FindBugs reports two "Equal Objects Must Have Equal Hashcodes" warnings for Daikon and three for Jackrabbit. All reported classes override their superclasses' `equals` methods without overriding `hashCode`. Instances of all reports are exercised in test suites. RFBI correctly confirms one report on Daikon and rejects another. To understand the confirmed Daikon case, note that Daikon takes an arbitrary client program, instruments it, and performs dynamic analysis. In the correctly confirmed error report, Daikon takes arbitrary objects from the client program and uses them as keys to a hash map. Since Daikon knows nothing about the client program a priori, this behavior is inherently unsafe. The rejected Daikon case involves a class whose instances are only inserted in an `ArrayList` structure, which does not employ `hashCode`.

  RFBI correctly rejects the three Jackrabbit reports. In two of these cases, instances of the suspicious class are used as values of a hash map, but never as keys (i.e., they are never hashed). Instead, a uniquely-identifying field of the suspicious class is used as the key of the `HashMap` structure. The field type is not involved in the FindBugs error report, since it correctly defines both `hashCode` and `equals`. The third Jackrabbit report, similar to the rejected Daikon case, involves a class whose instances are only used in an `ArrayList` structure.

- FindBugs reports one "Read Return Should Be Checked" warning for Jackrabbit, but the relevant code is not exercised dynamically.

In summary, although small, our experiments with Daikon and the Apache Jackrabbit core package show the potential for residual investigation.

2.3.1.2 Runtime overhead

Because Daikon's own instrumenter and Jackrabbit's testing framework are in conflict with RFBI's dynamic instrumenter, we pre-instrumented the code bases at build time. When we run pre-instrumented code, the overhead is mostly due to logging dynamic clues that reinforce static warnings. We do not measure the overhead to compute the DP values because we do not report them by default. The analysis time was measured for one run each on a 4-core 3.1 GHz Intel i5 with 8 GB of RAM. Running the uninstrumented code of Daikon takes 41m:19s, whereas running the pre-instrumented code of Daikon takes 43m:38s. Running the uninstrumented code of Jackrabbit takes 7m:32s, whereas running the pre-instrumented code of Jackrabbit takes 8m:30s. Thus, the runtime slowdown of running pre-instrumented code for these residual investigations is small. The overhead will, however, vary based on the kinds of dynamic analyses installed, as we will also see in the next section.

2.3.2 Evaluation on Large Systems

We evaluated RFBI on several large open-source systems: JBoss (v.6.0.0.Final), BCEL (v.5.2), NetBeans (v.6.9), Tomcat (7.0), JRuby (v.1.5.6), Apache Commons Collections (v.3.2.1), and Groovy (v.1.7.10). The advantage of using third-party systems for evaluation is that we get a representative view of what to expect quantitatively by the use of residual investigation. The disadvantage is that these systems are large, so great effort needs to be expended to confirm or disprove bugs by manual inspection.

It is common in practice to fail to confirm a static error report because of a lack of relevant dynamic observations. This is hardly a surprise since our dynamic observations are dependent on the examined program's test suite. For all systems, we used the test suite supplied by the system's creators, which in some cases was sparse (as will also be seen in our test running times).

Bug Pattern	FindBugs	RFBI	Dynamic Potential	Test Cases
Bad Covariant Definition of Equals	5	0	0	0
Cloneable Not Implemented Correctly	41	4	5	0
Dropped Exception	128	0	7	0
Equals Method May Not Be Symmetric	8	1	1	0
Equal Objects Must Have Eq. Hashcode	211	25	28	0
Non-Short-Circuit Boolean Operator	18	0	1	0
Read Return Should Be Checked	25	1	1	0
Total	436	31	43	0

Figure 2.3: Summary of results: reports by FindBugs vs. RFBI, the Dynamic Potential metric of how many of the static error reports had related methods that were exercised dynamically, and the number of original test cases that reported an error.

2.3.2.1 Volume of reports

Figure 2.3 shows the number of static error reports (FindBugs), reports produced by residual investigation (RFBI), and dynamic potential metric. The difference between FindBugs and RFBI reports is roughly an order of magnitude: a total of 436 potential bugs are reported by FindBugs for our test subjects and, of these, RFBI produces reports for 31. Thus, it is certainly the case that residual investigation significantly narrows down the area of focus compared to static analysis. Similarly, none of the test cases in our subjects' test suites failed. Therefore, the 31 reports by RFBI do generalize observations significantly compared to mere testing. Of course, these numbers alone say nothing about the *quality* of the reports—we examine this topic later.

By examining the dynamic potential metric in Figure 2.3, we see that much of the difference between the numbers of FindBugs and RFBI reports is due simply to the suspicious conditions not being exercised by the test suite. Most of the static bug reports are on types or methods that do not register in the dynamic metrics. This can be viewed as an indication of "arbitrariness": the dynamic analysis can only cover a small part of the static warnings, because of the shortcomings of the test suite. A different view, however, is to interpret this number as an indication of why static analysis suffers from the "slew of false positives" perception mentioned in Section 2.1. Programmers are likely to consider static warnings to be irrelevant if the warnings do not concern code that is even touched by the program's test suite.

2.3.2.2 Quality of residual investigation reports (summary)

RFBI narrows the programmer's focus compared to FindBugs but the question is whether the quality of the RFBI reports is higher than that of FindBugs reports, and whether RFBI succeeds as a classifier (i.e., whether it classifies well the dynamically exercised reports into bugs and non-bugs).

Since our test subjects are large, third-party systems, we cannot manually inspect all (436) FindBugs reports and see which of them are true bugs. Instead, we inspected the 43 reports that are dynamically exercised (per the DP metric) as well as a sample of 10 other FindBugs reports that were never dynamically exercised (and, thus, never classified by RFBI as either bugs or non-bugs). The latter were chosen completely at random (blind, uniform random choice among the reports).

If we view RFBI as a classifier of the 43 dynamically exercised FindBugs reports, its classification quality is high. As seen in Figure 2.3, RFBI classifies 31 of the 43 dynamically exercised reports as bugs (i.e., reinforces them), thus rejecting 12 reports. Figure 2.4 shows which of these RFBI classifications correspond to true bugs vs. non-bugs. The number for the correct outcome for each row is shown in boldface.

Dynamic reports	bug	non-bug	undetermined
31 reinforced	**24**	6	1
12 rejected	0	**11**	1
43 total	24	17	2

Figure 2.4: Quality of RFBI warnings on the 43 dynamically exercised FindBugs reports.

From Figure 2.4, we have that the precision of RFBI is $\geq 77\%$ (or that RFBI produces $< 23\%$ false warnings) and that its recall is $\geq 96\%$, over the 43 dynamically exercised FindBugs reports.

Bug Pattern	#confirmed bugs/dynamic potential(#executed methods)/avg. times executed						
	JBoss	BCEL	Net Beans	Tomcat	JRuby	Apache CC	Groovy
Bad Covariant Def. of Equals			0/0/0	0/0/0			
Cloneable Not Impl. Correctly			0/0/0	0/1/9		3/3/658	1/1/11
Dropped Exception	0/4/378		0/1/79	0/0/0	0/1/5		0/1/25
Equals Method Not Symmetric	0/0/0		0/0/0		1/1/2.6M		
Equal Objs ⇒ Eq. Hashcodes	1/2/1	20/20/77k	0/0/0	1/1/5k	2/2/3.5		1/3/14
Non-Short-Circuit Bool. Oper.			0/0/0		0/1/194		
Read Ret. Should Be Checked			0/0/0	0/0/0	0/0/0		1/1/571

Figure 2.5: Breakdown of all RFBI warnings as well as the dynamic potential metric for the warning. "*a*/*b*/*c*" means there were a RFBI warnings of this kind, b dynamic potential methods executed (zero typically means there was no opportunity for the residual investigation to observe an error of the statically predicted kind), and each of them was observed to execute c times (on average). Thus, the sum of all as is the number of RFBI reinforced reports (31) and the sum of all bs is the number of total dynamically exercised FindBugs reports (43). Empty cells mean that there were no static error reports for this test subject and analysis—this is in contrast to 0/0/0 cells, which correspond to static warnings that were never exercised dynamically.

For comparison, among the 10 non-exercised FindBugs reports that we sampled at random, only one is a true bug. Thus, the precision of FindBugs on this sample was 10%, which is a false warning rate of 90%. We see, therefore, that RFBI reports are of higher quality than FindBugs reports and the programmer should prioritize their inspection.

2.3.2.3 Detailed discussion of reports

We next discuss in detail the RFBI results that we inspected manually. This yields concrete examples of bug reports reinforced and rejected (both correctly and falsely) for the numbers seen above. Figure 2.5 breaks down the reports dynamically exercised by test subject and analysis, as well as their dynamic potential. Note that in the rest of this section we are not concerned with FindBugs reports that are not exercised dynamically.

- RFBI correctly confirms four dynamically exercised instances of "Cloneable Not Implemented Correctly" and rejects one. This is a sharp distinction, and, we believe, correct. In three of the four instances (in Apache Commons) `clone` directly constructs a new object, rather than calling the parent `clone`. One bug in Groovy arises in an instance where a delegator violates the cloning protocol by returning a clone of its delegate instead of a clone of itself. The rejected bug report is a `clone` method for a singleton object that returned `this`, which is entirely typesafe.

- RFBI rejects seven “Dropped Exception” reports, of which our manual inspection found six to have been rejected correctly and one to be unclear. The unclear case involved the NetBeans test harness ignoring an exception caused by backing store problems during synchronization; we expect that such an exception is likely to trigger further bugs and hence unit test failures but argue that it might have been appropriate to log the exception rather than ignoring it. Of the remaining six cases, four affected JBoss. In three of these cases the code correctly handles the erroneous case by exploiting the exceptional control flow in other ways (e.g., when an assignment throws an exception, the left-hand side retains its previous value, which the code can test for) or by falling back on alternative functionality (for example, JBoss attempts to use I/O access to `/dev/urandom` to generate random numbers, but falls back on the Java random number generator if that approach fails). The fourth JBoss case ignores exceptions that may arise while shutting down network connections. We assume that the programmers’ rationale is that they can do no more but trust that the library code tries as hard as possible to release any resources that it has acquired, and that afterwards the program should run in as robust a fashion as possible.

  In one of the two remaining cases (JRuby), exceptions are dropped in debug code and can only arise if the JRuby VM has been corrupted or has run out of memory. In the final case (Groovy), the dropped exception is a ClassLoaderException that could only arise if the Java standard library were missing or corrupt.

- We observed one out of eight “Equals Method May Not Be Symmetric” instances dynamically, in JRuby’s `RubyString` class. RFBI here indicated that the report was correct, pointing to an implementation of `equals` that differs subtly from the equivalent Ruby equality check for the same class. In practice, the ‘Java’ version of equality is only used in rare circumstances and unlikely to cause problems, unless the integration between Java and Ruby were to be changed significantly. We thus found it unclear whether this bug report was a true positive (as suggested by RFBI) or not.

- RFBI confirms 20 "Equal Objects Must Have Equal Hashcodes" reports for BCEL. The 20 RFBI reports concern classes that represent branch instructions in the Java bytecode language. All reported classes define an application-specific value equality `equals` method, without ever defining `hashCode`. Objects for branch instructions, however, get entered in a `HashSet`, as part of a seemingly oft-used call: `InstructionHandle.addTargeter`. Therefore, we believe that the bug warning is accurate for these instructions and can result in obscure runtime faults.

  RFBI incorrectly confirms two "Equal Objects Must Have Equal Hashcodes" bug reports in JRuby: in both cases, `equals` is overridden but `hashCode` is not. In one of the pertinent classes, the superclass `hashCode` implementation uses a custom virtual method table for Ruby to look up a correct subclass-specific `hashCode` implementation; such complex indirection is impossible to detect in general. In the other class, `hashCode` is only used to generate a mostly-unique identifier for debugging purposes, instead of hashing. This breaks the heuristic assumption that `hashCode` and `equals` collaborate. In Groovy, RFBI incorrectly confirms a bug for exactly the same reason. Meanwhile, the two bugs we rejected in Groovy again did not see invocations of the missing methods, and we consider our results to be accurate in those cases. RBFI also incorrectly confirms a bug for JBoss (and rejects one, correctly): although the `equals` method is overridden, it does nothing more than delegate to the superclass method, which also defines an appropriate `hashCode`.

- RFBI correctly rejects a "Non-Short Circuit Boolean Operator" bug report in JRuby involving a method call, as the method in question is only a getter method (and thus has no side effects that might unexpectedly alter program behavior).

- Only one report of “Read Return Should Be Checked” is exercised in unit tests. This report involves Groovy’s Json lexer, which in one instance does not check the number of bytes returned by a read operation. However, the bytes read are written into an empty character array that is immediately converted to a string, which is then checked against an expected result: if the number of bytes read was less than requested, this later check must fail, because the generated string will be too short. Such complex logic is beyond the scope of RFBI, which erroneously confirms the static report.

In summary, the few misjudged bug reports arose because the code violated the assumptions behind our concrete residual investigation heuristics (e.g., application-specific use of `hashCode`). Incorrect RFBI bug reports typically were due to complex mechanisms that achieve the desired result in a way that requires higher-level understanding yet proves to be semantically correct (e.g., leaving out a test for bytes read because it is subsumed by a string length check). It is unlikely that any automatic technique can eliminate bug reports that are erroneous because of such factors.

2.3.2.4 Runtime overhead

Figure 2.6 shows the runtime overhead of our residual investigation. We measured the analysis time of one run on a 4-core 2.4 GHz Intel i5 with 6 GB of RAM. As can be seen, compared to the baseline (of uninstrumented code) residual investigation slows down the execution of the test suite typically by a factor of 2-to-3, possibly going up to 6. The “dropped exception” analysis is the worst offender due to executing the test suite twice and watching a large number of the executed method calls.

2.3.2.5 Threats to validity

Our experimental evaluation of the efficacy of residual investigation shows that it yields higher-precision bug reporting and a reliable classification of bugs. The main threats to validity include the following threats to external validity.

Bug Pattern	Execution time with and without instrumentation [min:s]						
	JBoss	BCEL	Net Beans	Tomcat	JRuby	ApacheCC	Groovy
Bad Covariant Def. of Equals			13:07	3:04			
Cloneable Not Impl. Correctly			7:17	5:20		5:25	39:15
Dropped Exception	655:01		16:00	16:05	17:08		41:44
Equals Method Not Symmetric	531:42		8:23		11:03		
Equal Objs ⇒ Eq. Hashcodes	363:48	1:20	13:07	3:03	3:44		7:25
Non-Short-Circuit Bool. Oper.			9:36		11:13		
Read Ret. Should Be Checked			16:49	10:19	12:30		42:25
No Instrumentation	178:07	:23	6:42	3:03	3:28	2:05	7:13

Figure 2.6: Running times for all residual investigations. The baseline (bottom line of the table) is the non-instrumented test suite running time. Empty cells mean that there were no static error reports for this test subject and analysis.

- Choice of subject applications: We did not select our seven subject applications truly randomly from the space of all possible Java applications or even from all current Java applications. I.e., our empirical results may not generalize well to other applications. However, our applications cover a variety of application areas: we use a data structure library, two language runtime systems, a bytecode engineering library, a dynamic axiom detector, a content repository, two web servers, and an IDE. Given the large size of these subject applications, we suspect that our findings will generalize to a large extent, but this remains to be confirmed as part of a larger empirical study.

- Choice of FindBugs patterns: We did not select the patterns randomly from the list of all FindBugs patterns. I.e., residual investigation likely does not generalize to all FindBugs patterns. Our choice of patterns was influenced by subjective considerations such as how well-known we deemed a pattern to be. Six of our patterns have been described in an article [58] by the FindBugs authors. That article describes a total of 18 patterns. For our evaluation we picked patterns for which we suspected that FindBugs would produce false warnings on our subject applications. We argue that this is not a strong threat to validity, since we easily obtained strong results for a third of the sample presented by the earlier FindBugs article.

If we step back and review all current FindBugs bug patterns, we can easily identify several of them that are simple enough to allow for a fully precise static detector, and such a fully precise static detector will not benefit from residual investigation. However, many other bug patterns are too complex to allow for a precise static detector. For example, Hovemeyer and Pugh tested twelve out of the 18 patterns they described (including four of ours) for false positives in two applications. They found that ten of the twelve patterns (including ours) produced false positives with the pattern implementations they had available in 2004. We suspect that the 18 patterns that they described are at least somewhat representative of all FindBugs patterns. We selectively mention a few more FindBugs patterns in our discussion of the greater applicability of residual investigation in the next section.

- Choice of static analysis system: We did not select FindBugs, our static analysis system, truly randomly. We picked FindBugs because it is arguably the most widely known and used such tool for Java. We suspect that our findings will generalize to other static analysis tools and approaches. The next section discusses some such directions, but only anecdotally.

2.4 Residual Investigation for Race Detection

Residual investigation is a concept of much greater applicability than just our RFBI tool. The essence is to combine static analyses with predictive dynamic analyses that ascertain the validity or importance of the warned-about error. We show one example of the applicability of residual investigation by designing and implementing a residual analysis for race detection. Applying the residual analysis, we successfully analyze four real-world applications and evaluate the practical benefit experimentally.

2.4.1 Problem and Design

Static race detection is an analysis domain of very high recent interest [1, 34, 68, 81, 88, 100, 109]. At the same time, static race detection is notoriously fraught with a high false positive rate. In the most common static report pattern that causes programmer objections, the "race" is on a variable that is never publicized, i.e., it is only ever accessed by one thread.[5] It is hard to statically establish whether a memory location (object field, class field or array element) is ever accessed by multiple threads, especially since the number of memory locations is not statically bounded. A *thread-escape* analysis is often used but it is by nature both a whole-program analysis and quite conservative and overapproximate. In this way, memory locations are often conservatively considered thread-shared when they are not, thus resulting in false static race reports. Indeed, non-shared locations are more likely to result in static race reports since the code will access them with no synchronization.

This kind of false positive is a perfect fit for applying residual investigation. Our heuristic here is to check whether a memory location is shared: A memory location is shared in a given execution iff two different threads access the location and the thread performing the first access remains active (i.e., not yet joined) at the time of the second access to the location.

In this light, how we confirm a static race is intuitively clear: given a static warning for a pair of racing events, we should find a pair of dynamic events on a matching shared memory location. Here, "matching" merely means that the program locations match up; the exact choice of pair(s) of dynamic events that we use is left as an implementation choice.

At the same time, we can reject a static race if the test suite confirms that one of the events in the race report can occur but offers no evidence that a pair of events matching the static race report occur dynamically.

Specifically:

[5] This was brought to our attention by Stephen Freund, during the conference presentation of our work.

- We *confirm* a statically detected race if we can find a dynamic event pair with a shared memory location that can act as a dynamic witness to the race.

- We *reject* any statically detected races that we cannot confirm and for which we have at least one matching dynamic memory access.

The correctness of a static race that is neither confirmed nor rejected is undetermined. From our past experience, undetermined races are generally due to a sparse test suite.

Observing dynamically whether a suspicious memory location is indeed thread-shared is easy. It is also significantly predictive: having the variable be shared is a much more general condition than actually observing a race. Nevertheless, observing a suspicious variable being shared *in conjunction* with the static race report significantly enhances the suspicion of a program error and warrants a reinforced report to the programmer.

To further explore the applicability of residual investigation in this high-value but diverse domain (compared to our main work), we implemented a prototype based on the above insight, applying it to a state-of-the-art static data race detection tool for Java, JChord [81].

2.4.2 Implementation

We tailored our implementation to the needs of confirming or rejecting JChord reports. This task was far from trivial, mainly because of the impedance mismatch in detecting and reporting concurrent events statically vs. dynamically, as discussed next.

- Our implementation applies only to races in non-JDK classes. A lot of JChord's reports are on fields or arrays inside JDK classes instead of fields or arrays inside the application classes. For example, if an application class contains a field of type `java.io.ByteArrayInputStream` and there are some races that can occur on the `ByteArrayInputStream` object's fields, JChord will report those races on the JDK class fields (e.g., a race will be reported on a field such as `buf` or `pos` inside the

`ByteArrayInputStream` object). However, it is hard to map this to information collected from dynamic analyses. Without modifications of the underlying Java Virtual Machine, current dynamic race analysis can only report races on fields or arrays inside non-JDK classes (e.g., the field of type `ByteArrayInputStream` itself). The reason is that dynamic race analysis typically has to instrument each field or array of each class. Doing so for JDK classes would crash the Java Virtual Machine. To process JChord's reports through dynamic thread-local analysis, we limit our focus to races on shared variables inside non-JDK classes.

- We use dynamic stack traces to match paths of call sites in JChord's reports. Each JChord race report contains a field or an array, a pair of accesses represented by a pair of (call-)paths in the call graphs leading to a race on that field, and a pair of access types (read or write). Each path begins with the call site of the `main` method or the `run` method of class `java.lang.Thread` and ends with the call site of the access (at least one of them is a write) to the field. The internal states that we want to collect dynamically are quite similar. When a dynamic memory access event happens, if the memory sharing condition holds, we collect the field or array name representing the memory location, the most recent and current stack traces of accesses to the same memory location, and their access types. We use stack traces because static call sites are most accurately identified by using the full dynamic stack trace of the access instruction. However, collecting stack traces incurs a heavy runtime overhead. We minimize this overhead heuristically by collecting stack traces only for the first N unique accesses to a shared memory location. An access is uniquely determined given the stack trace and name of the variable on which the access is deemed to occur, i.e., if two memory accesses are by the same instruction with different stack traces or different variable names, they are considered different accesses. Furthermore, we only collect traces for the first M unique accesses to the same field/array in the program text. (The same field of different objects can point to different memory locations, so $M > N$. Currently $M = 300$, $N = 100$.) Once these limits are reached, any access to the memory location, or to all memory locations pointed by the same field is subsequently ignored.

