Combining Static and Dynamic Reasoning for Bug Detection

Yannis Smaragdakis[1] and Christoph Csallner[2]

[1] Department of Computer Science
University of Oregon, Eugene, OR 97403-1202, USA
yannis@cs.uoregon.edu
[2] College of Computing
Georgia Institute of Technology, Atlanta, GA 30332, USA
csallner@gatech.edu

Abstract. Many static and dynamic analyses have been developed to improve program quality. Several of them are well known and widely used in practice. It is not entirely clear, however, how to put these analyses together to achieve their combined benefits. This paper reports on our experiences with building a sequence of increasingly more powerful combinations of static and dynamic analyses for bug finding in the tools JCrasher, Check 'n' Crash, and DSD-Crasher. We contrast the power and accuracy of the tools using the same example program as input to all three.

At the same time, the paper discusses the philosophy behind all three tools. Specifically, we argue that trying to detect program errors (rather than to certify programs for correctness) is well integrated in the development process and a promising approach for both static and dynamic analyses. The emphasis on finding program errors influences many aspects of analysis tools, including the criteria used to evaluate them and the vocabulary of discourse.

1 Introduction

Programming is hard. As an intellectual task, it attempts to approximate real-world entities and conditions as abstract concepts. Since computers are unforgiving interpreters of our specifications, and since in software we can build up complexity with no physical boundaries, it is easy to end up with artifacts that are very hard to comprehend and reason about. Even moderate size programs routinely surpass in detail and rigor the most complex laws, constitutions, and agreements in the "real world". Not only can individual program modules be complex, but the interactions among modules can be hardly known. Most programmers work with only a partial understanding of the parts of the program that their own code interacts with. Faced with this complexity, programmers need all the help they can get. In industrial practice, testing has become significantly more intense and structured in the past decade. Additionally, numerous static analyses attempt to automatically certify properties of a program, or detect errors in it.

In the past few years, we have introduced three program analysis tools for finding program defects (bugs) in Java applications. *JCrasher* [3] is a simple, mostly dynamic analysis that generates JUnit test cases. Despite its simplicity it can find bugs that would require complex static analysis efforts. *Check 'n' Crash* [4] uses JCrasher as a post-processing step to the powerful static analysis tool ESC/Java. As a result, Check 'n' Crash is more precise than ESC/Java alone and generates better targeted test cases than JCrasher alone. *DSD-Crasher* [5] adds a reverse engineering step to Check 'n' Crash to rediscover the program's intended behavior. This enables DSD-Crasher to suppress false positives with respect to the program's informal specification. This property is more useful for bug-finding than for proving correctness, as we argue later.

In this paper, we report on our experience with these tools and present their comparative merits through a simple example. At the same time, we discuss in detail our philosophy in building them. All three tools are explicitly geared towards finding program errors and not towards certifying program correctness. Viewed differently, program analyses (regardless of the artificial static/dynamic distinction) can never accurately classify with full confidence all programs as either correct or incorrect. Our claim is that analyses that choose to be confident in their incorrectness classification (*sound for incorrectness*) are gaining ground over analyses that choose to be confident in their correctness classification (*sound for correctness*). We discuss this point next in more detail.

2 Bug Finding Musings

There are several dichotomies in program analysis. Clearly, analyses are often classified as *static* or *dynamic*. Additionally, analyses are often classified as *sound* or *complete*, or as *over-* and *under-approximate*. We next present some thoughts on these distinctions as well as the terminology they introduce.

2.1 Static and Dynamic Analysis

At first glance it may seem simple to classify an analysis as static or dynamic. The definition in the popular Wikipedia archive claims that:

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

This definition is not quite satisfying, however. Program execution only differs from program reasoning at the level of accuracy. This distinction is fairly artificial. First, there are languages where reasoning and execution are often thought of in the same terms (e.g., static analyses of Prolog programs often include steps such as "execute the program in a universe that only includes these values"). Second, even in imperative languages, it is often hard to distinguish between a virtual machine that executes the program and tools that reason about

it at some level of abstraction (e.g., model checking tools, or symbolic execution analyses). Finally, it is hard to classify analyses that execute a program with known inputs. Known inputs are by definition "static", in standard terminology, and these analyses give information about the program without executing it under "real" conditions. Yet at the same time, since the program is executed, it is tempting to call such analyses "dynamic".