The dynamic thread-local analysis is implemented using Roadrunner [38]. Roadrunner manages the low-level details of adding probe code to the target program's bytecode at load time and creates a stream of events for field and array accesses, thread joins, etc. Our probe code performs thread locality checks, while recording the states of both thread-local heap memory locations and shared memory locations.

Our residual investigation of JChord is accomplished via a three-phase algorithm: *pre-analysis* phase, *instrumented execution* phase, and *cross-checking* phase.

In the *pre-analysis* phase, we identify all fields and arrays that appear in JChord race reports. Accesses to these fields and arrays will be instrumented at class load time in the *instrumented execution* phase.

In the *instrumented execution* phase, we run the instrumented program once and store the application's observations for use in the *cross-checking* phase. There are two kinds of observations that we utilize (with duplicates eliminated) during a dynamic memory access:

- *shared access observations*, which are quintuples $\langle f, r', a', r, a \rangle$ if stack traces r' and r for accesses a' and a are accessing the same memory location of field/array f (with or without synchronization) and at least one of a' and a is a write. The components of this quintuple correspond directly to a JChord race report. We check whether the most recent access to the same memory location as a or a' was by a different thread (establishing an access to a shared memory location, per our initial definition) and at least one of the two accesses is a write. If so, the algorithm records the shared access observation $\langle f, r', a', r, a \rangle$.

- *single access observations*, which are triples $\langle f, r, a \rangle$. All access history for all memory locations is recorded. The components of a single access observation correspond directly to either endpoint of a JChord race report. We use these triples to reject JChord races that cannot be confirmed by any shared access observations.

We pre-process the above shared and single access observations, since dynamic stack traces contain richer information than static call-paths. For instance:

a) There are extra method calls in stack traces by Roadrunner: For example, methods $\langle init\rangle$ or $\langle clinit\rangle$ appear for instance or class initializers; every time an instrumented method is called, an extra call to a dynamically generated method (specific to the method called) is recorded.

b) In dynamic traces, the instrumented method name is getting prefixed with `$rr__Original___$rr` by Roadrunner.

c) There are missing method calls in JChord's call sites due to its imprecise call-graph construction. Missing methods are ignored when matching is performed in the *cross-checking* phase.

In addition, recall that establishing the shared memory condition requires checking that the accessing threads are not joined. To do this, in the event of a thread joining, the id of the joining thread is recorded. Every time a memory location is accessed, the most recent access is considered to be able to share the memory with the current access only if the thread id of the most recent access is not one of a joined thread.

Another factor to consider is that operations inside probe code can be involved in concurrency errors since they may be executed concurrently by the same threads they are monitoring. To avoid concurrency errors in probe code, synchronization is added to the code that will affect the reporting of a race, such as checking thread-locality and updating the most-recent-access information for the memory location.

The *instrumented execution* phase computes, in the form of the above access observations, all the dynamic information needed to classify static races. The subsequent *cross-checking* phase performs a two-pass matching: a first pass is responsible for confirming

JChord races and a second pass is responsible for rejecting the JChord races that are not confirmed. When confirming a JChord race, we check pairs of events in the JChord race against matching shared access observations (which we interpret as dynamic event pairs). If we do not find a shared access observation as witness to the static report, the JChord race is not confirmed and we continue to check its events against every single access observation to see if we can reject it, again following our earlier definitions. If we cannot reject the JChord race either, its classification is undetermined.

A critical step in either confirming or rejecting a JChord race is to define the logic of matching static events against dynamic events. We consider a dynamic event and a static event to match if (a) both refer to the same field or array name, (b) both have the same access type (read or write), and (c) the call stacks of the statically predicted invocation are *similar* to the ones in the dynamic invocation. For our purposes, two call stacks (one static, one dynamic) are similar iff all stack frames from the static invocation occur in the dynamic invocation, in order (but possibly separated by other stack frames). For instance, the call sites of the sequence $\langle foo, bar, baz \rangle$ appear in order in the stack frame sequence $\langle foo, bar, foobar, baz, fun \rangle$.

2.4.3 Experiments

We evaluate the efficiency of our prototype implementation by applying it to 4 open-source Java projects coupled with test harnesses: EM3D, JUnit, Lucene, and Weka. These test subjects were used in prior literature [88] and are heterogeneous: their domains are varied and their code sizes range from hundreds to hundreds of thousands lines. Also, as in prior work [88], we configure JChord so that:

- it reports races between instructions belonging to the same abstract thread.
- it filters out races between the non-main threads and main thread because the benchmarks only have parallelism in the form of parallel loops [83]. Reports of races

	JChord	Residual Investigation	
project	Races	Confirmed	Rejected
EM3D	11	0	4
JUnit	6	2	3
Lucene	3132	0	49
Weka	721	0	71

Figure 2.7: Summary of our experiments on race detection.

between the main thread and non-main threads are clearly false positives for these benchmarks.

- it ignores races in constructors.
- it does not use conditional must-not-alias analysis [80], as it is not currently available.
- it excludes a large number of known thread-safe JDK classes from analyses. A class is thread-safe if any invocation of it cannot be involved in races.

Our experimental results show that residual investigation precisely identifies real races and rejects false JChord warnings, in reasonable running time. All analyses were run on a machine with a 4-core Intel Xeon E5530, 2.40GHz processor and 24GB of RAM, using Oracle JDK version 1.6.45, with the maximum heap size set to 24GB.

Figure 2.7 shows the number of JChord races that our tool can confirm or reject in each benchmark. To assess how many of these confirmations and rejections are correct, we manually inspected JChord races for EM3D and JUnit. We did not inspect race warnings for Lucene and Weka due to a high number of warnings by JChord and the large size and non-trivial complexity of both Lucene and Weka.

Out of 11 EM3D races detected by JChord, 4 were rejected by our tool. Further inspection revealed that the 4 rejections were justified: EM3D is race-free. JUnit gave rise to 6 JChord races and 2 of them were confirmed by our tool. Examining the source code indicated that the 2 confirmed races were real races, both of which were related to unsynchronized accesses to a field used for printing JUnit test result. Again for JUnit, our tool

	JChord	Residual Investigation			Plain
project		Pre	Instrumented	Checking	
EM3D	51.00	0.01	0.81	0.01	0.11
JUnit	125.67	5.61	6.50	15.50	0.36
Lucene	138.67	18.25	15.08	23.62	6.56
Weka	150.67	37.79	946.29	35.13	3.90

Figure 2.8: Average running time in seconds for residual investigation on race detection.

correctly rejected 3 JChord races: All of these 3 pairs of accesses were well-ordered. For the remaining 2 benchmarks, Lucene and Weka, our tool classified fewer than 10% of the JChord race warnings as confirmed or rejected.

Figure 2.8 collects the runtime overhead of our residual investigation tool, in seconds. "JChord" gives the running time of JChord for the benchmark. "Pre", "Instrumented", and "Checking" report the running time in the *pre-analysis* phase, *instrumented execution* phase, and *cross-checking* phase of residual investigation, respectively. "Plain" reports the running time of the benchmark programs without any instrumentation.

The overall runtime overhead required for the three phases of residual investigation is occasionally significant but always practically feasible (especially considering the human time benefits that a better race classification affords). In particular, as can be seen in the "Instrumented" column, applications typically slow down by a factor of 2 to 20 during the *instrumented execution* phase, although the memory-intensive Weka results in roughly 300x slowdown. This is hardly surprising. Dynamic race detection tools, such as Eraser [91] and FastTrack [37], also incur a large slow-down on *Weka*. Eraser also suffers a roughly 300x slowdown; FastTrack cannot finish running in 24 hours since it requires $O(n)$ time, with n being the total number of threads created, and for Weka n grows to 15021.

2.5 Greater Applicability of Residual Investigation in Other Domains

In the following, we discuss several kinds of analyses to which residual investigation is applicable. We also present our experimental experience from attempting to apply residual investigation to randomly chosen open-source projects. This experience illustrates vividly the standard caveat not just of residual investigation but of every dynamic analysis: the quality of results depends on the test suite.

2.5.1 Applicability to Other Analyses

Static analysis typically attempts to be exhaustive and, thus, suffers the possibility of false positives. The challenge for applying residual investigation is to identify common failure scenarios that affect the false positive rates of the analysis yet can be easily ascertained dynamically, even when no fault is observed.

2.5.1.1 Flow Analyses: SQL Injection

Another kind of static analysis that will often result in false positives is analysis that deals with string operations and attempts to discern the structure of programmatically generated strings. An example of this is SQL injection analysis for SQL queries created in code written in an imperative language (typically Java, C++, C#). Although powerful analyses have been proposed (e.g., [49]), it is difficult to make an accurate determination of the possible contents of generated strings, and this difficulty is compounded with the obligation to also track how the strings can be used. Residual investigation approaches can be of help in this domain. For instance, in a course-based evaluation of the industrial static code analyzer Fortify[6], D'Souza et al. [31] observe false positives in "the report of a SQL injection at places where [they] did not find any hint of a dynamic SQL query generation." This observation directly suggests an easy dynamic test ("does the suspicious string ever get included in a SQL query?") that will eliminate the false positives, yet will allow reporting the error even when the current execution does not exhibit any fault. Thus, our argument is that if we combine a static SQL injection report with dynamic taint analysis [82] (i.e.,

[6] `http://www8.hp.com/us/en/software-solutions/static-code-analysis-sast/index.html`

tracking of which user-influenced strings flow where) we get a much higher-confidence SQL injection report. Note that in this case the taint analysis by itself is not an indication of a fault, unlike other security analyses where tainting of a function's input by user data is direct evidence of a problem.

Note that combinations of static and dynamic analyses for SQL injection detection have already been explored [53,60]. Huang et al. [60] insert dynamic guards to secure a variable that is found to be involed in an insecure statement by static verification. Halfond and Orso [53] use static analysis to build a conservative model of the legitimate queries generated by the program, yet follow it with a dynamic analysis that checks for conformance with that model. These approaches, however, do not quite follow the residual investigation pattern: the static analysis by itself does not detect errors but prepares the ground for a dynamic analysis that will report faults. Still, the residual investigation approach is largely applicable to this domain, if not already applied, to an extent.

Saner [4] composes static analysis and testing to find bugs of the input validation process for cross-site scripting and SQL injection flaws. Saner statically identifies incorrect or incomplete code sections of input validation routines and uses testing to confirm that these code sections are actually ineffective. Instead of using testing, residual investigation employs predictive dynamic analyses to confirm statically reported bugs. Thus, given the same test suite, residual investigation is less likely to miss a true vulnerability (but still nearly precise).

2.5.1.2 Other FindBugs Patterns

FindBugs currently supports several hundred bug patterns, each with its own analysis, ranging from plain textual matching to more complicated type- and flow-based analyses. We implement a set of bug patterns in RFBI (mostly from the 18 bug patterns in the original FindBugs publication [58]) and examine them in our main evaluation, but, as already mentioned, residual investigation is applicable to a lot more. Examples include:

- The "*Covariant compareTo method defined*" bug pattern reports classes that define method `compareTo` covariantly (i.e., with an argument type that is not `java.lang.Object` but a subtype of it). This, and other bug patterns on erroneous overriding of standard methods, can be handled in much the same way as the "Bad Covariant Definition of Equals" pattern, discussed earlier.

- The "*Private method is never called*" pattern is a source of false positives because some private methods are intended to be called only via reflection. In this case the method is private for static code, but public for reflective code. For this to happen, the `setAccessible` method needs to be called on the `Method` object corresponding to the suspicious method. Thus, the presence of a `setAccessible` call on this method is a hint to suppress the static error report.

- Reflection and dynamic loading are common in Java enterprise applications and invalidate several patterns. Yet the uses of reflection are typically simple and only limited aspects of reflective behavior change dynamically. The "*Call to equals comparing unrelated class and interface*" bug pattern, as well as the "*Call to equals comparing different interface types*" pattern both assume a closed view of the inheritance hierarchy. Dynamic code can invalidate the patterns by loading a new class that is a subtype of both compared types. The error warning is thus reinforced if we track loaded classes and observe that, indeed, while running the test suite no such common subtype arises.

2.5.2 Applicability Relative to Test Suites

It is worth considering the preconditions for applicability of residual investigation in terms of the analyzed program. Clearly, the program being examined should have a means for being exercised under representative usage. That is, one needs either an off-line system-level test suite or a human tester that will run the program through actual usage scenarios. In the absence of the above, residual investigation is not likely to bear fruit.

The state of the practice regarding test suites is rather grim, however. The vast majority of open source projects maintain no test suite alongside the code, or only integrate a rudimentary unit test suite. For an example, we considered the SF100 benchmark suite [42], which has recently been the focus of significant attention. SF100 selects at random 100 open-source projects from the SourceForge repository and puts them forward as a statistically sound sample for empirical SE results. We studied at random 40 of the 100 benchmark programs. A handful of them were further excluded as they were outdated—e.g., providing an API for Amazon's XML Web Service 3.0, which has been long replaced by Amazon Cloud Computing. (These benchmarks could not run in our environment, although the intent of the SF100 suite is to include all dependencies in the benchmark package.) We found that the vast majority of the SF100 projects do not contain a test suite for unattended system-level testing. This is perhaps expected, since most of the SF100 benchmarks are small-to-medium size: more than four-fifths of the programs have fewer than 100 classes, more than one-fifth have fewer than 10.

As a result, RFBI did not produce any useful results for our sampling of SF100 programs: the dynamic potential (DP) metric for all bugs was zero, reducing RFBI to merely FindBugs, which often also did not manage to produce good reports. This reinforces the dependence of residual investigation on a reasonable test suite. As already seen, this is no more stringent a requirement than already satisfied by test suites supplied by the creators of large, mature open-source projects (such as JBoss, BCEL, NetBeans, Tomcat, etc.).

Of course, this does not mean that residual investigation is not useful for the SF100 programs. However, it has to be used in conjunction with the usual testing process that developers employ for these programs. This is likely to be manual testing under inputs representative of actual usage. In principle, RFBI is fully compatible with the notion of manual testing: instead of collecting dynamic program behavior under automatic test execution, we could also collect this information during manual program execution. Generally, testing for residual investigation has a low bar to clear—we do not need to amass many observations, since just a few runs of a program are enough to reinforce or discount static warnings.

Secondarily, we posit that statistically sound benchmark samples, such as the SF100, will be of greater value if they also include a *biased* sampling of applications based on specific characteristics, such as size or the integration of a complete test suite. Some empirical SE studies (e.g., on automatic test generation, which was the first application of the SF100 [42]) have no such need, but others (such as our study) do.

2.5.3 Applicability to Refactoring

Refactoring is the process of changing a program's structure without altering its behavior. Developers have used test suites to manually gauge if behavior remains unchanged [41], but automatic refactoring tools provide a complementary approach through static analysis [50, 77]. These static analyses, in turn, necessarily come with trade-offs in the same way that the analyses of static bug-checkers do. Consider a seemingly innocuous refactoring such as "rename." This refactoring will systematically replace all occurrences of one particular identifier (e.g., a method name) throughout the program by another. In a language like Java, this transformation is not necessarily behavior-preserving, as the relevant method might be called through reflection, or via an external client program that is not visible to the refactoring engine.

Existing refactoring tools might thus plausibly reject any attempt at renaming a large subset of Java identifiers, i.e., those that could be read or modified via Java reflection. For example, program metamorphosis [89], a generalization of traditional refactoring, materializes such concerns as persistent warnings that users can review during the refactoring process. This is analogous to a static bug report issued by FindBugs in that developers may now raise objections, such as "this method is private and not supposed to ever be called via reflection."

Production refactoring tools anticipate this objection to the point where they concede defeat to it: we are not aware of any production refactoring engine that will reject a rename refactoring because it cannot exclude the possibility of behavioral change.

Residual investigation can serve to improve recall of such lenient refactoring systems, as well as precision of more conservative systems (such as the aforementioned program metamorphosis system). For example, for any newly renamed method, a refactoring system can statically determine name captures (e.g., accidental method overriding or overloading). The system can determine *some* instances of behavioral change due to reflection, by analyzing string usage [22]. To detect the remaining cases of reflective method invocation, we can introduce two dynamic checks: (1) a search for reflective invocations of methods with the old (pre-renaming) name, regardless of the dynamic type of their target, and (2) any use of reflection on instances of the refactored class (perhaps including its superclasses). If either or even both checks trigger, the risk of a behavioral change increases and may now warrant a more severe warning to the user.

These dynamic checks can then complement the existing static checks in the same vein as the dynamic analyses we have described for static bug checkers: whenever a static check cannot conclusively determine that the refactoring is safe or unsafe, it can emit the above checks as additional safeguards of behavior preservation.

CHAPTER 3

SECOND-ORDER CONSTRAINTS IN DYNAMIC INVARIANT INFERENCE

This chapter centers around the enhancement of dynamic invariant inference with static second-order constraints. We call these constraints second-order because they are constraints over constraints: they relate classes of invariants (first-order constraints). Such second-order constraints are provided directly by programmers or vetted by programmers among candidates suggested automatically by a tool we developed. By incorporating second-order static information about invariants, we will improve the quality of derived invariants in ways that greatly enhance the utility of dynamically discovered invariants in program analysis, testing, and program understanding.

3.1 Introduction and Motivation

Systematically understanding program properties is a task at the core of many software engineering activities. Software testing, software documentation, and software maintenance benefit directly from any significant advance in automatically inferring program behavior. *Dynamic invariant detection* is an intriguing behavior inference approach that has received considerable attention in the recent research literature. Tools such as Daikon [36], DIDUCE [54], and DySy [28] monitor a large number of program executions and heuristically infer abstract logical properties of the program, expressed as invariants.

Dynamic invariant inference is fundamentally a search in the space of propositions over program variables for the tiny subset of propositions that are both true (on all executions) and relevant (to the programmer, or to some client analysis such as a theorem prover or test data generator). Any such search requires a way of generating candidate propositions and a way of evaluating them. Even limiting consideration to propositions of fixed length, over a particular vocabulary of operations and relations among variables, the space is enormous, and for a long time it seemed folly to even attempt such a search based on dynamic program execution. Daikon [36] was the first demonstration that a generate-and-test tactic could be feasible, generating a moderately large set of candidate propositions from syntactic templates and quickly pruning the vast majority as they were refuted by program execution.

In this work we enhance the dynamic invariant detection approach by including second-order constraints. The invariants inferred should be consistent with these constraints. More specifically, we identify a vocabulary of constraints on inferred invariants. We call these constraints "second-order" because they are constraints over constraints: they relate classes of invariants (first-order constraints). Such second-order constraints can be known even though the invariants are unknown. (To avoid confusion, we try to consistently use the term "constraints" for second-order constraints, as opposed to the first-order "invariants", although the term "second-order invariants" would have been equally applicable.) Our vocabulary includes concepts such as *Subdomain* (one method's precondition implies that of another); *Subrange* (one method's postcondition implies that of another); *CanFollow* (one method's postcondition implies the precondition of another); *Concord* (two methods satisfy a common property for the common part of their domains, although some inputs for one may be invalid for the other); and more.

Such constraints arise fairly naturally. Two methods may be applicable at exactly the same state of an input. For instance, the `top` and `pop` methods of a stack data structure will have constraints:

```
Subdomain(top, pop)
Subdomain(pop, top)
```

This signifies that their preconditions are equivalent. As another example, a method may be a specialized implementation of another (e.g., a matrix multiplication with a faster algorithm, applicable to upper triangular matrices only, relative to a general matrix multiplier). This induces a `Subdomain(triangMatMul(), matMul())` constraint. Similarly, a method may be preparing an object's state for other operations, as in the common case of `open` and `read`: `CanFollow(open(), read())`.

The purpose served by second-order constraints is dual:

- First, when the constraints are discovered dynamically, without any human effort expended, they can serve as a concise and deeper documentation of program behavior—e.g., a single second-order constraint can distill the meaning of many Daikon invariants. This can be seen as an effort to generalize the observations that the Daikon system already makes via sophisticated post-processing. For instance, state-machine behavior (e.g., calls to method `foo` can follow calls to method `bar`) can be documented in the vocabulary of second-order constraints.

- Second, when the (second-order) constraints are either supplied by the programmer or vetted by the programmer (among candidate constraints suggested automatically) they can help enhance dynamically inferred (first-order) invariants by effectively cross-checking invariants against others. In this way, the noisy inferences caused by having a limited number of dynamic observations are reduced: the second-order constraint links the invariant at a certain program point with a larger set of observed values from different program points. In exchange for modest specification effort, second-order constraints can eliminate inconsistencies and produce more relevant and more likely-correct invariants.

In more detail, the main contributions of this chapter are as follows:

- We identify and describe the idea of second-order constraints that help improve the consistency and relevance of dynamically inferred invariants.

- We define a vocabulary of common second-order constraints.

- We present a generalization of Daikon's dynamic inference approach to second-order constraints, by appropriately defining the Daikon "confidence" metric for second-order properties.

- We discuss how second-order constraints can be implemented, how their enforcement is affected by properties of the dynamic invariant inference system (we identify *monotonicity* as a particularly important property). We provide an implementation of the constraints in our vocabulary in the context of the Daikon system.

- We provide experimental evidence for the value of second-order constraints and their dynamic inference. We evaluate our approach both in controlled micro-benchmarks and in realistic evaluations with the Apache Commons Collections and the AspectJ compiler. A low-cost effort to write constraints (e.g., a few hours spent to produce 26 constraints) results in significant differences in the dynamically inferred invariants. Furthermore, the vast majority (99%) of a random sample among our dynamically inferred second-order constraints are found to be true.

3.2 Invariants and Constraints

We begin with a brief background on dynamic invariant inference, and subsequently present our vocabulary of second-order constraints. For illustration purposes, our discussion is in the context of Java and the Daikon invariant inference tool. This is the most mainstream combination of programming language and dynamic invariant inference tool, but similar observations apply to different contexts and tools.

3.2.1 Daikon and Dynamic Invariant Inference

Daikon [36, 87] tracks a program's variables during execution and generalizes their observed behavior to invariants. Daikon instruments the program, 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). 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 discards those invariants that are refuted by an execution trace—for example, it might process a situation where `m` returned null and it would therefore discard the above invariant. Daikon summarizes the behavior observed in the execution traces as invariants and generalizes it by proposing that the invariants hold in all other executions as well. Daikon can annotate the program's source code with the inferred invariants as preconditions, postconditions, and class invariants in the JML [69] specification language for Java.

3.2.2 A Vocabulary of Constraints

To express rich constraints on invariants, we use a vocabulary including constraints such as *Subdomain* and *Subrange*. These constraints are specified explicitly by the user of the invariant inference system by employing an annotation language. Furthermore, we give the user an opportunity to express one more useful kind of information regarding inferred constraints: a set of "care-about" program variables and fields that are allowed in inferred invariants. Our vocabulary includes the following concepts:

- *OnlyCareAboutVariable(⟨var⟩)* and *OnlyCareAboutField(⟨fld⟩)*: These annotations instruct the invariant inference system that the derived invariants of a method should only contain the listed method formal argument variables or class fields. This prevents the system from deriving facts irrelevant to the property under examination, which would also be more likely to be erroneous. Additionally, it allows restricting invariants so that complex constraints can be expressed. The *OnlyCareAbout...* constraints are not consistency constraints on invariants, but they are essential background for use together with the consistency constraints that follow.

- *Subdomain(m1,m2)*: The precondition of method *m1* implies the precondition of method *m2*. For instance, the author of a matrix library may specify the constraint:

  `Subdomain(processDiagonal, processUpperTriangular)`

  This means that method `processDiagonal`, which works on diagonal matrices, has a precondition that implies that of method `processUpperTriangular`, which works on upper triangular matrices: the latter method is applicable wherever the former is. (Matrix libraries often have a large number of related operations, in terms of interface and applicability, but with different implementations for performance reasons [52]. In this section, we draw several examples from this domain for illustration purposes.)

  Preconditions are not only expressible on method parameters but also on member fields of an object. E.g., one may specify that the preconditions of method `append` imply those of method `write`: If the object is in an appendable state, it can also be written to.

  In order to specify useful *Subdomain* constraints, one may need to limit the inferred invariants by employing the *OnlyCareAboutVariable* and *OnlyCareAboutField* annotations presented earlier. (This also holds for many of the later constraints.) For instance, methods *m1* and *m2* may be on separate objects. Thus, their invariants would normally include distinct sets of variables. Yet if we specify their common variables with *OnlyCareAboutVariable*, then the *Subdomain* constraint can be applicable.[1]

- *Subrange(m1,m2)*: This constraint is analogous to *Subdomain* but for postconditions. It means that the postcondition of method *m1* is stronger than (i.e., implies) that of *m2*. Again, the condition may apply to parameter or return values—e.g., a method with postcondition `isDiagonal(return)` is related with *Subrange* to one with postcondition

[1] Our implementation provides some default restrictions on which variables are considered when second-order constraints are used, so that common cases are covered without explicit *OnlyCareAbout...* annotations. Namely, when two methods are linked by a second-order constraint, their invariants are defined over "matching" arguments and fields. Roughly, argument variables match if they are in the same position and the methods have the same arity, and class fields match if they have the same name and type.

`isUpperTriangular(return)`. (Where `return` stands for the return value of the method.) Yet the condition can also be on object state. For instance, we may have constraints such as:
`Subrange(listTail,listRange)`

This means that the state of the return value and overall object after execution of the method `listTail` is also a valid state after `listRange`: the former method produces a subset of the possible results and effects of the latter.

- *CanFollow(m1,m2)*: This constraint means that the postcondition of method *m1* implies the precondition of method *m2*. Execution of method *m2* is enabled by execution of *m1* in this sense, and it is a natural way to capture temporal sequencing rules, e.g.,:
`CanFollow(open,write)`

  Such specifications are common and are often expressed in the form of state machines. Note that the above constraint means that "an `open` can be followed by a `write`", not "a `write` has to be preceded by an `open`". (The latter is inexpressible, since our vocabulary is only concerned with the relationship between conditions at two program points, but there is nothing to prohibit other program points from establishing the same conditions.)

- *Follows(m1,m2)*: This constraint means that the precondition of method *m2* implies the postcondition of method *m1*. Informally, this means that the only states that *m2* can start from are states that *m1 may* result in. *m1* does not guarantee such a state, though: *m1*'s postcondition is necessary but not sufficient for preparing to execute *m2*. In practice, this constraint prevents the postcondition of *m1* from being more specific than necessary. For instance, we may have:
`Follows(add,remove)`

  This does not signify that `add` needs to execute before `remove` (a different method may be reaching the same states as `add`). But it means that if we are in a state where the last operation cannot have been an `add` (e.g., an empty structure), then we cannot `remove`.

Constraint	**Pre/Post**	**Dataflow**
Subdomain(m_1, m_2)	$Pre_{m_1} \implies Pre_{m_2}$	$\mathsf{in}_t(m_1) \subseteq \mathsf{in}_t(m_2)$
Subrange(m_1, m_2)	$Post_{m_1} \implies Post_{m_2}$	$\mathsf{out}_t(m_1) \subseteq \mathsf{out}_t(m_2)$
CanFollow(m_1, m_2)	$Post_{m_1} \implies Pre_{m_2}$	$\mathsf{out}_t(m_1) \subseteq \mathsf{in}_t(m_2)$
Follows(m_1, m_2)	$Pre_{m_2} \implies Post_{m_1}$	$\mathsf{in}_t(m_2) \subseteq \mathsf{out}_t(m_1)$
Concord(m_1, m_2)	$Pre_{m_1} \wedge Pre_{m_2} \implies (Post_{m_1} \implies Post_{m_2})$	$Pre_{m_1}(t) \wedge Pre_{m_2}(t) \implies \mathsf{out}_t(m_1) \subseteq \mathsf{out}_t(m_2)$

Figure 3.1: Meanings of second-order constraints.

- *Concord(m1,m2)*: *Concord* means that the postcondition of method *m1* implies that of method *m2* but only for the values that satisfy the preconditions of both *m1* and *m2* (or, put another way, only for the values for which the precondition of *m1* implies that of *m2*). This is a valuable constraint for methods that specialize other methods. For instance, there can be a fully general matrix multiply routine, a more efficient one for when the first argument is an upper triangular matrix, one for when the first argument is a diagonal matrix, etc. The operations' invariants are linked with a *Concord* constraint:

  ```
  Concord(triangularMultiply, matrixMultiply)
  ```

  Thus, the postcondition of the specialized operation (`triangularMultiply`) should imply that of the more general operation (indeed, in this case the conditions should be equivalent) for all their common inputs. Nevertheless, the general operation, `matrixMultiply`, is applicable in more cases and, thus, has a weaker precondition than the specialized one. For inputs valid for `matrixMultiply` but not `triangularMultiply`, the *Concord* constraint imposes no restriction.

3.2.3 Meaning and Completeness of Second-Order Constraints

We intend for second-order constraints to be devices for expressing relationships between program elements at a high level. This means that the precise interpretation of the constraints will depend on the specifics of the invariant inference system we use and on its capabilities. As a guideline, we present the intended meaning of five of our constraints in Figure 3.1, column *Pre/Post*. (We omit the 'OnlyCareAbout' predicates as they are straightforward.) In that column we summarize the meanings of each constraint as logical relations over the preconditions Pre_m and postconditions Post_m of all involved methods.

We assume in this presentation that all preconditions use the same names for parameters in the same position, and all postconditions use the same names for their return value. In other words, $\mathit{Pre}_{m_1} \implies \mathit{Pre}_{m_2}$ means that any constraints that Pre_{m_1} places on the first (second, third, $\ldots$) parameter of m_1, Pre_{m_2} also places on the first (second, third, $\ldots$) parameter of m_2.

As we can see, the first four predicates capture all possible 'straightforward' implications between the preconditions and postconditions of two methods. Predicate *Concord* meanwhile captures a more complex but practically important second-order constraint.

We will return to this figure later, when we describe our implementation.

3.2.4 Discussion

Second-order constraints can serve as documentation of program behavior in much the same way that first-order invariants can (i.e., for program understanding purposes). Furthermore, second-order constraints can offer external knowledge that helps steer invariant inference in the right direction. Certainly one reason to do this is to avoid erroneous invariants: second-order constraints serve to cross-check invariants and thus enhance the dynamic opportunities to invalidate them, even with a limited number of observations. Another important use of constraints, however, is in deriving more *relevant* invariants. Clearly, a simple use of *OnlyCareAbout...* can prevent some irrelevant invariants. Greater benefit can be obtained by more abstract constraints, however. Consider, for example, a constraint on the constructor of a class `C` and on one of `C`'s methods, `m`. If we have the constraint *Follows*(`C`, `m`) then we have a hint that the constructor establishes part of the precondition for `m`. Without the constraint, we may observe the constructor's execution and derive for a certain field `fld` of the class the postcondition "`this.fld == 100`". This inference would certainly be reasonable, if, whenever the constructor is executed, `fld` is assigned the value 100. Yet the invariant could be overly specific. Given that we know that the constructor's postcondition is implied by `m`'s precondition, we can observe all the different circumstances when `m` is called (not necessarily right after construction). This may yield the broader precondition "`this.fld > 0`" for `m`. Because of the *Follows* constraint, the constructor needs to also register all the same values and, hence, changes its postcondition to "`this.fld > 0`" instead of the more specific "`this.fld == 100`". The generality can mean that the new postcondition is more meaningful from a program understanding standpoint. The specific value 100 may be just an artifact of the specific test cases used (i.e., the postcondition "`this.fld == 100`" may be erroneously over-specific) but, even if it is correct, it may be the result of an arbitrary technicality or a detail likely to change in the next version of the program. Certainly, the *Follows* constraint steers the invariant inference to the kind of invariant the user wants to see in this case: the condition that (partly) enables `m` to run.

It is worth noting that the principle of *behavioral subtyping* [69] can be viewed as a combination of constraints from our vocabulary. Informally, behavioral subtyping is the requirement that an 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 [24]) and design methodologies (e.g., "subcontracting" in Design by Contract [78, p.576]). When a method `C.m` overrides another, `S.m`, the behavioral subtyping constraints consist of a combination of *Subdomain*(`S.m`, `C.m`) and *Concord*(`C.m`, `S.m`). (Implicitly there are also *OnlyCareAboutVariable* and *OnlyCareAboutField* constraints that restrict both the preconditions and postconditions to be over parameters of method `m` and member variables of the superclass `S`.) In past work [27] we discussed how consistent behavioral subtyping can be supported in the context of dynamic invariant inference systems, and similar ideas have informed the present work.

3.3 Design and Implementation

We next discuss the crucial design questions regarding a second-order constraints facility, as well as our own implementation decisions. Before this, however, we introduce the concept of monotonicity for a dynamic invariant inference system. This concept plays a central role in our discussion and understanding.

3.3.1 Background: Monotonicity

An important consideration for our later implementation arguments is whether our invariant inference system is (mostly) *monotonic*: If supplied more values to observe, the conditions it will output will be logically weaker (i.e., broader). Note that monotonicity applies to the logical predicates and not to the number of invariants. There may be fewer invariants produced when more values are observed, but these will be implied by the invariants produced for any subset of the values.

Most dynamic invariant inference tools are in principle monotonic. Tools like DySy [28] and Krystal [64] use a combination of dynamic and symbolic execution. The invariants are computed from the boolean conditions that occur inside the program text, when these are symbolically evaluated. The more executions are observed, the more the invariants of a program point are weakened by the addition of extra symbolic predicates (in a disjunction). Similarly, tools like Daikon or DIDUCE have invariant templates that are (multiply) instantiated to produce concrete candidate invariants. The candidate invariants are conceptually organized in hierarchies from more to less specific. Values observed during execution are used to falsify invariants and remove them from the candidate set. The *most specific* non-falsified candidate invariant of each hierarchy is a reported invariant. For instance, if two candidate invariants "`x == 0`" and "`x >= 0`" are satisfied by all observed values of `x`, then the former will be reported, as it is more specific. This is a monotonic approach: It means that as candidate invariants are falsified, they are replaced by more general conditions.

Nevertheless, invariant detection tools often include non-monotonicities in their inference logic. For instance, Daikon uses a *confidence* threshold for invariants: An invariant is not output even when consistent with all observations, if the number of pertinent observations is small. Similarly, Daikon treats some boundary values and invariant forms specially. For instance, if the maximum value for a variable is -2, Daikon will not output an invariant of the form `x <= -2`, but if it additionally observes the value 0 for the variable, it will output `x <= 0` (an invariant that would have been true even without observing the 0 value, but would not have been reported). Furthermore, Daikon produces the weakest predicate (*true*) when no invariants can be inferred. These features entail non-monotonicity: Observing more inputs will produce a stricter invariant.

In general, some non-monotonicity is unavoidable in practically useful dynamic invariant inference tools, even if just in corner cases. For instance, if a tool is strictly monotonic, then it has to infer a precondition and postcondition of *false* for every method that happens to not be exercised by the test suite. (This is the only logical condition that is guaranteed to be weakened—as monotonicity prescribes—when we add an execution that does exercise the method.) Computing invariants of *false* is a sound approach (accurately captures observations) but largely useless. Any further use of the produced invariants cannot employ the method in any way, as it can never establish the preconditions for calling it or know anything about its results. Effectively, monotonic invariant inference would have to report that a method is forever unusable if it was not observed to be used in the test run. This and other similar examples in practically significant cases preclude the use of strictly monotonic dynamic invariant inference.

3.3.2 Implementation

We implemented second-order constraints in the Daikon system. (Daikon is not only the mainstream option for dynamic invariant inference, but also convenient for engineering purposes. The DySy tool [28], on which we have worked in the past, was another interesting prospect, but requires C#/.NET and a specialized, closed-source, dynamic-symbolic execution infrastructure.) There are two aspects to the implementation. First, we infer second-order automatically through dynamic observations of program behavior. Second, we allow treating second-order constraints as "ground truth" and use them to influence regular invariant inference.

***Dynamically inferring second-order constraints*:** The key construct in most of our second-order constraint is an implication between pre/post conditions *P* and *Q* at two different program points (with each one being either the entry or the exit of a program procedure). Our strategy for proving the second-order constraint $P \implies Q$ is to first use Daikon (Rev. 2eded4baf181) to produce *P* and *Q* and then employ the Simplify theorem prover to check whether *P* can imply *Q*. The full implication check may fail, even if the implication holds in principle. One reason for this is that an inadequate test suite may lead Daikon to produce a slightly inaccurate *P* or *Q*. Furthermore, the theorem prover may be unable to decide whether an implication is satisfiable or not. Therefore, a more realistic check, since *P* and *Q* are expressed as conjunctions of invariants, is to see if a particularly large number of invariants in *Q* are subsumed by invariants of *P* (i.e., implied by the full conjunction *P*).

In order to find whether a sufficiently large number of implications between *P* and the constituent invariants of *Q* are true, we define the *success rate* (*SA*) of the implication as $SA = \frac{N}{M}$, where *N* represents the number of invariants in *Q* that can be implied by *P* and *M* denotes the number of invariants of *Q*.

The success rate on its own is often not enough for comparing the likelihood of a second order constraint being true. In our definition of success rate, all invariants are treated equally, so in a case with lots of junk invariants in *P* but not in *Q* that ratio may get overwhelmed by the junk. Daikon uses a *confidence* metric to approximate the probability that an invariant cannot hold by chance alone. If we sum up the confidence values for all invariant implications between *P* and each constituent invariant of *Q*, then the weight of the junk invariants in the ratio will decrease. Thus, we extend the definition of confidence to second-order constraints in order to approximately measure the probability that a second-order constraint could not hold by chance alone. For regular invariants, Daikon defines the confidence of an implication relation (*Inv1* $\implies$ *Inv2*) to be the product of the confidence of the guard (*Inv1*) and the confidence of the consequent (*Inv2*). Similarly, the second-order constraint confidence of $P \implies Q$ can be defined as the mean of the confidences of the implication relations between *P* and each invariant of *Q* that is implied (according to Simplify). Representing the confidences of invariants in *P* by pc_1, pc_2, ..., pc_Z, and denoting the confidences of invariants of *Q* that can be implied by *P* by qc_1, qc_2, ..., qc_N, we calculate the second-order constraint confidence, *MA*, as $MA = \frac{pc_1 \cdot pc_2 \cdot \ldots \cdot pc_Z \cdot (qc_1 + qc_2 + \ldots + qc_N)}{N}$.

We filter out second-order constraints whose success rate is below a threshold (by default 0.75) and rank the unfiltered second-order constraints according to confidence. It is meaningful for the user to change the success rate threshold, possibly based on the accuracy of invariants produced by Daikon for the program at hand. The fewer redundant and irrelevant invariants a program has, the higher the program's success rate filter can be.

The time complexity of our algorithm is quadratic over the number of methods. We consider all combinations of program points inside a class as candidate second-order constraints. If there are n methods in a class, there are $2n$ program points, i.e., $2n(2n-1)$ potential second-order constraints. To verify a candidate second-order constraint, we invoke the Simplify prover on implications between invariants. Removing irrelevant and redundant invariants before applying our tool can help reduce theorem proving time. (Standard avenues for doing so with Daikon include using more representative test suites and employing the DynComp tool [51] for more accurate invariants.)

We use a few heuristics to improve the accuracy of derived second-order constraints. One heuristic is to normalize parameters: a parameter's name is based on its position in the parameter list. For example, if we examine the second-order constraint *Subdomain*(`foo(int i, int j)`,`bar(int x, int y)`) then we do not use the names *i* and *j* when we compare to `bar(int x, int y)`. Instead, we substitute *arg1* for *i* and *x*, and *arg2* for *j* and *y*. We implement this normalization via regular expression transformation on the data trace file generated by Chicory (the Java instrumenter inside Daikon). In addition, if a variable only exists in the consequent-side program point of a second-order constraint, it often gets in the way of verification for the second-order constraint. We ignore invariants that contain such variables in the consequent-side program point. Another heuristic is that invariants over variables that are likely to be meaningless for cross-method comparisons are ignored. For instance, in the case of *Follows*(`bar`, `foo`) and *Subrange*(`foo`, `bar`), we ignore invariants over the *orig(x)* variable (which refers to the value of variable x upon entry to a procedure) at the exit of `bar`; for *Subrange*(`foo`, `bar`) and *CanFollow*(`foo`, `bar`), we ignore procedure parameters at both the entry and exit point of `bar`; for *Follows*(`bar`, `foo`), we ignore the return variable at the exit of `bar`.

***Using second-order constraints for better invariants*:** We extended Daikon with an annotation mechanism for second-order constraints. Users can pass a separate configuration file with second-order constraints to Daikon. This changes the Daikon processing of the low-level observations and allows refining the inferred invariants without having to re-run any test suites.

The main element of our approach is that when the precondition (or postcondition) of a method *m1* is constrained to imply the precondition (resp. postcondition) of another method *m2* (as shown in our earlier Figure 3.1, column *Dataflow*), we propagate the values observed at entry (resp. exit) of *m1* to *m2*. This ensures that producing *m2*'s invariants takes into account all the behavior observed for *m1*. Note that *m2* need not be executed during invariant inference—the conditions observed/established for *m1* are simply registered as if *m2* had also observed/established them. These observations are suitably adapted to be over common variables, as dictated by *OnlyCareAbout...* constraints (including implicit assumptions, as mentioned in Footnote 1).

For the first four second-order constraints of Figure 3.1 the implementation of the above value propagation is straightforward, as it can leverage existing Daikon facilities. Specifically, the implementation extensively hijacks the Daikon *dataflow hierarchy* mechanism. This Daikon internal facility propagates primitive invariants between different program points (a program point is a line of code or the entry or exit from a method). The machinery is therefore quite suitable (with some minor adjustment to suppress filtering in some cases) for implementing our value propagation.

However, the *Concord* second-order constraint is more complex: we only want to flow primitive invariants from $\mathsf{out}_t(m_1)$ to $\mathsf{out}_t(m_2)$ if at execution point t the preconditions of both m_1 and m_2 hold. However, at this point in Daikon's execution we do not yet know what the preconditions of m_1 and m_2 should be, since we have not even seen all primitive invariants yet. We thus evaluate *Concord* constraints in two passes. The first pass stores all data from $\mathsf{in}_t(m_1)$, $\mathsf{in}_t(m_2)$, and $\mathsf{out}_t(m_1)$. At the end of this pass we ask Daikon to compute the preconditions P to m_1 and m_2 using Daikon's usual heuristics (and any other second-order constraints). In the second pass, we now use these preconditions on the data from $\mathsf{in}_t(m_1)$ and $\mathsf{in}_t(m_2)$: $P(m_i, t)$ holds iff $\mathsf{in}_t(m_i)$ satisfies the precondition to m_i at point t. We thus iterate over t one more time to copy all primitive invariants from $\mathsf{out}_t(m_1)$ to $\mathsf{out}_t(m_2)$ and update Daikon's results as needed.

For example, assume that we are examining two methods int m1(int x) and int m2(int y). We make the following observations about m1:

m1: t	$\mathsf{in}_t(\mathsf{m1})$	$\mathsf{out}_t(\mathsf{m1})$
t_1	$\texttt{arg1} = -3$	$\texttt{return} = -1$
t_2	$\texttt{arg1} = 0$	$\texttt{return} = 0$
t_3	$\texttt{arg1} = 3$	$\texttt{return} = 1$
t_4	$\texttt{arg1} = 6$	$\texttt{return} = 2$

That is, the method is invoked at time t_1 with its first (and only) parameter x bound to -3, and returns -1, etc. From the table above, Daikon might plausibly infer the precondition $\mathit{Pre}_{\mathsf{m1}} \equiv \mathsf{true}$ and postcondition $\mathit{Post}_{\mathsf{m1}} \equiv \texttt{return} = \texttt{arg1}/3$.

Now assume that we make the following observations about m2:

m1: t	$\mathsf{in}_t(\mathsf{m2})$	$\mathsf{out}_t(\mathsf{m2})$
t_5	$\texttt{arg1} = 0$	$\texttt{return} = 0$
t_6	$\texttt{arg1} = 1$	$\texttt{return} = 0$
t_7	$\texttt{arg1} = 2$	$\texttt{return} = 0$

Here, Daikon sees no negative inputs and a constant output, so it might infer precondition $\mathit{Pre}_{\mathsf{m2}} \equiv \mathsf{arg1} \geq 0$ and postcondition $\mathit{Post}_{\mathsf{m2}} \equiv \mathsf{return} = 0$.

If the user intended m2 to be a specialized version of m1 for nonnegative integers, the precondition for m2 would be correct but the postcondition would be wrong. The user can address this by adding the constraint Concord(m1, m2). This constraint will make our system first compute preconditions and postconditions as above, then compute all the times t at which $\mathit{Pre}_{\mathsf{m1}}(t)$ agrees with $\mathit{Pre}_{\mathsf{m2}}(t)$. $\mathit{Pre}_{\mathsf{m1}}$ is always true, so we will only filter out t_1 via $\mathit{Pre}_{\mathsf{m2}}$. Consequently, Daikon will get to see all inputs and outputs from t_2, t_3 and t_4 in addition to the ones it was already considering for method m2. With this additional data, Daikon can no longer infer $\mathit{Post}_{\mathsf{m2}} \equiv \texttt{return} = 0$ but might instead conclude $\mathit{Post}_{\mathsf{m2}} \equiv \texttt{return} = \texttt{arg1}/3 \wedge \texttt{return} \geq 0$, which matches the user's intention.

3.3.3 Discussion

It is evident from the previous section that our two mechanisms (that of dynamically inferring second-order constraints and that of taking them into account when inferring invariants) operate differently. The former follows a static approach for checking invariant implication: the Simplify system is used as a symbolic prover. However, the mechanism of enforcing invariant implications (to produce different invariants by taking second-order constraints into account) eschews symbolic reasoning in favor of propagating more dynamic observations. Why do we not just take the conjunction of the produced invariants and declared second-order constraints, simplify it symbolically, and report it as the new produced invariants? The reason is that there is noise introduced when generalizing from observed executions to invariants (e.g., because the invariant patterns are uneven), and this carries over to the combined invariants. *It is better to generalize (i.e., compute invariants)* *from more observations than to generalize from fewer ones and then combine the generalizations symbolically.*

The above insight is evident in the case of non-monotonic invariant inferences, which are virtually unavoidable, as mentioned in Section 3.3.1. For instance, consider a *Subdomain(m1,m2)* constraint. If P is the precondition of *m1* and Q is the precondition of *m2*, then we can statically satisfy the constraint by considering the real precondition of *m2* to be $P \vee Q$, so that it is always implied by P. If, however, method *m1* is never called in our test suite, Daikon will infer *true* as its precondition. This will make *true* also be the precondition of *m2*, even though we have actual observations for that method! This result is counter-intuitive and so common in practice as to significantly reduce the value of produced invariants.

In contrast, in our chosen approach, a second-order constraint just causes more observations to register. These observations are then generalized using the same approach as the base inference process—i.e., just as if the system had really registered these observations. This is particularly beneficial in cases of non-monotonicity. Consider again our example of an *m1* and *m2* with preconditions P and Q, respectively, and a constraint *Subdomain(m1,m2)*. If *m2* observes the exact values that led to the inference of P, then these values, combined with the ones that led to the inference of Q, may induce higher confidence for an invariant more specific than either P or Q, which will now be reported (because of crossing a confidence threshold).

***Caveats*:** Note that our approach (of propagating observations from one program point to another) does not strictly guarantee that the dynamically inferred invariants satisfy the second-order constraints. Interestingly, *if the inference process is monotonic, correctness is guaranteed*: under monotonic invariant inference, taking into account the union of two sets of observations should produce a condition that is weaker than either individual condition. This observation argues for why our approach is expected to be correct: as long as there are enough observations, dynamic invariant inference is typically monotonic, as discussed in Section 3.3.1.

Additionally, any implementation of dynamic invariant inference under second-order constraints suffers from the possibility of a disconnect between observed executions and values reflected in an invariant. The source of the problem is that we are allowing an invariant to be influenced by values not really seen at that program point during execution. These values will be reflected in a reported invariant but will not be reflected in other dependent invariants. For instance, using second-order constraints we may infer a more general precondition, but not the corresponding postcondition. Thus, readers who consider the two invariants together may misinterpret their meaning even when the invariants are individually correct.

3.4 Evaluation

There are two questions that our evaluation seeks to answer:

- Do second-order constraints aid the inference of better first-order invariants?
- Can correct second-order constraints be inferred dynamically?

The next two sections address these questions in order.

3.4.1 Impact of Second-Order Constraints

We explored the utility of second-order constraints in three case studies: one of a manageable, small example application with a relatively thorough test suite, and two of large, unfamiliar programs, with their actual test suites. In all studies, we first took existing classes and examined their APIs. We then added second-order constraints to manifest implicit relationships between API methods in the same class or in different classes. We ran a series of experiments to determine the effects that adding these constraints had on the observed pre- and postconditions reported by Daikon.

```
public class StackAr {
    public boolean isEmpty();
    public boolean isFull();
    public void makeEmpty();
    public void push(Object);
    public void pop();
    public Object top();
    public Object topAndPop();
}
```

Figure 3.2: The array-based stack StackAr, shipped with Daikon

In these three case studies, second-order constraints were written by hand and their correctness was verified by inspection. This is a feasible approach in several settings since second-order constraints are much sparser/coarse-grained than first-order invariants: one needs to write few second-order constraints to affect a large number of first-order invariants.

3.4.1.1 StackAr

Our first case study is `StackAr`, an array-based fixed-size stack implementation that ships with Daikon and is perhaps the most common Daikon benchmark and demonstration example. `StackAr` is interesting because it is the most controlled of our case studies (due to test suite coverage and small size).

Figure 3.2 lists the class and its methods. Methods `isEmpty` and `isFull` are straightforward. `makeEmpty` clears the stack. `push(o)` pushes element `o` to the top of the stack. `pop()` removes the top stack entry, but does not return a value. Instead, `top()` peeks at the top of the stack and returns the most recently pushed value; while `topAndPop()` returns the top of the stack before removing it. Operations `top` and `pop` raise an exception if the stack is empty, while `topAndPop` returns `null` in that case.

To explore our invariants, we examined this API and determined second-order constraints that we considered to be meaningful for this class. The process took one of the authors negligible time (less than a minute, although there was previous discussion of interesting invariants, hence the exact effort cost is unclear). We split these constraints

#	Classes	second-order constraints	First-order invariants			
			pre	post	*δ*pre	*δ*post
1	ArrayStack	*CanFollow(push, peek)*	1	3	-1	
2	ArrayStack	*Subrange(peek, get)*	0	5		-2
3	ListUtils	*Subdomain(removeAll, retainAll)*	12	0	-2	
4	ListUtils	*Subdomain(sum, subtract)*	0	0		
5	CollectionUtils	*Subdomain(subtract, disjunction)*	8	0	+2	
6	CollectionUtils	*Subdomain(subtract, intersection)*	14	0		
7	TreeBidiMap	*Subdomain(containsKey, get)*	101	0	+6-7	
8	TreeBidiMap	*Subdomain(nextKey, previousKey)*	108	0		
9	UnmodifiableSortedBidiMap	*Subrange(tailMap, subMap)*	0	16		
10	TreeBidiMap	*Subdomain(remove, get),* *Subdomain(get, remove)*	99	0	+2-7	
11	TreeBidiMap	*Subdomain(getKey, removeValue),* *Subdomain(removeValue, getKey)*	93	0	+2-7	
12	DualTreeBidiMap	*Subrange(tailMap, subMap)*	0	22		
13	UnboundedFifoBuffer	*Follows(add, remove)*	11	34		+1-1
14	SetUniqueList	*Follows(addAll, remove)*	3	8		-1
15	AbstractBidiMapDecorator	*Subrange(getKey, removeValue)*	0	10		+3-1
16	ReverseListIterator	*Subrange(previousIndex, nextIndex),* *Subrange(nextIndex, previousIndex)*	0	20		+5-1
17	CursorableLinkedList	*CanFollow(addNode, removeNode)*	3	6		
18	CursorableLinkedList	*CanFollow(addNode, updateNode)*	5	6		
19	LinkedMap	*Subdomain(get, remove),* *Subrange(get, remove)*	4	6	+1-1	+1-1
20	ListOrderedMap	*Subrange(put, remove)*	0	16		-3
21	CompositeMap	*CanFollow(addComposited, removeComposited)*	8	19	+6-3	
22	CompositeCollection	*Concord(addComposited(Collection),* *addComposited(Collection, Collection))*	12	28		+2-2

Figure 3.3: Summary of our experiments on the Apache Commons Collections.

into separate experiments and explored the effect they had on Daikon's invariant detection mechanism:

- Experiment 1 (**Ex1**): *Subdomain* on `topAndPop`, `top`, and `pop`. The operations `top`, `pop`, and `topAndPop` all require a nonempty stack, so we instructed the system to treat all of them as having identical subdomains.

- Experiment 2 (**Ex2**): Any *push* sets up the stack for a *top*, *pop*, or *topAndPop*. We experimented with setting up *CanFollow* relations between *push* and the three *top/pop* operations.

To ensure the highest-quality invariants, we ran these experiments with the DynComp tool [51] enabled.

Experiment 1: We instructed the system to treat all of `top`, `pop`, and `topAndPop` as having the same subdomains, using specifications such as

```
Subdomain(StackAr.topAndPop(), StackAr.pop())
Subdomain(StackAr.pop(), StackAr.top())
Subdomain(StackAr.top(), StackAr.topAndPop())
```

The above specification captures a circular subrange dependence, and hence equality: it specifies that all three operations should have effectively the same preconditions. There are other ways to express this equality: we experimented with reversing the circular dependencies and with establishing mutual *Subdomain* constraints between all three interesting pairs of the three operations (for a total of six constraints). We found these approaches to be equivalent.

The experiment results in the elimination of five spurious invariants from `pop`:

- '`this` has only one value'
- '`this.theArray` has only one value'
- '`size(this.theArray[]) == 100`'
- '`this.theArray[this.topOfStack] != null`'
- '`this.topOfStack < size(this.theArray[])-1`'

Also, "`this.topOfStack < size(this.theArray[])-1`" is replaced by the weaker (and correct) "`this.topOfStack <= size(this.theArray[])-1`". Our constraints further helped infer two new correct preconditions: one establishing an inequality between a stack's default capacity and the size of `this.theArray`, and one establishing that the top of the stack does not exceed the size of the internal array. As a result, the preconditions between the three methods were identical.

Experiment 2: For this experiment we specified that the `push` operation sets up a stack for using `top` and similar operations:

```
CanFollow(StackAr.push(Object), StackAr.top())
CanFollow(StackAr.push(Object), StackAr.pop())
CanFollow(StackAr.push(Object),
        StackAr.topAndPop())
```

This specification improved the inferred invariants, removing four incorrect preconditions (the same as for experiment #1, except for '`this` has only one value') and adding three invariants:

- One states that `topOfStack` cannot exceed the internal array size (as in experiment #1).
- One establishes an inequality between stack default capacity and array size (as in experiment #1).
- One establishes that the `topOfStack` is nonnegative.

Again the changes affected method '`pop`', while '`top`' and '`topAndPop`' remained unaffected, as they already had high test coverage.

In summary, the impact of second-order constraints on `StackAr`, under the standard test suite was overwhelmingly positive. All of the additional invariants were correct and most were insightful, while spurious invariants were eliminated.

3.4.1.2 Apache Commons Collections

Our second case study is the Apache Commons Collections library,[2] version 3.2.1. This library contains 356 classes, of which we used a total of 18 explicitly[3] for our experiments.

For our experiments, one of the authors (unfamiliar with the library) examined the above classes and constructed a specification (while consulting the API documentation) with a total of 27 second-order constraints, comprising 22 experiments. This process took approximately 3.5h. The author spent another 2h double-checking and fixing the invariants after specification. This amount of effort is in the noise level, compared to the development effort for a library of this size.

[2] `http://commons.apache.org/collections/`

[3] Some classes may use other classes from the library internally.

We then examined the utility of our specifications by running Daikon first with and then without our constraints. Since running the entire test suite with Daikon's instrumentation tool would have consumed harddisk space in excess of 10 GB (compressed), we instructed Daikon's instrumentation tool to only report invariants for the methods we were interested in. We only ran these experiments with DynComp disabled, as using the DynComp-instrumented JDK caused errors during execution.

We recorded the invariants of all affected methods and analyzed the generated differences. Figure 3.3 summarizes our results. We first list the affected class or classes, then the concrete second-order constraints we introduced (slightly compressed for space). Next, we considered the invariants inferred by Daikon, restricted to invariants of the particular methods occurring in our constraints. For example, in experiment #1, we considered methods `push` and `peek` in class `ArrayStack` only. The final four columns summarize these invariants: first, we give the number of invariants in the absence of any second-order annotations, separated into pre- and postconditions (**pre** and **post**). Finally we give the differences over any preconditions (δ**pre**) and postconditions we observed (δ**post**). We use the notation $+x - y$ to indicate that we added x new invariants and removed y existing ones.

Qualitatively, the use of second-order constraints on the Apache Commons Collections was a clear win. All 35 invariants removed were false, to the best of our understanding. We added 26 invariants, of which our manual inspection found 25 to be true (i.e., expected to hold for all executions, not just the ones observed) and 1 to be false. The added invariants arise due to non-monotonicity. In experiment 16, for example, the augmented observations in the presence of second-order constraints enable two additional true invariants (i.e., a stronger inference) in the precondition of method `nextIndex`: `this.list != null` and `this.iterator != null`. The invariant "`orig(value) != null`" was incorrectly added to `AbstractBidiMapDecorator.removeValue(Object)` in experiment 15, where parameter `value` does not have to be non-null. Furthermore, we replaced 5 invariants with more general invariants. Consider, for example, experiment 13, with second-order constraint `Follows(add,remove)`. The postcondition of method `UnboundedFifoBuffer.add(Object)` originally contained invariants such as "`this.head one of {0, 1, 2}`". Such false invariants are replaced with a more insightful "`this.head >= 0`".

3.4.1.3 AspectJ Compiler

#	Classes	second-order constraints	First-order invariants			
			pre	post	δpre	δpost
1	AjTypeSystem, AdviceImpl	*Subrange(getAjType, getDeclaringType)*	0	9		++2
2	ProgramElement	*Subdomain(toSignatureString(boolean), toLabelString(boolean))*	76	0		
3	ProgramElement	*Subdomain(toLabelString(boolean), toSignatureString(boolean))*	76	0		
4	ProgramElement	*Subrange(toLabelString(), toLabelString(boolean))*	0	142		
5	ProgramElement	*Subdomain(getParent, getChildren)*	76	0		
6	ProgramElement	*Subrange(genModifiers(), getModifiers(int))*	0	80		
7	ProgramElement	*Subdomain(toLabelString(), toSignatureString())*	81	0	+1-5	
8	ProgramElement	*Subdomain(toSignatureString(), toLabelString())*	81	0		
9	ProgramElement	*Subdomain(getChildren, getParent)*	76	0	-1	
10	FieldSignatureImpl	*Subdomain(getDeclaringTypeName, getDeclaringType)*	0	0		
11	FieldSignatureImpl	*Subdomain(getDeclaringType, getDeclaringTypeName)*	0	0		
12	MethodSignatureImpl	*Subdomain(getField, getFieldType)*	0	0		
13	MethodSignatureImpl	*Subdomain(toShortString, toLongString)*	0	0		
14	MethodSignatureImpl	*Subdomain(getDeclaringType, getDeclaringTypeName)*	0	0		
15	SignatureImpl	*Subdomain(getDeclaringTypeName, getDeclaringType)*	66	0		
16	SignatureImpl	*Subdomain(getDeclaringType, getDeclaringTypeName)*	66	0	+4-4	
17	SignatureImpl	*Subdomain(toLongString, toShortString)*	71	0	+3	
18	SignatureImpl	*Subdomain(toShortString, toLongString)*	71	0	+3-1	
19	BcelWeaver	*CanFollow(prepareForWeave, weave(UnwovenClassFile, BcelObjectType))*	52	80	+3-4	
20	BcelWeaver	*CanFollow(prepareForWeave, weave(UnwovenClassFile, BcelObjectType, bool.))*	41	80		
21	BcelWeaver	*CanFollow(prepareForWeave, weaveAndNotify)*	54	80		
22	BcelWeaver	*CanFollow(prepareForWeave, weaveNormalTypeMungers)*	53	80		
23	BcelWeaver	*CanFollow(prepareForWeave, weaveParentTypeMungers)*	53	80		
24	BcelWeaver	*CanFollow(prepareForWeave, weave(IClassFileProvider))*	52	80		
25	BcelWeaver	*CanFollow(prepareForWeave, weaveParentsFor)*	57	80		
26	BcelWeaver	*CanFollow(prepareForWeave, weaveWithoutDump)*	51	80		

Figure 3.4: Summary of our experiments on the AspectJ Compiler.

Our third case study is the AspectJ compiler. We followed the same approach as for the Apache Commons Collections, collecting invariants from unit tests and integration tests. Since AspectJ lacks detailed API documentation, one of the authors (unfamiliar with the library) directly inspected the source code of AspectJ and derived a total of 27 second-order constraints. The combined process of understanding the foreign code base and writing invariants cost the author approximately 10h. Again DynComp's instrumented JDK caused problems during execution, so we only tested with DynComp disabled.

We summarize our results in Figure 3.4. We removed 12 invariants. All of the 12 invariants were false. For instance, most "variable has only one value" and "variable is one of {...}" invariants (largely due to limitations in the test suite) were removed or replaced by more accurate invariants. We also added 1 false invariant in experiment 16 (again, due to non-monotonicity) by assuming a variable is always less than a constant. Meanwhile, we added 12 true invariants and replaced 3 invariants with more general ones. For instance, without the second-order constraint `Subrange(getAjType, getDeclaringType)` in experiment 1, Daikon reports no invariants for the exit of the `getDeclaringType` method. Our `Subrange` constraint yields two new postconditions, "`return != null`" and "`return.getClass() == AjTypeImpl.class`", due to observations on `getAjType`.

3.4.2 Inferring Second-Order Constraints

We next evaluate the success of our dynamic process of inferring second-order constraints. Note that dynamically inferred second-order constraints are useful for many reasons:

- As documentation of program behavior, on their own, i.e., as deeper invariants than typical Daikon invariants.

- For finding bugs in manually stated second-order constraints.

- For offering the programmer a set of mostly-correct second-order constraints to choose from.

Although the first benefit is very important, it is hard to quantify experimentally. Therefore, we focus on the second and third benefits, and specifically on evaluating the correctness of automatically inferred second-order constraints.

***Correctness of Inferred Constraints*:** We inspected the results of our automatic mechanism for inferring second-order constraints (Section 3.3.2) on four randomly selected classes from ACC and AspectJ (two each). One of the authors manually verified all of the generated second-order constraints. For this experiment, we considered a second-order constraint to be correct whenever it did not disagree with the implementation of the given class. Figure 3.5 lists our precision results. We note that overall precision exceeds 99%, suggesting that our confidence metric is highly effective at identifying low-quality constraint candidates.

Our five losses in `LstBuildConfigManager` were three *Follows* and two *Subrange* constraints involving methods with low-quality invariants. For example, four of the five constraints involved an `addListener` method for which Daikon had failed to observe the method's effect on the internal object state.

Although these constraints are true, they are not necessarily all interesting. Many of them just reflect implementation artifacts and would not arise if the methods in question had interesting invariants to begin with. For instance, for two methods that have a very small number of invariants in their preconditions, it is easy to find meaningless agreement—e.g., on the fact that their argument is never `null`.

For `SingletonMap`, we observed more than 800 proposed high-confidence constraints that claimed that most methods were in some relationship to each other. We found that `SingletonMap` is an immutable class, meaning that methods do not influence each other on subsequent calls—therefore many invariants remained the same across methods, resulting in second-order constraints being derived. (These include the two second-order constraints we manually wrote for `SingletonMap` but also many others of much less value.) This suggests the existence of other useful higher-order constraints beyond the catalog we have proposed herein; for `SingletonMap`, a meta-constraint *Immutable* would be the most concise way to express the properties that we observed.

					Second-Order Constraints #		
Library	Class	methods#	Program points #	Daikon variants #	total	correct	incorrect
ACC	AbstractMapBag	25	7	85	2	2	0
ACC	SingletonMap	34	54	635	806	806	0
AspectJ	Reflection	17	10	192	30	30	0
AspectJ	LstBuildConfigManager	18	23	778	112	107	5

Figure 3.5: Results of inferring second-order constraints. In the above, 'program points' lists only program points that our inference considers, i.e., `:::ENTER` and `:::EXIT` program points.

In summary, we found that dynamic second-order constraint inference is highly effective at identifying high-quality sets of second-order constraints, even though we allow some margin of error over already-erroneous or imprecise axioms.

***Inferred vs. Manual Constraints*:** Our manually derived second-order constraints of the previous section were never intended as a full or ideal set, but as a set of constraints that take low effort to produce and have significant effect over first-order invariants. Still, it is instructive to compare them with automatically inferred constraints. For this purpose, we ran our inference mechanism on all classes that we had manually written second-order constraints for.

Indeed, our manual effort to produce constraints for ACC and the AspectJ compiler originally yielded 64 constraints and not just the 52 shown in Figures 3.3 and 3.4. 12 manually derived second-order invariants were removed exactly because their absence from the set of automatically inferred constraints caused us to re-inspect them and discover they were erroneous! This pattern is likely to also occur in practice when dealing with unfamiliar code (in fact, 8 of the 12 were in AspectJ, which lacks documentation). Note that in a scenario in which software authors write their own second-order constraints, such disagreements between hand-written and inferred second-order constraints would point to more serious issues; likely to poor test suites or to implementation bugs.

Of the 52 constraints in Figures 3.3 and 3.4 our automatic inference mechanism produced 37 and missed 15. On closer inspection we found that 6 of those missing hand-written second-order constraints are rejected (although true) due to low success rate. This is caused by “noise” invariants participating in the corresponding preconditions and postconditions, due to very few actual observations for the relevant methods. 7 more second-order constraints are missing due to having no data samples at all for the methods. The last 2 second-order constraints are missing since our current implementation does not support detecting the *Concord* constraint or constraints relating methods in two different classes.

Thus, overall the automatic inference facility produces quite high-quality results on its own, and is found to be a strong complement for manual derivation of second-order constraints.

CHAPTER 4

SEDGE: SYMBOLIC EXAMPLE DATA GENERATION FOR DATAFLOW PROGRAMS

This chapter describes SEDGE: a dynamic symbolic execution engine for dataflow programs–indeed, the first such engine, to our knowledge. Dynamic symbolic execution is a systematic test case generation method. It statically collect path constraints while dynamically executing test cases generated by solving those path constraints using a constraint solver. We demonstrate that SEDGE beats the previous state-of-the-art in terms of test case coverage.

4.1 Introduction

Dataflow programming has emerged as an important data processing paradigm in the area of big data analytics. Dataflow programming consists of specifying a data processing program as a directed acyclic graph. Internal nodes of the graph represent operations on the data, for example, using relational algebra primitives such as filter, project, and join, or functional programming primitives such as “map” applications of user-defined local functions and “reduce” operations that collect values over sets of data. The edges in the graph represent data tables or files passed between operators (nodes) in the graph. Many recently proposed data processing languages and systems, such as Pig Latin [84], DryadLINQ [61], and Hyracks/ Asterix [6] resemble dataflow programming on datasets of enormous sizes. A user can develop dataflow programs by either writing the programs directly using the above languages or compiling queries written in declarative languages such as SQL and Hive [102].

When a user writes a dataflow program, he/she will typically employ example data or test cases to validate it. Validating with large real data is impractical, both for reasons of efficiency (running on large data sets takes a long time) and for reasons of ease-of-validation (it is hard to tell whether the result is what was expected). One alternative is to sample the real data available. The sample data need to thoroughly exercise the program, covering all key behavior of each dataflow operator. This is very hard to achieve via random sampling, however. For instance, equi-joining two sample data tables of small size is likely to produce an empty result, if the values being joined are distributed arbitrarily.

Another alternative is to synthesize representative data. Such data synthesis is complicated by the complexity of dataflow language operators as well as by the presence of user-defined functions. Current state-of-the-art in example data generation for dataflow programs [84] is of limited help. Such techniques can generate high-coverage data for dataflow programs with simple constraints. However, for dataflow programs with complex constraints, e.g., with numerous filters, arithmetic operations, and user-defined functions, the generated data are incomplete due to shortcomings in constraint searching and solving strategies.

In this chapter, we address the problem of efficient example data generation for complex dataflow programs by bringing *symbolic reasoning* to bear on the process of sample data generation. We present the first technique and system for systematically generating representative example data using dynamic symbolic execution (DSE) [18, 46, 104] of dataflow programs. Our concrete setting is the popular Pig Latin language [85]. Our DSE technique analyzes the program while executing it using sampled data, determines whether the sampled input data are complete (i.e., achieve full coverage), and, if not, attempts to synthesize input tuples that result in the joint sampled and synthesized data being a complete example data set for the program.

We have implemented this approach in SEDGE, short for *Symbolic Example Data GEneration*. SEDGE is a reimplementation of the example generation part in the Apache Pig dataflow system, which currently implements the closest comparable past research, by Olston et al. [84].

Illustration

For a simple demonstration, consider an application scenario in computational astrophysics. We surveyed 11 queries in the Sloan Digital Sky Survey[1] for analyzing star and galaxy observations, and rewrote them using the Pig Latin language. The most complex query contains 34 filters and 2 joins. For ease of exposition, we show a simple example query in Listing 4.1 by combining features from two actual queries and will use it as a running example in the chapter. (For more details on the real queries, see the evaluation section.)

```
A = LOAD 'fileA' using PigStorage()
  AS (name:chararray, value:int);
B = LOAD 'fileB' using PigStorage()
  AS (u:double, class:int);
C = FILTER A BY value < 100 AND value >= 0;
D = FILTER B BY math.POW(u,2.0) > 0.25;
E = JOIN C ON value, D ON class;
```

Listing 4.1: An example Pig Latin program

The program begins by loading tables `A` and `B` from files containing measurements. Both kinds of measurements need to be filtered. The first filter keeps only measurements in a certain value range and the second filter removes low u values. The tuples that survive the filtering get joined.

Imagine that we execute the program for a small number of sampled input tuples from `fileA` and `fileB`. If we want to achieve perfect coverage on random sampling of actual data alone, we are unlikely to be successful if the sample is small. The data from the two tables need to pass filters and (even more unlikely) have their `value` and `class` fields coincide. This is a case where targeted test data generation can help.

[1] http://skyserver.sdss.org/public/en/help/docs/realquery.asp

Past techniques for example data generation cannot handle this example well. Olston et al.'s technique [84] will synthesize data by considering operators one-by-one in reverse order in the Pig Latin program. It will attempt to create data to satisfy the `JOIN` first, without concern for the `FILTER` conditions that the same data have to satisfy. This will likely fail to satisfy even the first `FILTER` operator: the range 0-99 will have to be hit purely by chance. The problem for the Olston technique is that `value` is not a free variable once the `JOIN` constraint is satisfied: it is limited to the values that the system arbitrarily chose in order to have the `JOIN` operator produce output.

Even more importantly, the second `FILTER` operator is hard to process. It contains a user-defined function, `math.POW`. Although this function is simple, it will still befuddle an automatic test data generation system. Furthermore, an essential part of dataflow programming is the ability to use user-defined functions freely, however complex these functions may be. The large volume of work on automatic data generation in other settings (e.g., SQL databases [9]) does not address user-defined functions.

Our approach overcomes such problems by modeling the entire program in a powerful reasoning engine, handling complex conditions, and dealing with user-defined functions with the aid of concrete values observed over sample data. We process the program using a domain with (symbolic) variables, such as *value*, *class*, etc. A symbolic variable "*columnname*" represents the value of one column of an input table for a set of tuples. We start with a concrete execution of the program using small samples of real input data. During such concrete execution we observe, first, which program cases are covered, and, second, what are the values of user-defined functions for real data. E.g., a tuple $(3.3, 32)$ of table B will register the value pair $(u : 3.3, \mathit{math.POW(u,2.0)} : 10.89)$ for the user-defined function. This value will later help when trying to solve symbolic constraints.

After the concrete execution, our approach uses symbolic reasoning to cover program cases that were not already covered by the concrete execution. The approach performs a symbolic execution of the program, gathering constraints along each path to the sources.

We use the Z3 SMT solver [29] to solve the constraints. Concrete values for user-defined functions are supplied to the solver. That is, the user-defined function is treated as a black-box and the solver is supplied extra constraints of the form $u = 3.3 \Rightarrow$ *math.POW(*$u, 2.0) = 10.89$. These can aid the solver in producing satisfying assignments. Essentially, we try to make an educated guess: Whenever we do not know how to generate example data for a constraint that depends on a user defined function, we can always simplify this constraint by replacing the symbolic representation of the user-defined function with concrete values.

Contributions

In brief, the contributions of our work are as follows:

- We detail a translation of dataflow operators into symbolic constraints. These constraints are subsequently solved using an SMT solver.

- We adapt the technique of dynamic-symbolic execution to the domain of dataflow languages. By doing so, we exploit the unique features of this domain, thus enabling high coverage. Specifically, we exploit the absence of side-effects in order to perform a multiple-path analysis: observations on the values of a user-defined function on different execution paths can help solve constraints involving the user-defined function.

- As a result of the above, we produce an example data generation technique that achieves higher coverage than past literature, managing to produce data that exercise all operators of all programs that we examined. We show extensive measurements to confirm our approach's advantage. Our technique achieves full coverage in all benchmark programs with a boost in performance for most benchmarks.

4.2 Background and Context

We next discuss some pertinent background on dataflow programming as well as on concepts and mechanisms introduced in closely related past work.

4.2.1 Dataflow Program

A dataflow program is a directed bipartite graph, separating computations (i.e., operators) in one partition and computational intermediate results (i.e., data tables) in the other partition. In other words, it is a graph in which data tables *flow into* operators, and operators flow into data tables. A data table is a collection of tuples with possible duplicates. A tuple is typically a sequence of atomic values (integer, long, float, chararray, etc.) or complex types (tuple, bag, map). An operator usually has some input tables and one output table. We say that a data table is an input table of an operator in a dataflow program if the data table flows into the operator. Similarly, we say that a data table is an output table of an operator in a dataflow program if the operator flows into the data table. If operator `A`'s output table is one of operator `B`'s input tables, `A` is said to be an *upstream neighbor* of `B` and `B` is said to be a *downstream neighbor* of `A`. An operator without any upstream neighbor is called a *leaf* operator, and an operator without any downstream neighbor is called a *root* operator—the root operator generates the final output.

4.2.2 Pig Latin

Pig Latin is a well-known dataflow programming language and the language front-end of the Apache Pig infrastructure for analyzing large data sets. The Pig compiler translates Pig Latin programs into sequences of map-reduce programs for Hadoop. A Pig Latin program is a sequence of statements where each statement represents a data transformation. In a Pig Latin statement, an operator processes a set of input tables and produces an output table. Following are the core operators of Pig Latin [85].

1. LOAD: Read the contents of input data files.

2. FILTER: Discard data that do not satisfy a built-in logic predicate or a user-defined boolean function.

3. COGROUP: Divide one or more sets (more accurately "bags", but we use the term "set" informally) of input tuples into different groups according to some specification. Each resultant output tuple consists of a group identifier and a nested table containing a set of input tuples satisfying the specification.

4. GROUP: A special case of COGROUP when only one set of input tuples is involved.

5. TRANSFORM: Apply a transformation function to input tuples. Transformation functions include projection, built-in arithmetic functions (e.g., incrementing a numeric value), user-defined functions, and aggregation. An aggregation is implemented by first invoking COGROUP or GROUP, and then doing transformation group by group. For example, $\mathit{Average}\,(\cdot)$ is an aggregation that averages the values in each group of input tuples.

6. JOIN: Equijoin tuples from two input tables.

7. UNION: Vertically glue together the contents of two input tables into one output table.

8. FOREACH: Apply some processing to every tuple of the input data set. FOREACH is often followed by a GENERATE clause to pick a subset of all available fields.

9. DISTINCT: Remove duplicate tuples from the input data set.

10. SPLIT: Split out the input data set into two or more output data sets. A condition argument determines the partition that each tuple of the input data goes into.

11. STORE: Write the output data set to a file.

4.2.3 Equivalence Class Model

Our work tries to maximize branch coverage in Pig Latin programs. An interesting question is what constitutes full coverage of a Pig Latin operator. In some cases the answer is clear: the `FILTER` operator, for instance, is well-covered when its input contains tuples that satisfy the filter condition and tuples that fail the filter condition. In other cases, the definition of coverage is not as simple. For instance, do we consider a `UNION` operator sufficiently covered if all its output tuples come from a single input table (i.e., if one of its input tables is empty)? The choice is arbitrary but the more reasonable option seems to be to require that both inputs of a `UNION` operator be non-empty. Furthermore, whether an operator is covered may be more convenient to discern in some cases by observing its input and in others by observing its output. For instance, a `JOIN` is well-covered when its output is non-empty, while a `UNION` is well-covered when its inputs are both non-empty.

To specify the coverage of operators we inherit the definition of *equivalence classes* from Olston et al. [84]—the research work that has formed the basis of the example generation functionality in Apache Pig. Each Pig Latin operator yields a set of equivalence classes for *either its input or its output tuples*. Equivalence classes partition the actual set of tuples—each tuple can belong to at most one equivalence class per operator. To generate example data with 100% coverage, the input or output table of each operator (when the program is evaluated with the example data) must contain at least one tuple belonging to each of the operator's equivalence classes.

We summarize the equivalence class definitions for the operators of Pig Latin below. The definitions are from Olston et al.'s publication [84] and the implementation in Apache Pig.[2]

- LOAD/STORE/FOREACH/TRANSFORM: Every input tuple is assigned to the same class E_1. (I.e., the operator is always covered, as long as its input is non-empty.)

[2] `http://pig.apache.org/`

- FILTER: Every input tuple that passes the filter is assigned to a class E_1; all others are assigned to a class E_2. (The intention is to show at least one record that passes the filter, and one that does not pass.)

- GROUP/COGROUP: Every output tuple is assigned to the same class E_1. For every output tuple, the nested table for every group identifier must contain at least two tuples. (The purpose of E_1 is to illustrate a case where multiple input records are combined into a single output record.)

- JOIN: Every output tuple is assigned to the same class E_1. (The intention is to illustrate a case of two input records being joined.)

- UNION: Every input tuple from one input table is assigned to E_1, tuples from the other input table are assigned to E_2. (The aim is to show at least one record from each input table being placed into the unioned output.)

- DISTINCT: Every input tuple is assigned to the same class E_1. For at least one input tuple to DISTINCT, there must be a duplicate, to show that at least one duplicate record is removed.

- SPLIT: Every input tuple that passes condition i is assigned to class E_{i1}; input tuples that do not pass i are assigned to a class E_{i2}. The number of equivalence classes of a SPLIT depends on how many conditions the SPLIT has. If a SPLIT has n conditions, it yields $2n$ equivalence classes. (The aim is to show, for each split condition, at least one record that passes the condition, and one that does not pass.)

4.2.4 Quantitative Objectives

We use two metrics to describe the quality of example data and follow earlier terminology [84]:

1. Completeness: The average of per-operator completeness values. The completeness of an operator is the fraction of the equivalence classes of the operator for which at

least one example tuple exists. An ideal algorithm should find example data for every equivalence class of every operator in a Pig Latin program.

2. Conciseness: The average of per-operator conciseness values. The conciseness of an operator is the ratio of the number of operator equivalence classes to the total number of different example tuples for the operator (with a ceiling of 1). An ideal algorithm should use as few example tuples as possible to illustrate the semantics of an operator.

The completeness metric is clearly a metric of coverage, as defined earlier. Specifically, it corresponds to *branch coverage* in the program analysis and software engineering literature. Branch coverage counts the percentage of control-flow branches that get tested.

4.3 SEDGE Design

Our system, SEDGE, uses a three-step algorithm to generate example data in Pig Latin programs.

(1) Downstream Propagation: execute programs using sampled real data, record values of user-defined functions—see Section 4.3.2;

(2) Pruning Pass: eliminate redundant data so that each covered equivalence class only contains a single member;

(3) Upstream Pass: generate constraints and synthesize data for equivalence classes that the sampled test data do not explore by performing DSE.

The last pass (upstream pass) is the key new element of our approach and is described next.

4.3.1 Constraint Generation

The essence of our approach is to represent equivalence classes symbolically and to produce symbolic constraints that describe the data tuples that belong in each equivalence class. Solving the constraints (i.e., producing data that satisfy them) yields our test inputs. Our constraint generator steps through the dataflow graph to compute all equivalence classes for each Pig Latin operation, starting at root (i.e., final) operators. We assume that each root operator is of the form STORE W, without loss of generality (the analysis enters dummy nodes of this form when they are implicit). Similarly we assume that all variable names in our program are unique.

Pig Latin code	**Equivalence class constraints**	**Cardinality constraints**
`STORE A`	$\mathcal{C}(A) \supseteq \{P\}$, where $\forall t : P(t)$	$\#\mathcal{T}(P) \geq 1$
`A = FILTER B BY Q`	$\mathcal{C}(B) \supseteq \{P_\neg\}$, where $P_\neg(t) \equiv \neg[Q]_b(t)$. for all $P \in \mathcal{C}(A) : \mathcal{C}(B) \supseteq \{P'\}$ where $P'(t) \equiv P(t) \wedge [Q]_b(t)$	$\#\mathcal{T}(P'_\neg) \geq 1$ $\#\mathcal{T}(P') \geq \#\mathcal{T}(P)$
`A = UNION B, C`	for all $P \in \mathcal{C}(A) : \mathcal{C}(B) \supseteq \{P\}$ and $\mathcal{C}(C) \supseteq \{P\}$	
`A = JOIN B BY x, C BY y`	for all $P \in \mathcal{C}(A) : \exists A_f.\mathcal{C}(B) \supseteq \{P'_x\}$ and $\mathcal{C}(C) \supseteq \{P'_y\}$ where $P'_x(t) \equiv P(t) \wedge t.x = A_f$ and $P'_y(t) \equiv P(t) \wedge t.y = A_f$	$\#\mathcal{T}(P'_x) \geq \#\mathcal{T}(P)$ $\#\mathcal{T}(P'_y) \geq \#\mathcal{T}(P)$
`A = DISTINCT B`	for all $P \in \mathcal{C}(A) : \exists A_t.\mathcal{C}(B) \supseteq \{P'\}$ where $P'(t) \equiv P(t) \wedge t \approx A_t$	$\#\mathcal{T}(P') \geq 1 + \#\mathcal{T}(P)$
`A = GROUP B BY x`	for all $P \in \mathcal{C}(A) : \exists A_f.\mathcal{C}(B) \supseteq \{P'\}$ where $P'(t) \equiv P(t) \wedge t.x = A_f$	$\#\mathcal{T}(P') \geq 1 + \#\mathcal{T}(P)$

Figure 4.1: Summary of representative translations from Pig Latin statements into equivalence classes, manifested as constraints. The above constraints are all *binding* constraints, except for the terminating $P_\neg$ in FILTER, and for P' in DISTINCT which is both terminating and binding. In the above, $[\cdot]_b$ translates boolean Pig expressions into our term language, and $\mathcal{T}(P)$ is the set of sample tuples for constraint P. Every rule introduces fresh symbolic names for equivalence classes, we use fresh variables A_f to refer to individual values, and A_t to refer to tuples.

We represent the set of constraints (one for each equivalence class) of a statement V $= \ldots$ as $\mathcal{C}(V)$. We consider two kinds of equivalence classes: *terminating* equivalence classes, which represent paths of tuples that end at a given operator (e.g., filtered out), and *binding* equivalence classes, which represent paths through which tuples continue downstream.

For illustration, consider our running example, reproduced here for ease of reference.

```
A = LOAD 'fileA' using PigStorage()
  AS (name:chararray, value:int);
B = LOAD 'fileB' using PigStorage()
  AS (u:double, class:int);
C = FILTER A BY value < 100 AND value >= 0;
D = FILTER B BY math.POW(u,2.0) > 0.25;
```

```
E = JOIN C ON value, D ON class;
```

Here, our root node consumes variable E. Our analysis considers E as if it were flowing upstream from a `STORE` operation. We invent a symbolic name, P, for the (single) constraint induced by the `STORE`. Its constraint is satisfied by all tuples:

$$\mathcal{C}(E) \supseteq \{P\}, \text{where } \forall t : P(t)$$

Note the use of $\supseteq$. A dataflow node could receive constraints from several downstream neighbors so our constraint inference is using subset reasoning: we know that $\mathcal{C}(E)$ includes at least P, but it could include other constraints as well. (In this example it does not.)

$\mathcal{C}(E)$ is then propagated to the `JOIN` statement that constructs `E`. Throughout this section, given a statement Q and its downstream neighbor with constraints P, P' denotes the refined constraints by conjoining P and the new constraints needed to flow a tuple upstream out of the downstream neighbor. `JOIN`s require tuples to agree on particular fields (`value` and `class`, here), so we enforce this property by encoding it via P' in our constraints:

$$\begin{aligned}
\text{for all}\quad & P \in \mathcal{C}(E) : \exists e_x. \\
& \mathcal{C}(C) \supseteq \{P'_{\texttt{value}}\} \\
& \mathcal{C}(D) \supseteq \{P'_{\texttt{class}}\} \\
& \text{where}\quad P'_{\texttt{value}}(t) \equiv P(t) \wedge t.\texttt{value} = e_x \\
& \text{and}\qquad P'_{\texttt{class}}(t) \equiv P(t) \wedge t.\texttt{class} = e_x
\end{aligned}$$

("for all $P \in \mathcal{C}(E) : \mathcal{CS}$" here means that we iterate over all P in $\mathcal{C}(E)$ and generate constraints $\mathcal{CS}$ for each such P.)

Continuing the propagation process, we pass the above constraints on to the `FILTER` operators of our example. For instance, consider the statement `D = FILTER...`, which eliminates all elements for which `math.POW(u,2.0)` $>$ 0.25 does not hold. This statement first introduces a binding equivalence class for the constraints flowing upstream via

$\mathcal{C}(D)$. The statement also introduces a single terminating equivalence class ($P_\neg$) to capture the case of tuples that do not pass the filter:

$$
\begin{aligned}
&\mathcal{C}(B) \supseteq \{P_\neg\}\\
&\quad \text{where } P_\neg(t) \equiv \neg(\texttt{math.POW(}t.\text{u}\texttt{,2.0)} > 0.25)\\
&\text{and for all } P \in \mathcal{C}(D)\\
&\quad \mathcal{C}(B) \supseteq \{P'\}\\
&\quad \text{where } P'(t) \equiv P(t) \wedge \texttt{math.POW(}t.\text{u}\texttt{,2.0)} > 0.25
\end{aligned}
$$

In our representation, we have preserved the user-defined function `math.POW` as an example of a function that the theorem prover cannot handle directly (see Section 4.3.2).

We handle the other `FILTER` statement similarly and reach the `LOAD` statement which completes the analysis. The resulting $\mathcal{C}$ sets contain symbolic names for all equivalence classes and our symbolic constraints can be used to define members of these classes.

Figure 4.1 gives the general form of our reasoning for representative constructs (also including `DISTINCT` statements and cardinality constraints, discussed below). For all operators for which our first two analysis passes observed insufficient coverage, we collect constraints using the above scheme to generate the constraints P that represent each insufficiently covered equivalence class. For each P we attempt to add elements to its corresponding set of samples $\mathcal{T}(P)$. We synthesize such tuples t as follows:

1. Pass P to the theorem prover and query for witnesses for the existentially qualified fields. If there are no witnesses, abort; either the equivalence class is empty/not satisfiable due to conflicting requirements, or the theorem prover lacks the power to synthesize a representative tuple.

2. Otherwise, extract the witnesses into tuple t'.

3. For any field f required by the type constraints over t in P, extract t_2.f from randomly chosen t_2 from our observed samples. Combine t' with all the t_2.f into t''.

4. For any still-missing fields (i.e., if no matching t_2 exists), fill the field with randomly synthesized data, yielding t.

5. If $t \in \mathcal{T}(P)$ already, repeat the previous two steps as needed, otherwise insert t into $\mathcal{T}(P)$.

As the last steps (and Table 4.1) show, there is another dimension in our sample generation, namely generating the right amount of sample data. Specifically, recall that our binding equivalence class for

```
F = GROUP B BY x
```

requires at least two tuples. To capture this constraint, we permit constraints on the cardinality of our sets of witness tuples, notation $\#\mathcal{T}(P') \geq 2$, where predicate P' represents the binding equivalence class in the above. All such constraints are greater-than-or-equal constraints, and we always pick the minimum cardinality that satisfies all constraints.

Another subtlety of our constraint notation comes from the `DISTINCT` statement, as in

```
G = DISTINCT B
```

This statement eliminates duplicate tuples. Since set semantics have no notion of duplicates, we extend all of our tuples with a unique *identity* field that does not occur in the Pig program. We write $t_1 \approx t_2$ iff the tuples t_1 and t_2 have the same fields, ignoring the *identity* field.

To support aggregation operations in sample synthesis, we further permit reasoning about our sampled tuples. For example, Pig Latin allows us to write

```
A = LOAD ...
sum = SUM(A.x)
B = FILTER A BY count == sum
```

We translate aggregations such as `sum = SUM(A.x)` into aggregations over our sets of samples. Whenever we synthesize samples for one of `A`'s binding equivalence classes, e.g., represented by P, we simply set $\texttt{sum} = \sum_{t \in \mathcal{T}(P)} t.\texttt{x}$. The translation is analogous for other aggregators (AVG, MAX, etc.). Aggregators enforce $\#\mathcal{T}(P) \geq 2$.

4.3.2 User-defined Function Concretization

In earlier sections, we classified our approach as *dynamic-symbolic*, following other similar work in different settings [18, 46, 93, 104]. The important aspect of a dynamic-symbolic execution approach to test generation is that dynamic (i.e., concrete) observations are used to help the symbolic solving process. The foremost aspect where this benefit is apparent in our setting is when dealing with user-defined functions (UDFs). A user-defined function is any side-effect-free operator that has a definition external to the language. In the Pig Latin world, this typically means a Java function used to process values, e.g., in a FILTER. What a dynamic-symbolic execution engine can do is to treat a UDF as a black-box function. Inside a constraint, a use of a UDF is replaced by a set of function values from the concrete semantics, under the assumption that some invocations of UDFs (and return values thereof) have already been observed.

Consider the example Pig Latin program shown in Listing 4.2. Our objective is to generate complete example data with one tuple passing and one tuple not passing the FILTER. This program's key step is the application of the UDF HASH to perform filtering, which takes an integer as argument and returns its hash value.

```
A = LOAD 'fileA' using PigStorage()
  AS (x:int, y:int);
B = FILTER A BY x == HASH(y) AND x > 50;
```

Listing 4.2: Example Pig Latin program calling user-defined function HASH.

A simplified implementation of HASH is shown in Figure 4.2. In this implementation, HASH extends the EvalFunc class (which is required by Pig Latin to construct Java user-defined functions[3]).

[3] See Pig's implementation guide for user-defined functions at http://pig.apache.org/docs/r0.9.2/udf.html

```
public class HASH extends EvalFunc<Integer> {
  public Integer exec(Tuple input) {
    if (input == null || input.size() == 0)
      return null;
    Integer y = (Integer) input.get(0);
    int hash = y*(y+3);
    return hash % 60;
  } // ..
}
```

Figure 4.2: The implementation of function `HASH`.

Assume that we run the program with two input tuples `(33, 42)` and `(47, 19)` that do not pass the `FILTER` (since 33 and 47 are not the hash values of 42 and 19, respectively). On these two executions, we obtain two evaluations of `HASH`: `y = 42, HASH(y) = 30` and `y = 19, HASH(y) = 58`. For our technique to have 100% completeness, we need to generate example data for `A(x,y)` such that `x == HASH(y) && x > 50`. Using the two evaluations of `HASH`, we construct two simplified versions of the constraint: `x == 30 && y == 42 && x > 50` and `x == 58 && y == 19 && x > 50`. In the simplified constraints the function call `HASH(y)` has been *concretized* to the observed values (30 and 58, respectively). The second simplified constraint is satisfiable while the first is not. Using the satisfying assignment, we derive a new example input `(58, 19)` for `A(x,y)`.

Thus, our approach *records* concrete values for UDFs during the downstream pass, *concretizes* constraints using recorded concrete data, and *solves* them via automatic constraint solvers, in the upstream pass. We use uninterpreted functions to encode a concretized constraint. An uninterpreted function (UF) [15, 16] is a black box with no semantic assumptions other than the obligation that it behave functionally: equal parameters yield equal function values. To encode UDFs as uninterpreted functions for our constraint solver, we supply concrete observations as implications, using the if-then-else operator (*ite*) over boolean formulas and concrete values.

Consider the example of Listing 4.2 again, in which we need to find assignments to `(x,y)` to satisfy the constraint `x == HASH(y) && x > 50`. We supply the constraint solver Z3 with concrete observations on the `HASH` UDF via the following commands:

```
(declare-const x Int)
(declare-const y Int)
(define-fun HASH ((x!1 Int)) Int
    (ite (= x!1 42) 30
    (ite (= x!1 19) 58
     0)))
(assert (not (= (HASH y) 0)))
(assert (= (HASH y) x))
(assert (> x 50))
```

The first two `declare-const` commands declare two integer variables. The `define-fun` command creates a UF that takes a parameter representing an integer and returns a constant value. `x!1` is the argument of the UF. We have observed two invocations of the function `HASH`, `HASH` applied to `y == 42` yields 30, and `HASH` applied to `y == 19` yields 58. To complete the definition of the UF, we need to relate unknown parameter values with a default return value, which in this case we arbitrarily choose to be zero. Still, we assert that `HASH(y)` is not zero to avoid accidental satisfaction. Finally we provide the constraint `x == HASH(y) && x > 50` that we want to solve. Using three `assert` commands, the system pushes three formulas into Z3's internal constraint stack. We solve the concretized constraints by asking Z3 to produce a satisfying assignment for variables in the constraints.

Of course, when the observations of the UDF are not sufficient to obtain the desired coverage, Z3 will deem a concretized constraint to be unsatisfiable or unknown. To increase the chance of finding a satisfying assignment for an abstract constraint, we also try a second constraint solver, CORAL [14], when Z3 returns unsatisfiable or unknown for a concretized constraint. The distinction between Z3 and CORAL concerns the kind of formulas that they can solve: Z3 can derive models and check satisfiability of formulas in decidable theories, while CORAL can deal with numerical constraints involving undecidable theories. As a consequence of supporting undecidable theories, CORAL can solve constraints involving UDFs in the form of common math functions (e.g., power function) directly without concretization. If neither concretization-and-employing-Z3 nor calling CORAL can solve a constraint, SEDGE will be unable to obtain perfect coverage.

Note that our approach to solving UDFs reasons about all observed values of the UDF in parallel. These UDF observations may be produced in different paths through the program, including executions of different test cases. Still, the observations can be used together (i.e., we can assume that all of them hold) because of the lack of side-effects in a dataflow program. In contrast, in dynamic-symbolic execution of an imperative program, only values (of user-defined functions) observed during the current dynamic execution can be leveraged at a given constraint solving point.

4.4 Implementation

The SEDGE system has required non-trivial implementation effort, in the support of different data types, the interfacing with the Z3 constraint solver, and the integration of string generation capabilities.

4.4.1 Symbolic Representation of Values

SEDGE maintains an intermediate level of abstract syntax trees for communication between Z3 and Pig Latin constraints. Each node of the tree denotes a symbolic variable occurring in the Pig Latin constraints. The high-level idea is that SEDGE maps an execution path to a conjunction of arithmetic or string constraints over symbolic variables and constants. Each symbolic variable has a name and a data type, such as int and long, mapping to a field of a table in a Pig Latin script with the same name and data type. SEDGE then invokes Z3 to find a solution to that constraint system. If the constraint solver finds a solution, SEDGE maps it back to input tuples (tuples from LOAD). SEDGE supports mapping all Pig Latin data types into a symbolic variable, with support for overflow and underflow checked arithmetic.

1) int: Integers are represented as 32-bit signed bit-vectors, since the Z3 constraint solver has better support for bit-vector arithmetic than for integer arithmetic. Arithmetic calculations over integers are thus simulated with arithmetic calculations over 32-bit-vectors. The simulation is accurate and takes into account the Java (Apache Pig is written in Java) representation of values of type int as 32-bit-vectors. Additional constraints are created to check that the bit-wise computation does not overflow and underflow. Standard library conversion functions (e.g., *java.lang.Long.parseLong(String)*) are used to translate back from 32-bit-vectors into integers.

2) long: Similar to *int*, long integers are represented by 64-bit signed bit-vectors, since Z3 has no builtin support for long integers.

3) float/double: Floating point numbers are represented by real numbers in the form of fractions of long integers. No current constraint solvers have good support for floating-point arithmetic. Calculations with floating point numbers are thus approximated by real-valued calculations. A real number in the form of fractions of long integers can be translated to a floating point number by first representing the fraction using *BigFraction* from *Apache Common Math Library*,[4] and invoking *BigFraction.floatValue()* (or *BigFraction.doubleValue()*) to get the fraction as a float (or double, respectively).

4) chararray: A character array is represented by *java.lang.String*, which is also the inner representation of a character array in Pig Latin.

5) bytearray: We do not support byte arrays directly. We try to identify the type that the byte array can convert to at runtime and cast it.

6) boolean: A boolean variable is represented by an integer with 3 values: -1 for *FALSE*; 0 for *UNDEF*; 1 for *TRUE*.

[4] http://commons.apache.org/math/

Arithmetic and string constraints are typically expressed over fields of simple types as listed above. Therefore, we do not define complex types (tuple, bag, map) for symbolic variables.

4.4.2 Arithmetic and String Constraint Solving

As mentioned earlier, SEDGE uses Z3 [29] to solve arithmetic constraints. Since Z3 provides a C interface and SEDGE is implemented using Java, to have access to Z3's C API from Java, we employ SWIG.[5] We wrap Z3's C API using Java proxy classes and generate JNI [72] wrapper code automatically. A problem in wrapping C programs for Java is that values may be returned in function parameters in C, but in Java values are typically returned in the return value of a function. SEDGE uses typemaps in SWIG, a code generation rule that is attached to a specific C data type, to overcome the problem. Given the data type *D* of a value returned in function parameters, SEDGE constructs a structure *S* containing a member variable of type *D*. It also registers a typemap such that 1) any occurrence of a function parameter of type *D* in a function call in Z3 is converted into *S*, 2) the return parameter *S*'s value can be read after returning from the function in Java.

For string constraints, the main new element of our implementation concerns reasoning about string constraints containing regular expressions. Our approach is based on Xeger[6] a Java library for generating a sample string for a regular expression. Xeger builds a deterministic finite automaton (DFA) for a string constraint in the form of a regular expression, and follows the edges of the DFA probabilistically, until it arrives at an accepting state of the DFA. Xeger is suboptimal for two reasons: first, it may keep visiting the same state until a "stack overflow" error happens; second, it does not support union, concatenated repetition, intersection, concatenation, or complement of regular expressions. To avoid the "stack overflow" error, our approach keeps a map from state ID to the number of times a

[5] `http://www.swig.org/`

[6] `http://code.google.com/p/xeger/`

state has been entered and reduces the probability of re-entering that state proportionally. To support operations such as union and intersection, we add an intermediate step between building the DFA and following DFA edges to return a new deterministic automaton for the appropriate regular expression. For example, when generating a satisfying assignment that maps a string to a value so that the constraint matches '.*apache.*' and '.*commons.*', we intersect the automata representing the strings.

4.5 Evaluation

In this section, we evaluate the implementation of SEDGE by running a wide spectrum of actual Pig Latin programs. We measured both the completeness of generated example tuples and the run-time of example generation for SEDGE and for the current state-of-the-art: the original Pig example data generator (abbreviated to "Olston's system" in our discussion). Compared to Olston's system, our experiments confirm that SEDGE achieves higher completeness. In most experiments, SEDGE also incurs a lower running time.

4.5.1 Benchmark Programs

To evaluate our system, we applied it to two benchmark suites:

- We use the entirety of the *PigMix* benchmark suite, consisting of 20 Pig programs designed to model practical Pig problems.
- We use eleven sample SQL queries (the first ten in the list and an 11-th selected for being complex) from the Sloan Digital Sky Survey (SDSS) set[7] and hand-translated them directly into Pig code. The complex query contains 34 `FILTER` operations and 2 `JOIN` operations.

The PigMix benchmark provides Pig Latin programs for testing a set of features such as "data with many fields, but only a few are used" and "merge join". The SDSS sample

[7] `http://skyserver.sdss.org/public/en/help/docs/realquery.asp`

queries typically search for an astronomical object based on some criteria. For example, Program 5 of the SDSS set is below:

```
A = LOAD Galaxy3 using PigStorage()
  AS (colc_g : float, colc_r : float,
      cx : float, cy : float);
B = FILTER A BY
    (-0.642788 * cx + 0.766044 * cy >=0.0)
  AND (-0.984808 * cx - 0.173648 * cy <0.0);
```

Listing 4.3: SDSS program 5

The program finds galaxies in a given area of the sky, using a coordinate cut in the unit vector cx, cy, cz. As can be seen, these benchmark programs are typically short, with only a handful of them exhibiting interesting complexity.

For each query set, we used the input data that accompany the relevant queries. PigMix ships with a tuple synthesizer that generates such data. The SDSS benchmark suite is designed for the digital sky survey data from the SDSS data release 7. We selected a random sampling of tuples from the database of this benchmark, in which the total amount of data is 818 GB, and the total number of rows exceeds 3.4 billion.

4.5.2 Methodology and Setup

Since the importance of tuple synthesis varies not only by benchmark but also by the size of the tuples supplied to the first analysis pass (in the ideal case, tuple synthesis is entirely unnecessary), we ran our benchmarks for sample input tuple sizes of 10, 30, 100, 300, and 1000 tuples using our system and Olston's system. For each sample input tuple size, we executed each benchmark program 10 times with different randomly sampled input tuples. All experiments were performed on a four core 2.4 GHz machine with 6 GB of RAM.

We configured our system to compare directly to Olston's system, which is implemented as the "illustrate" command in Pig Latin. Unfortunately, the current implementation of Olston's system has some limitations not mentioned in the published paper. We wanted to evaluate against the approach and not against the implementation. To that end, we addressed such limitations or tweaked the benchmark programs so that the problems do not manifest themselves. The first issue is that the downstream pass would discard all sample input containing null fields. In the upstream pass, if all input tuples happen to contain null fields and thus all of them are discarded, there would be a *NullPointerException*. We sidestep the null field issue by ensuring that at least one sample input does not contain null fields. In addition, the system can only handle 32 `FILTER` conditions at most, as it encodes pertinent equivalence classes for `FILTER` conditions as individual bits in a (32-bit) integer index variable. This problem affects one PigMix benchmark program intended for scalability testing. The benchmark has a very large set (500+) of `FILTER` conditions. We sidestepped the problem by making changes to the PigMix program so that the resulting program has only 31 `FILTER` conditions. Also, when reasoning about a `JOIN` in the upstream pass, a *NullPointerException* is thrown if no data are observed in the input side of a `JOIN` (typically because one of its upstream neighbors is a highly selective operator). We address this issue by skipping the `JOIN` if its input has no data and then attempt to continue upstream propagation. Moreover, by mistakenly setting a non-tuple field to a tuple in a method involved in upstream propagation, a type casting error arises, which impedes the ability of Olston's system to reason over the `JOIN` and `FOREACH` operations if their downstream neighbor is a `FILTER` operator. We disallow assigning the non-tuple field to a tuple in the problematic method.

4.5.3 Results

We ran each experiment 10 times and averaged the completeness of 10 runs (since the completeness may theoretically vary due to different random choice of initial samples). The size of the input data has little effect. The results are almost the same for all sample input sizes.

Figures 4.3 and Figure 4.4 show the average completeness for each Pig Latin program in the PigMix and SDSS sets, respectively, for a sample input size of 100 tuples. Every bar corresponds to one program (with the exception of program *L12* in Figure 4.3, in which there are three subprograms and example data were generated for three different root operators corresponding to the three subprograms) for a total of 20 programs.

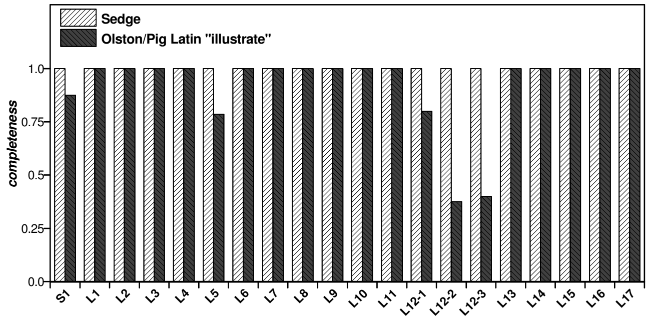

Figure 4.3: Completeness of sample data generation for the PigMix benchmarks

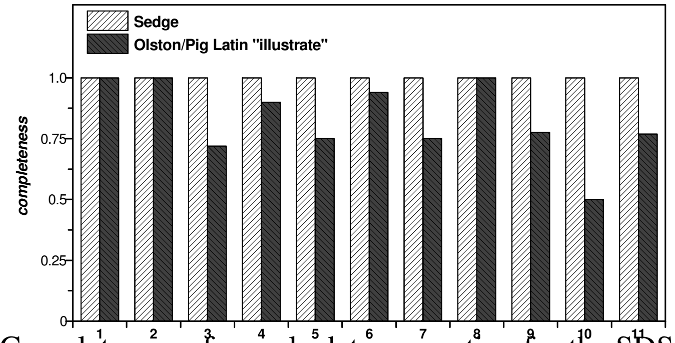

Figure 4.4: Completeness of sample data generation for the SDSS benchmarks

As can be seen, we improve on completeness for 5 out of 20 PigMix benchmark programs and 7 out of 11 SDSS benchmark programs. Although the benchmark programs are small and much of their coverage is achieved with random sampling of real inputs, they demonstrate clearly the benefits of our approach. *Practically every program in the two benchmark sets that has any kind of complexity (either more than one operator in the same path, or a user-defined function, or complex filter conditions) is not fully covered by Olston's approach.* For example, Olston's system cannot generate data that fail the FILTER in the presence of grouping, projecting, UDF invocation in the following program (program *S1* in Figure 4.3).

```
A = LOAD ’$widerow’ using PigStorage()
```

```
  AS (name: chararray, c0: int, c1: int, ..., c31: int);
B = GROUP A BY name;
C = FOREACH B GENERATE group, SUM(A.c0) as c0, SUM(A.c1) as c1,
   ..., SUM(A.c31) as c500;
D = FILTER C BY c0 > 100 AND c1 > 100 AND c2 > 100 ... AND c31 >
    100;
```

Listing 4.4: PigMix program S1

In fact, Sedge achieves perfect coverage (i.e., full completeness) for all benchmark programs. Compared to Olston's approach our improved coverage is due to stronger constraint solving ability (for programs *4,5,6,7* in Figure 4.4), to UDF handling ability (for programs *9,10* in Figure 4.4) and also to inter-related constraints and global reasoning (for programs *S1,L5,L12-1,L12-2,L12-3* in Figure 4.3 and programs *3,11* in Figure 4.4).

We also recorded how long it took Sedge and Olston's system to finish example generation. We include the infrastructure bootstrap time on each benchmark program. Both Sedge and Olston's system need to prepare the Hadoop execution environment for new executions. Sedge needs to load its constraint solver Z3 and CORAL as well.

As can be seen in Figure 4.5 and Figure 4.6, Sedge is faster on average than Olston's system in 18 out of 20 PigMix benchmark programs and 9 out of 11 SDSS benchmark programs. For the rest of benchmark programs, Sedge incurs a little higher running time than Olston's system. From these numbers we can infer that, although we have to conduct path exploration and constraint solving, there are even time savings in most cases due to avoiding the step of pruning redundant tuples after the upstream pass (because our approach does not generate redundant data).

4.6 Discussion: Why High-Level DSE

A natural qualitative comparison is between a Dynamic Symbolic Execution (DSE) engine at the level of the Pig Latin language and DSE engines for imperative languages, since Pig Latin code is eventually compiled into imperative code that uses a map-reduce library. The expected benefits from our approach are a) simplicity; b) conciseness of the generated

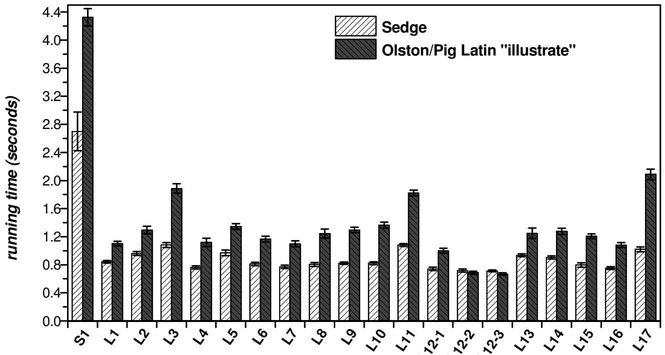

Figure 4.5: Running time of sample data generation for the PigMix benchmarks

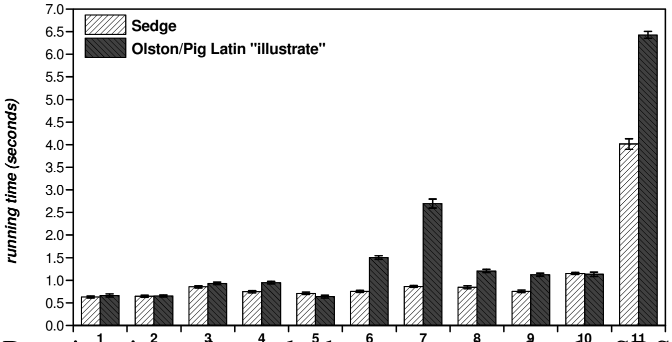

Figure 4.6: Running time of sample data generation for the SDSS benchmarks

test cases (i.e., the same coverage with fewer tests); and c) completeness: an imperative DSE engine may have trouble solving constraints over the logically more complex generated code, rather than the original Pig Latin code. Furthermore, an imperative DSE engine cannot take advantage of the lack of side-effects in order to better concretize user-defined functions, as discussed in Section 4.3.2.

We compared SEDGE with the Pex [104] state-of-the-art DSE engine in a limit study. Pex accepts `C#` input, hence we hand-translated Pig Latin programs into `C#` programs.[8] The resulting `C#` programs are single-threaded without any call to the map-reduce API, in order to test the applicability of Pex in the ideal case. (The inclusion of the map-reduce library complicates the control-flow of the imperative program even more and can easily

[8] Although there are DSE engines for Java—e.g., Dsc [62]—they do not match the industrial-strength nature of Pex. Dsc, for instance, does not support programs with floating point numbers, which are common in Pig Latin.

cause the DSE engine to miss a targeted branch of test execution, leading to low coverage of generated test cases [112].)

For our translation, we inspected the Java code generated by the Pig compiler and made a best-effort attempt to replicate it in C#, without map-reduce calls. We translated 13 programs from our Pig Latin benchmark suites. Since Pex has no knowledge of the original input (it accepts concrete values only when passed into the test method as parameters with primitive types) we enable just the 3rd pass (upstream pass) of SEDGE, for a fair comparison (i.e., SEDGE also does not benefit from sampled real data—this also disadvantages SEDGE as it removes the advantage of better UDF handling).

The results confirm our expectation. The conciseness of the test suite generated by Pex is low since Pex needs to examine a lot of irrelevant low-level branches or constraints that are not necessary for equivalence class coverage of the high-level Pig Latin control flow. For example, for step A in the Pig Latin program in Listing 4.3, Pex generates 30 tuples within 11 tables, of which 3 tuples pass the filter in step B, while SEDGE generates 2 tuples within exactly 1 table, of which 1 tuple passes the filter in step B. The conciseness of the test suite generated by Pex is 0.05, while the conciseness of the test suite generated by SEDGE is 0.75. Furthermore, for specific complex constructs we also get much higher completeness, although quite often Pex also gets perfect coverage. In our experience, for Pig Latin programs containing FILTER statements after (CO)GROUP or JOIN statements, the test suites yielded by Pex lack in completeness. For instance, in the SDSS program with 34 FILTER operations and 2 JOIN operations, the Pex completeness is only 0.09.

CHAPTER 5

RELATED WORK

There is a lot of interesting work on the three different research directions covered by this dissertation. This chapter discusses representative work for each area.

5.1 Residual Investigation: Predictive and Precise Bug Detection

Static and dynamic analyses are routinely chained together for checking program correctness conditions in programming languages, i.e., in compilers and runtime systems. Compilers check certain properties statically and insert runtime checks for remaining properties. A classic example is checking that an array read does not access a memory location outside the bounds of the array. To enforce this property, Java compilers traditionally insert a dynamic check into the code before each array read. To reduce the runtime overhead, static analyses such as ABCD [13] have been developed that can guarantee some reads as being within bounds, so that only the remaining ones have to be checked dynamically. Beyond array bounds checking, a similar static dynamic analysis pipeline has been applied to more complex properties. The Spec# extended compiler framework [5] is a prime example: it can prove some pre- and post-conditions statically and generates runtime checks for the remaining ones. [48] uses a combination of static and dynamic analysis for enforcing object protocols. Their approach separates the static checking of protocol correctness from a dynamic check of program conformance to the protocol.

In residual dynamic typestate analysis, explored by [32], [10] and [11], a dynamic typestate analysis that monitors all program transitions for bugs is reduced to a residual analysis that just monitors those program transitions that are left undecided by a previous static analysis. This approach exploits the fact that a static typestate analysis is typically complete, i.e., it over-approximates the states a program can be in. If for a small sub-region of the program the over-approximated state sets do not contain an error state, all transitions within such a region can be safely summarized and ignored by the subsequent residual dynamic analysis. At a high level, our approach adopts this idea, by only monitoring those aspects of the program that the static analysis has flagged as suspicious. However, our approach is more general in two dimensions, (1) typestate analysis is restricted to verifying finite state machine properties ("do not pop before push"), while our approach can be applied to more complex properties ("do not pop more than pushed") and (2) our dynamic analysis is predictive: it leverages dynamic results to identify bugs in code both executed and not executed during the analysis.

Beyond typestates, the idea of speeding up a dynamic program analysis by pre-computing some parts statically has been applied to other analyses, such as information flow analysis. For example, recent work by [23] provides a fast dynamic information flow analysis of JavaScript programs. JavaScript programs are highly dynamic and can load additional code elements during execution. These dynamically loaded program elements can only be checked dynamically. Their staged analysis statically propagates its results throughout the statically known code areas, up to the borders at which code can change at runtime. These intermediate results are then packaged into residual checkers that can be evaluated efficiently at runtime, minimizing the runtime checking overhead.

Our analysis can be seen in a similar light as residual dynamic typestate analysis and residual information flow analysis. If we take as a hypothetical baseline somebody running only our residual checkers, then adding the static bug finding analysis as a pre-step would indeed make the residual dynamic analysis more efficient, as the static analysis focuses the residual analysis on code that may have bugs. However, our goals are very different. Our real baseline is an established static analysis technique whose main problem is over-approximation, which leads to users ignoring true warnings.

Our earlier work on Check'n'Crash [25] and DSD-Crasher [26,95] can be seen as strict versions of residual analysis. These earlier techniques share our goal of convincing users of the validity of static bug warnings. However, Check'n'Crash and DSD-Crasher guarantee that a given warning is true, by generating and executing concrete test cases that satisfy the static warning, until a static warning can be replicated in a concrete execution or a user-defined limit is reached. While such proof is very convincing, it also narrows the technique's scope. I.e., our earlier tools could only confirm very few warnings. In our current residual investigation, we relax this strict interpretation and also consider predictive clues that are likely to confirm a static warning.

Dynamic symbolic execution is a recent combination of static and dynamic program analysis [18,46,63,103]. Dynamic symbolic execution is also a strict approach that warns a user only after it has generated and executed a test case that proves the existence of a bug. Compared to our analysis, dynamic symbolic execution is heavier-weight, by building and maintaining during program execution a fully symbolic representation of the program state. While such detailed symbolic information can be useful for many kinds of program analyses, our current residual investigations do not need such symbolic information, making our approach more scalable.

Monitoring-oriented programming (MOP) [19] shows how runtime monitoring of correctness conditions can be implemented more efficiently, even without a prefixed static analysis. JavaMOP, for example, compiles correctness conditions to Java aspects that add little runtime overhead. This technique is orthogonal to ours. I.e., as some of our dynamic analyses are currently implemented manually using AspectJ, expressing them in terms of JavaMOP would be a straightforward way to reduce our runtime overhead.

Our analysis can be viewed as a ranking system on static analysis error reports. There has been significant work in this direction using data mining. [67] sorts error reports by their probabilities. A model is used for computing probabilities for each error report by leveraging code locality, code versioning, and user feedback. The effectiveness of the model depends on (1) a fair number of reports and (2) strong clustering of false positives. [66] prioritize warning categories by mining change log history messages. Intuitively, they expect a warning category to be important if warning instances from the category are removed many times in the revision history of a system. Their method requires change logs of good quality. For static tools such as FindBugs, which analyze Java bytecode to generate warnings, they also require compilation of each revision. [12] use a machine-learning approach to filter out false positives of their static analyses for validating a finite state property. They identify cases where imprecise static analysis information can appear due to factors such as dynamic class loading and interprocedural data flow. They then use each case as a feature for decision tree learning. The key differene between our approach and data-mining-based ones is that our approach considers the possible objections to certain static error reports. As a result, we can validate more complicated error situations that require understanding what a programmer has in mind in practice when writing their code.

Besides Eraser [91], some other related work on race detection comes under the label of predictive race analysis. [20] predicts races by logging only relevant information in a program trace and then model-checking all feasible trace permutations. jPredictor [21] presents a polynomial algorithm that can search a thread scheduling for a potential race that did not occur in the observed execution. jPredictor is not sound. [98] define the causally-precedes relation (CP) that weakens the traditional Happens-Before relation. The CP approach can generalize beyond an observed execution and guarantee soundness and polynomial complexity. These sophisticated predictive approaches can offer greater precision than residual investigation in terms of race detection, whereas our residual investigation based race detector can have fewer false negatives, due to the better coverage of static analysis tools and the conservativeness of our dynamic analysis.

The precision of bug detection software can be greatly enhanced with human supervision at residual steps. After an automated static program verification step, [30] relies on humans to help reduce false positives by asking humans simple and relevant questions. [113]

proposes to use humans to provide guidance to otherwise intractable problems faced by test case generators. It would be interesting to combine such techniques in our approach. For instance, residual investigation's precision is influenced by the native test suite coverage. With the help of human users, additional test cases can be constructed if required.

5.2 Second-Order Constraints in Dynamic Invariant Inference

There is a wealth of other work on invariant inference. We next selectively focus on some recent approaches that were not covered in the body of the Chapter 3 (typically because they focus on static invariant inference techniques).

For reverse engineering, Gannod and Cheng [44] proposed to infer detailed specifications statically by computing the strongest postconditions. Nevertheless, pre/postconditions obtained from analyzing the implementation are usually too detailed to understand and too specific to support program evolution. Gannod and Cheng [45] addressed this deficiency by generalizing the inferred specification, for instance by deleting conjuncts, or adding disjuncts or implications. Their approach requires loop bounds and invariants, both of which must be added manually.

There has been some recent progress in inferring invariants using abstract interpretation. Logozzo [73,74] infers loop invariants while inferring class invariants. The limitation of his approach are the available abstract domains; numerical domains are best studied. Resulting specifications are expressed in terms of fields of classes.

Flanagan and Leino [39] propose a lightweight verification-based tool, named Houdini, to statically infer ESC/Java [40] annotations from unannotated Java programs. Based on pre-set property patterns, Houdini conjectures a large number of possible annotations and then uses ESC/Java to verify or refute each of them. The ability of this approach is limited by the patterns used. In fact, only simple patterns are feasible, otherwise too many candidate annotations will be generated, and, consequently, it will take a long time for ESC/Java to verify complicated properties.

Taghdiri [101] uses a counterexample-guided refinement process to infer over-approximate specifications for procedures called in the function being verified. In contrast to our approach, Taghdiri aims to approximate the behaviors for the procedures within the caller's context instead of inferring specifications of the procedure.

Henkel and Diwan [55] have built a tool to dynamically discover algebraic specifications for interfaces of Java classes. Their specifications relate sequences of method invocations. The tool generates many terms as test cases from the class signature. The results of these tests are generalized to algebraic specifications. They use a second tool to dynamically compare their specifications against implementations by executing both simultaneously and comparing their behavior [56].

Much of the work on specification mining is targeted at inferring API protocols dynamically. Whaley et al. [111] create a finite state machine into which a transition from method A to method B is added if the post-condition of method A is not mutually exclusive with the pre-condition of method B. Meghani and Ernst [76] build upon Whaley's work by using Daikon to determine the likely pre/post-condition of each method. Other approaches use data mining techniques. For instance Ammons et al. [2] use a learner to infer nondeterministic state machines from traces; similarly, Yang and Evans [114] built Terracotta, a tool to generate regular patterns of method invocations from observed runs of the program. Li and Zhou [71] apply data mining in the source code to infer programming rules, i.e., usage of related methods and variables, and then detect potential bugs by locating the violation of these rules. Gabel and Su [43] dynamically infer and verify method call ordering constraints, and report the constraints only if they are violated. Beschastnikh et al. [7] use mined temporal invariants from logs to derive a refined finite state machine. Although not explicitly our goal, some of our second-order constraints can be thought of as a way to express temporal API protocols. For example, we can find the general has$*$, next$*$ type specification [43] by checking if *CanFollow(has$*$(return==true),next$*$)*, or that the postcondition of has$*$ when has$*$ returns true implies the precondition of next$*$. It is interesting future work to see how to integrate existing techniques on specification mining with our approach to derive automata that describe correct behavior.

5.3 SEDGE: Symbolic Example Data Generation for Dataflow Programs

Dataflow languages such as Pig can been seen as a compromise between declarative languages, such as SQL, and imperative languages, such as C and Java. That is, Pig combines the declarative feature of straightforward parallel computation with the imperative feature of explicit intermediate results. There is little work (discussed in earlier sections) that addresses test data generation for dataflow languages. Instead, the related work from various research communities has focused on the extreme ends of this spectrum, i.e., either on SQL or on Java-like programming languages.

Specifically, related work in the software engineering community has focused on traditional procedural and object-oriented database-centric programs, tested via combinations of static and dynamic reasoning [96]. The main approaches use static symbolic execution [75] or dynamic symbolic execution [33, 70, 86]. While our work is inspired by such earlier dynamic symbolic execution approaches, we adapted this work to dataflow programs and their execution semantics. At the other end, there is work that automatically generates database data that satisfy external constraints [97] but there is no coverage or conciseness goal and no application to dataflow languages. Other work [106] has introduced the idea of code coverage to SQL queries. For our purposes, we reused the concept of coverage for Pig Latin as defined by Olston et al. [84].

In the formal methods community, Qex is generating test inputs for SQL queries [107]. Similar to our work, Qex maps a SQL query to SMT and uses the Z3 constraint solver to infer data tables. However Qex differs from our work in that Qex does not have a dynamic program analysis component and therefore cannot observe how a query processes existing example data. Earlier work in the software engineering community on dynamic symbolic execution has shown that dynamic analysis can make such program analysis more efficient and enable it to reason about user-defined functions, which we leverage in our work.

In the database community, a common methodology for testing a database management system or a database application is to generate a set of test databases given target query workloads. Overall our problem differs in that, instead of a whole database, we aim to generate a small (or minimum if desired) set of tuples that have perfect path coverage of a given dataflow program. The recent work on reverse query processing [8] takes an application query and a result set as input, and generates a corresponding input database by exploiting *reverse relational algebra*. In comparison, our work focuses on dataflow programs for big data applications, where many operators are non-relational, e.g., map(), reduce(), and arbitrary user-defined functions, and hence a "reverse algebra" may not exist. The QAGen system [9] further takes into account a set of constraints, usually cardinality and data distribution in input and operator output tables, and aims to generate a database that satisfies these constraints. Analogously to earlier work in the formal methods community, this work performs a static symbolic analysis and does not obtain additional information from a dynamic analysis.

CHAPTER 6

FUTURE WORK AND CONCLUSIONS

6.1 Future Work

Our future research projects are along two lines. The first part of our future projects is at the core of program analysis. For instance, now there is not a strong open-source dynamic-symbolic execution engine for Java programs. We are working on a cutting-edge and open-sourced dynamic-symbolic execution engine that is actually usable for most Java programs. A strong open-source Java dynamic-symbolic engine is of direct value for test case generation as described in Section 1.2. The second part of our future projects lies at the intersection of systems, program understanding, and bug detection. A wide variety of static and dynamic analysis tools are available for program understanding and bug detection. Unfortunately, existing tools are often not practical for use in huge and complex systems with distributed memory and parallel processing capabilities due to high overhead. There is significant value in providing scalable and computationally efficient ways to run those analyses. For example, any extensions of existing techniques for local concurrency bugs to the distributed context can greatly help developers debug their systems. The following are several immediate research opportunities.

Type constraints Type constraints are logical formulas related to types. Type constraints have important application areas such as verification and testing. One important application area of type constraints is dynamic symbolic execution of object-oriented programs, which often needs to create objects with correct types. Due to polymorphisms such as subtyping and dynamic dispatch, the dynamic type of a reference variable is often not equal to its declared static type. The challenge of type constraints reasoning is the analysis must be acceptably accurate not only in complete type hierarchy but also in incomplete type hierarchy while keeping the process at reasonable performance cost. We are working on specifying algorithms of type constraints analyses using Datalog, a database query language that is guaranteed to have a polynomial complexity.

Invariant inference Currently there are four popular approaches for invariant inference: generate-and-test using pre-defined invariant templates, generating propositions from symbolic execution along actual program execution paths, abstract interpretation, and machine-learning. Each method has its own advantages and disadvantages in different contexts. We plan to explore the relationship among different approaches in an effort to create tools that follow a hybrid approach.

Program analysis on map-reduce programs We believe that the area of dataflow programs (esp. map-reduce pipelines) can benefit from the application of program analysis techniques, to a much greater extent than current practice. As an example, we would like to explore how to generate program data that reveal changes among program versions. The generated example data can help understand code change, augment existing test suite, and verify intended change (e.g., whether the change brings any anomaly).

Debugging for Distributed Systems Debugging distributed system often incurs high-cost or significant human effort due to exponentially large state spaces. There is a great need for tools that can sift through a large number of state spaces and help programmers come up with hypothesis of bug causes. By combining the proper kinds of data mining, log analysis, and search facilities with program analysis, one could design an inference algorithm to learn what events should be responsible for a bug.

6.2 Conclusions

The thrust of this dissertation is to provide analysis algorithms for program understanding and bug detection by statically analyzing mixed code and data and dynamically analyzing the behavior of running programs. Our analysis algorithms emphasize precision and prediction in bug detection, correctness and relevancy in invariant inference, and completeness, realness, and efficiency in test case generation. More specifically, our main contribution are as follows:

1. We presented residual investigation: the idea of accompanying a static error analysis with an appropriately designed dynamic analysis that will report with high confidence whether the static error report is valid. We believe that residual investigation is, first and foremostly, an interesting *concept*. Identifying this concept helped us design dynamic analyses for a variety of static bug patterns and implement them in a tool, RFBI. We applied RFBI to a variety of test subjects to showcase the potential of the approach.

2. Second-order constraints can steer dynamic invariant inference to avoid erroneous invariants and to derive more relevant invariants while reducing noise. We have defined a vocabulary of second-order constraints and described how each of them encodes information that is typically known by programmers and useful to a dynamic invariant detector. We have taken an approach in which the second-order constraints control the propagation of the observations on which invariant detection is based. We have also extended the Daikon system so as to infer second-order constraints. Overall, we consider second-order constraints to be a particularly promising idea not just as a meaningful documentation concept but also for improving the consistency and quality of dynamically inferred invariants—a major challenge in this area.

3. Generating example input data for dataflow programs has emerged as an important challenge. We presented SEDGE: an approach and tool for generating example data of dataflow programs using dynamic symbolic execution, in order to achieve high coverage. SEDGE builds symbolic constraints over the equivalence classes induced by dataflow programming language constructs and can reason over constraints on user-defined functions by exploiting dynamic values as hints. We implemented our

technique for the Pig dataflow system and compared it empirically with the most closely related prior work. Our evaluation on third-party applications demonstrates that, with similar computing resources, our technique achieves better coverage of a given dataflow program.

BIBLIOGRAPHY

[1] ABADI, M., FLANAGAN, C., AND FREUND, S. N. Types for safe locking: Static race detection for Java. *ACM Transactions on Programming Languages and Systems (TOPLAS) 28*, 2 (Mar. 2006), 207–255.

[2] AMMONS, G., BODIK, R., AND LARUS, J. R. Mining specifications. In *Proc. 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)* (Jan. 2002), ACM, pp. 4–16.

[3] AYEWAH, N., AND PUGH, W. The Google FindBugs fixit. In *Proc. 19th International Symposium on Software Testing and Analysis (ISSTA)* (2010), ACM, pp. 241–252.

[4] BALZAROTTI, D., COVA, M., FELMETSGER, V., JOVANOVIC, N., KIRDA, E., KRUEGEL, C., AND VIGNA, G. Saner: Composing static and dynamic analysis to validate sanitization in web applications. In *Security and Privacy, 2008. SP 2008. IEEE Symposium on* (May 2008), pp. 387–401.

[5] BARNETT, M., LEINO, K. R. M., AND SCHULTE, W. The Spec# programming system: An overview. In *Proc. International Workshop on the Construction and Analysis of Safe, Secure, and Interoperable Smart Devices (CASSIS)* (Mar. 2004), Springer, pp. 49–69.

[6] BEHM, A., BORKAR, V. R., CAREY, M. J., GROVER, R., LI, C., ONOSE, N., VERNICA, R., DEUTSCH, A., PAPAKONSTANTINOU, Y., AND TSOTRAS, V. J. Asterix: towards a scalable, semistructured data platform for evolving-world models. *Distributed and Parallel Databases 29*, 3 (2011), 185–216.

[7] Beschastnikh, I., Brun, Y., Schneider, S., Sloan, M., and Ernst, M. D. Leveraging existing instrumentation to automatically infer invariant-constrained models. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering* (New York, NY, USA, 2011), ESEC/FSE '11, ACM, pp. 267–277.

[8] Binnig, C., Kossmann, D., and Lo, E. Reverse query processing. In *Proc. 23rd International Conference on Data Engineering (ICDE)* (Apr. 2007), IEEE, pp. 506–515.

[9] Binnig, C., Kossmann, D., Lo, E., and Özsu, M. T. QAGen: Generating query-aware test databases. In *Proc. ACM SIGMOD International Conference on Management of Data (SIGMOD)* (June 2007), ACM, pp. 341–352.

[10] Bodden, E. Efficient hybrid typestate analysis by determining continuation-equivalent states. In *Proc. 32nd ACM/IEEE International Conference on Software Engineering (ICSE)* (May 2010), ACM, pp. 5–14.

[11] Bodden, E., Hendren, L., and Lhoták, O. A staged static program analysis to improve the performance of runtime monitoring. In *Proc. 21st European Conference on Object-Oriented Programming (ECOOP)* (July 2007), pp. 525–549.

[12] Bodden, E., Lam, P., and Hendren, L. Finding programming errors earlier by evaluating runtime monitors ahead-of-time. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering* (New York, NY, USA, 2008), SIGSOFT '08/FSE-16, ACM, pp. 36–47.

[13] Bodik, R., Gupta, R., and Sarkar, V. ABCD: Eliminating array bounds checks on demand. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)* (June 2000), ACM, pp. 321–333.

[14] BORGES, M., D'AMORIM, M., ANAND, S., BUSHNELL, D., AND PASAREANU, C. S. Symbolic execution with interval solving and meta-heuristic search. In *Proceedings of the 2012 IEEE Fifth International Conference on Software Testing, Verification and Validation* (Washington, DC, USA, 2012), ICST '12, IEEE Computer Society, pp. 111–120.

[15] BRYANT, R. E., GERMAN, S. M., AND VELEV, M. N. Exploiting positive equality in a logic of equality with uninterpreted functions. In *Proc. 11th International Conference on Computer Aided Verification (CAV)* (1999), Springer, pp. 470–482.

[16] BURCH, J. R., AND DILL, D. L. Automatic verification of pipelined microprocessor control. In *Proc. 6th International Conference on Computer Aided Verification (CAV)* (1994), Springer, pp. 68–80.

[17] CADAR, C., DUNBAR, D., AND ENGLER, D. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proc. 8th USENIX Conference on Operating Systems Design and Implementation (OSDI)* (2008), USENIX, pp. 209–224.

[18] CADAR, C., AND ENGLER, D. R. Execution generated test cases: How to make systems code crash itself. In *Proc. 12th International SPIN Workshop on Model Checking Software* (Aug. 2005), Springer, pp. 2–23.

[19] CHEN, F., AND ROŞU, G. Mop: An efficient and generic runtime verification framework. In *Proc. 22nd ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)* (Oct. 2007), ACM, pp. 569–588.

[20] CHEN, F., AND ROŞU, G. Parametric and sliced causality. In *Proceedings of the 19th International Conference on Computer Aided Verification* (Berlin, Heidelberg, 2007), CAV'07, Springer-Verlag, pp. 240–253.

[21] Chen, F., Serbanuta, T. F., and Rosu, G. jpredictor: A predictive runtime analysis tool for java. In *Proceedings of the 30th International Conference on Software Engineering* (New York, NY, USA, 2008), ICSE '08, ACM, pp. 221–230.

[22] Christensen, A. S., Møller, A., and Schwartzbach, M. I. Extending Java for high-level Web service construction. *ACM Transactions on Programming Languages and Systems 25*, 6 (November 2003), 814–875.

[23] Chugh, R., Meister, J. A., Jhala, R., and Lerner, S. Staged information flow for JavaScript. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)* (June 2009), ACM, pp. 50–62.

[24] 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.

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

[26] 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)* (July 2006), ACM, pp. 245–254.

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

[28] Csallner, C., Tillmann, N., and Smaragdakis, Y. DySy: Dynamic symbolic execution for invariant inference. In *International Conference on Software Engineering (ICSE)* (May 2008).

[29] De Moura, L., and Bjørner, N. Satisfiability modulo theories: introduction and applications. *Commun. ACM 54*, 9 (Sept. 2011), 69–77.

[30] Dillig, I., Dillig, T., and Aiken, A. Automated error diagnosis using abductive inference. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation* (New York, NY, USA, 2012), PLDI '12, ACM, pp. 181–192.

[31] D'Souza, D., Kim, Y. P., Kral, T., Ranade, T., and Sasalatti, S. Tool evaluation report: Fortify. http://www.cs.cmu.edu/ aldrich/courses/654/tools/dsouza-fortify-07.pdf, Apr. 2007. Accessed Jan. 2014.

[32] Dwyer, M. B., and Purandare, R. Residual dynamic typestate analysis exploiting static analysis: Results to reformulate and reduce the cost of dynamic analysis. In *Proc. 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE)* (Nov. 2007), ACM, pp. 124–133.

[33] Emmi, M., Majumdar, R., and Sen, K. Dynamic test input generation for database applications. In *Proc. ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)* (July 2007), ACM, pp. 151–162.

[34] Engler, D. R., and Ashcraft, K. RacerX: Effective, static detection of race conditions and deadlocks. In *Proc. 19th ACM Symposium on Operating Systems Principles (SOSP)* (Oct. 2003), ACM, pp. 237–252.

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

[36] 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) 27*, 2 (Feb. 2001), 99–123.

[37] Flanagan, C., and Freund, S. N. Fasttrack: Efficient and precise dynamic race detection. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation* (New York, NY, USA, 2009), PLDI '09, ACM, pp. 121–133.

[38] Flanagan, C., and Freund, S. N. The roadrunner dynamic analysis framework for concurrent programs. In *Proceedings of the 9th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering* (New York, NY, USA, 2010), PASTE '10, ACM, pp. 1–8.

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

[40] 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)* (June 2002), ACM, pp. 234–245.

[41] Fowler, M., Beck, K., Brant, J., Opdyke, W., and Roberts, D. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.

[42] Fraser, G., and Arcuri, A. Sound empirical evidence in software testing. In *Proc. International Conference on Software Engineering (ICSE)* (June 2012), IEEE, pp. 178–188.

[43] Gabel, M., and Su, Z. Online inference and enforcement of temporal properties. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1* (New York, NY, USA, 2010), ICSE '10, ACM, pp. 15–24.

[44] GANNOD, G. C., AND CHENG, B. H. C. Strongest postcondition semantics as the formal basis for reverse engineering. In *Proc. Second Working Conference on Reverse Engineering (WCRE)* (July 1995), IEEE, pp. 188–197.

[45] GANNOD, G. C., AND CHENG, B. H. C. A specification matching based approach to reverse engineering. In *Proc. International Conference on Software Engineering* (May 1999), ACM, pp. 389–398.

[46] GODEFROID, P., KLARLUND, N., AND SEN, K. Dart: Directed automated random testing. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)* (June 2005), ACM, pp. 213–223.

[47] GODEFROID, P., LEVIN, M. Y., AND MOLNAR, D. Sage: Whitebox fuzzing for security testing. *Queue 10*, 1 (Jan. 2012), 20:20–20:27.

[48] GOPINATHAN, M., AND RAJAMANI, S. K. Enforcing object protocols by combining static and runtime analysis. In *Proc. 23rd ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)* (Oct. 2008), ACM, pp. 245–260.

[49] GOULD, C., SU, Z., AND DEVANBU, P. JDBC checker: A static analysis tool for SQL/JDBC applications. In *Proc. International Conference on Software Engineering (ICSE)* (May 2004), IEEE, pp. 697–698.

[50] GRISWOLD, W. G., AND NOTKIN, D. Program Restructuring as an Aid to Software Maintenance. Tech. rep., University of Washington, Seattle, WA, USA, August 1990.

[51] GUO, P. J., PERKINS, J. H., MCCAMANT, S., AND ERNST, M. D. Dynamic inference of abstract types. In *Proceedings of the 2006 international symposium on Software testing and analysis* (New York, NY, USA, 2006), ISSTA '06, ACM, pp. 255–265.

[52] GUYER, S. Z., AND LIN, C. An annotation language for optimizing software libraries. In *Second Conference on Domain-Specific Languages (DSL)* (1999), pp. 39–52.

[53] HALFOND, W. G., AND ORSO, A. Combining Static Analysis and Runtime Monitoring to Counter SQL-Injection Attacks. In *Proc. 3rd International Workshop on Dynamic Analysis (WODA)* (May 2005), pp. 22–28.

[54] 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)* (May 2002), ACM, pp. 291–301.

[55] HENKEL, J., REICHENBACH, C., AND DIWAN, A. Discovering documentation for Java container classes. *IEEE Transactions on Software Engineering (TSE) 33*, 8 (Aug. 2007), 526–543.

[56] HENKEL, J., REICHENBACH, C., AND DIWAN, A. Developing and Debugging Algebraic Specifications for Java classes. *ACM Trans. Softw. Eng. Methodol. 17*, 3 (2008).

[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)* (Oct. 2004), ACM, pp. 132–136.

[58] HOVEMEYER, D., AND PUGH, W. Finding bugs is easy. *SIGPLAN Notices 39*, 12 (Dec. 2004), 92–106.

[59] HOVEMEYER, D., AND PUGH, W. Finding more null pointer bugs, but not too many. In *Proc. 7th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE)* (June 2007), ACM, pp. 9–14.

[60] HUANG, Y.-W., YU, F., HANG, C., TSAI, C.-H., LEE, D.-T., AND KUO, S.-Y. Securing web application code by static analysis and runtime protection. In *Proceedings of the 13th International Conference on World Wide Web* (New York, NY, USA, 2004), WWW '04, ACM, pp. 40–52.

[61] ISARD, M., BUDIU, M., YU, Y., BIRRELL, A., AND FETTERLY, D. Dryad: Distributed data-parallel programs from sequential building blocks. In *Proc. 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems (EuroSys)* (2007), ACM, pp. 59–72.

[62] ISLAM, M., AND CSALLNER, C. Dsc+mock: a test case + mock class generator in support of coding against interfaces. In *Proceedings of the Eighth International Workshop on Dynamic Analysis* (New York, NY, USA, 2010), WODA '10, ACM, pp. 26–31.

[63] ISLAM, M., AND CSALLNER, C. Generating test cases for programs that are coded against interfaces and annotations. *ACM Transactions on Software Engineering and Methodology (TOSEM) 23* (May 2014), 21:1–21:38.

[64] KANNAN, Y., AND SEN, K. Universal symbolic execution and its application to likely data structure invariant generation. In *ISSTA '08: Proceedings of the 2008 Int. Symposium on Software Testing and Analysis* (New York, NY, USA, 2008), ACM, pp. 283–294.

[65] KICZALES, G., HILSDALE, E., HUGUNIN, J., KERSTEN, M., PALM, J., AND GRISWOLD, W. G. An overview of AspectJ. In *Proc. 15th European Conference on Object Oriented Programming (ECOOP)* (June 2001), Springer, pp. 327–353.

[66] KIM, S., AND ERNST, M. D. Which warnings should I fix first? In *Proc. 11th European Software Engineering Conference and the 15th ACM SIGSOFT Symposium on Foundations of Software Engineering (ESEC/FSE)* (Sept. 2007), ACM, pp. 45–54.

[67] KREMENEK, T., ASHCRAFT, K., YANG, J., AND ENGLER, D. Correlation exploitation in error ranking. In *Proc. 12th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)* (Oct. 2004), ACM, pp. 83–93.

[68] LAHIRI, S. K., QADEER, S., AND RAKAMARIĆ, Z. Static and precise detection of concurrency errors in systems code using smt solvers. In *Proc. 21st International Conference on Computer Aided Verification (CAV)* (June 2009), Springer, pp. 509–524.

[69] 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.

[70] LI, C., AND CSALLNER, C. Dynamic symbolic database application testing. In *Proc. 3rd International Workshop on Testing Database Systems (DBTest)* (June 2010), ACM.

[71] LI, Z., AND ZHOU, Y. PR-miner: automatically extracting implicit programming rules and detecting violations in large software code. In *Proc. 13th International Symposium on Foundations of Software Engineering (FSE)* (Sept. 2005), ACM, pp. 306–315.

[72] LIANG, S. *Java Native Interface: Programmer's Guide and Specification*. Prentice Hall, June 1999.

[73] LOGOZZO, F. Automatic inference of class invariants, 2004.

[74] LOGOZZO, F. *Modular Static Analysis of Object-Oriented Languages*. PhD thesis, Ecole Polytechnique, June 2004.

[75] MARCOZZI, M., VANHOOF, W., AND HAINAUT, J.-L. Test input generation for database programs using relational constraints. In *Proc. 5th International Workshop on Testing Database Systems (DBTest)* (May 2012), ACM.

[76] MEGHANI, S. V., AND ERNST, M. D. Determining legal method call sequences in object interfaces, May 2003.

[77] MENS, T., AND TOURWE, T. A survey of software refactoring. *IEEE Trans. Softw. Eng. 30*, 2 (2004), 126–139.

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

[79] 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)* (July 2003), Elsevier.

[80] NAIK, M., AND AIKEN, A. Conditional must not aliasing for static race detection. In *Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (New York, NY, USA, 2007), POPL '07, ACM, pp. 327–338.

[81] NAIK, M., AIKEN, A., AND WHALEY, J. Effective static race detection for Java. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)* (June 2006), ACM, pp. 308–319.

[82] NEWSOME, J. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *Proc. Network and Distributed System Security Symposium (NDSS)* (Feb. 2005), The Internet Society.

[83] Okur, S., and Dig, D. How do developers use parallel libraries? In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering* (New York, NY, USA, 2012), FSE '12, ACM, pp. 54:1–54:11.

[84] Olston, C., Chopra, S., and Srivastava, U. Generating example data for dataflow programs. In *Proc. 2009 ACM SIGMOD International Conference on Management of Data (SIGMOD)* (2009), ACM, pp. 245–256.

[85] Olston, C., Reed, B., Srivastava, U., Kumar, R., and Tomkins, A. Pig latin: A not-so-foreign language for data processing. In *Proc. ACM SIGMOD International Conference on Management of Data (SIGMOD)* (2008), ACM, pp. 1099–1110.

[86] Pan, K., Wu, X., and Xie, T. Generating program inputs for database application testing. In *Proc. 26th IEEE/ACM International Conference on Automated Software Engineering (ASE)* (Nov. 2011), IEEE, pp. 73–82.

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

[88] Radoi, C., and Dig, D. Practical static race detection for java parallel loops. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis* (New York, NY, USA, 2013), ISSTA 2013, ACM, pp. 178–190.

[89] Reichenbach, C., Coughlin, D., and Diwan, A. Program Metamorphosis. In *European Conference on Object-Oriented Programming (ECOOP)* (2009), Springer, pp. 394–418.

[90] 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)* (Nov. 2004), IEEE, pp. 245–256.

[91] Savage, S., Burrows, M., Nelson, G., Sobalvarro, P., and Anderson, T. Eraser: A dynamic data race detector for multi-threaded programs. In *Proc. 16th Symposium on Operating Systems Principles (SOSP)* (Oct. 1997), ACM, pp. 27–37.

[92] Schwartz, E. J., Avgerinos, T., and Brumley, D. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In *Proceedings of the 2010 IEEE Symposium on Security and Privacy* (Washington, DC, USA, 2010), SP '10, IEEE Computer Society, pp. 317–331.

[93] Sen, K., and Agha, G. Cute and jCute: Concolic unit testing and explicit path model-checking tools. In *Proc. 18th International Conference on Computer Aided Verification (CAV)* (Aug. 2006), Springer, pp. 419–423.

[94] 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)* (Sept. 2005), ACM, pp. 263–272.

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

[96] Smaragdakis, Y., and Csallner, C. Combining static and dynamic reasoning for bug detection. In *TAP* (2007), Y. Gurevich and B. Meyer, Eds., vol. 4454 of *Lecture Notes in Computer Science*, Springer, pp. 1–16.

[97] Smaragdakis, Y., Csallner, C., and Subramanian, R. Scalable automatic test data generation from modeling diagrams. In *Automated Software Engineering conference (ASE)* (Nov. 2007), ACM Press, pp. 4–13.

[98] SMARAGDAKIS, Y., EVANS, J., SADOWSKI, C., YI, J., AND FLANAGAN, C. Sound predictive race detection in polynomial time. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (New York, NY, USA, 2012), POPL '12, ACM, pp. 387–400.

[99] SONG, D., BRUMLEY, D., YIN, H., CABALLERO, J., JAGER, I., KANG, M. G., LIANG, Z., NEWSOME, J., POOSANKAM, P., AND SAXENA, P. Bitblaze: A new approach to computer security via binary analysis. In *International Conference on Information Systems Security* (2008), Springer, pp. 1–25.

[100] STERLING, N. Warlock: A static data race analysis tool. In *USENIX Winter Technical Conference* (Jan. 1993), Usenix, pp. 97–106.

[101] TAGHDIRI, M. Inferring specifications to detect errors in code. In *Proc. 19th IEEE International Conference on Automated Software Engineering (ASE)* (Sept. 2004), IEEE, pp. 144–153.

[102] THUSOO, A., SARMA, J. S., JAIN, N., SHAO, Z., CHAKKA, P., ANTHONY, S., LIU, H., WYCKOFF, P., AND MURTHY, R. Hive - a warehousing solution over a map-reduce framework. *PVLDB 2*, 2 (2009), 1626–1629.

[103] TILLMANN, N., AND DE HALLEUX, J. Pex - white box test generation for .Net. In *Proc. 2nd International Conference on Tests And Proofs (TAP)* (Apr. 2008), Springer, pp. 134–153.

[104] TILLMANN, N., AND DE HALLEUX, J. Pex: White box test generation for .Net. In *Proc. 2nd International Conference on Tests and Proofs (TAP)* (2008), Springer, pp. 134–153.

[105] 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)* (July 2007), ACM, pp. 97–107.

[106] TUYA, J., SUÁREZ-CABAL, M. J., AND DE LA RIVA, C. Full predicate coverage for testing SQL database queries. *Software Testing, Verification & Reliability (STVR) 20*, 3 (Sept. 2010), 237–288.

[107] VEANES, M., TILLMANN, N., AND DE HALLEUX, J. Qex: Symbolic SQL query explorer. In *Proc. 16th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR)* (Apr. 2010), Springer, pp. 425–446.

[108] 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)* (July 2004), ACM, pp. 97–107.

[109] VOUNG, J. W., JHALA, R., AND LERNER, S. Relay: Static race detection on millions of lines of code. In *Proc. ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)* (Sept. 2007), ACM, pp. 205–214.

[110] 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)* (May 2005), Springer, pp. 40–55.

[111] WHALEY, J., MARTIN, M. C., AND LAM, M. S. Automatic extraction of object-oriented component interfaces. In *Proc. International Symposium on Software Testing and Analysis (ISSTA)* (July 2002), ACM, pp. 218–228.

[112] XIAO, X., XIE, T., TILLMANN, N., AND DE HALLEUX, J. Precise identification of problems for structural test generation. In *Proceedings of the 33rd International Conference on Software Engineering* (New York, NY, USA, 2011), ICSE '11, ACM, pp. 611–620.

[113] XIE, T. Cooperative testing and analysis: Human-tool, tool-tool and human-human cooperations to get work done. In *Source Code Analysis and Manipulation (SCAM), 2012 IEEE 12th International Working Conference on* (2012), pp. 1–3.

[114] YANG, J., AND EVANS, D. Dynamically inferring temporal properties. In *Proc. 5th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE)* (June 2004), ACM, pp. 23–28.

[115] 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)* (Oct. 2004), ACM, pp. 97–106.