We believe that there is a continuum of analyses and the static vs. dynamic classification is not always easy to make. Our working definition is as follows:

> An analysis is "dynamic" if it emphasizes control-flow accuracy over data-flow richness/generality, and "static" if it emphasizes data-flow richness/generality over control-flow accuracy.

There is always a trade-off between these trends. The undecidability of most useful program properties entails that one cannot make statements about infinitely many inputs without sacrificing some control-flow accuracy.

Although the definition is approximate, we believe that it serves a useful purpose. It reflects the intuitive understanding of the two kinds of analyses, while emphasizing that the distinction is arbitrary. A more useful way to classify analyses is in terms of *what* they claim not *how* they maintain the information that leads to their claims.

2.2 Soundness for Incorrectness

Analyses can be classified with respect to the set of properties they can establish with confidence. In mathematical logic, reasoning systems are often classified as *sound* and *complete*. A sound system is one that proves only true sentences, whereas a complete system proves all true sentences. In other words, an analysis is sound iff $\mathit{provable}(p) \Rightarrow \mathit{true}(p)$ and complete iff $\mathit{true}(p) \Rightarrow \mathit{provable}(p)$. Writing the definitions in terms of what the analysis claims, we can say:

Definition 1 (Sound). $\mathit{claim}_{true}(p) \Rightarrow \mathit{true}(p)$.

Definition 2 (Complete). $\mathit{true}(p) \Rightarrow \mathit{claim}_{true}(p)$.

When we analyze programs we use these terms in a qualified way. For instance, a type system (the quintessential "sound" static analysis) only proves correctness with respect to certain errors.

In our work, we like to view program analyses as a way to prove programs *incorrect*—i.e., to find bugs, as opposed to certifying the absence of bugs. If we escape from the view of program analysis as a "proof of correctness" and we also allow the concept of a "proof of incorrectness", our terminology can be adjusted. Useful program analyses give an answer for all programs (even if the analysis does not terminate, the programmer needs to interpret the non-termination-within-time-bounds in some way). In this setting, an analysis is sound for showing program correctness iff it is complete for showing program incorrectness. Similarly, an analysis is sound for showing program incorrectness iff it is complete for showing program correctness.

These properties are easily seen from the definitions. We have:

Lemma 1. *Complete for program correctness ≡ Sound for program incorrectness.*

Proof. Complete for program correctness
$\equiv correct(p) \Rightarrow claim_{cor}(p)$
$\equiv \neg incorrect(p) \Rightarrow \neg claim_{incor}(p)$
$\equiv claim_{incor}(p) \Rightarrow incorrect(p)$
≡ Sound for program incorrectness

Lemma 2. *Complete for program incorrectness ≡ Sound for program correctness.*

Proof. Complete for program incorrectness
$\equiv incorrect(p) \Rightarrow claim_{incor}(p)$
$\equiv \neg correct(p) \Rightarrow \neg claim_{cor}(p)$
$\equiv claim_{cor}(p) \Rightarrow correct(p)$
≡ Sound for program correctness

In the above, we considered the complementary use of the analysis, such that it claims incorrectness whenever the original analysis would not claim correctness. Note that the notion of "claim" is external to the analysis. An analysis either passes or does not pass programs, and "claim" is a matter of interpretation. Nevertheless, the point is that the same base analysis can be used to either soundly show correctness or completely show incorrectness, depending on how the claim is interpreted.

The interesting outcome of the above reasoning is that we can abolish the notion of "completeness" from our vocabulary. We believe that this is a useful thing to do for program analysis. Even experts are often hard pressed to name examples of "complete" analyses and the term rarely appears in the program analysis literature (in contrast to mathematical logic). Instead, we can equivalently refer to analyses that are "sound for correctness" and analyses that are "sound for incorrectness". An analysis does not have to be either, but it certainly cannot be both for interesting correctness properties.

Other researchers have settled on different conventions for classifying analyses, but we think our terminology is preferable. For instance, Jackson and Rinard call a static analysis "sound" when it is sound for correctness, yet call a dynamic analysis "sound" when it is sound for incorrectness [12]. This is problematic, since, as we argued, static and dynamic analyses form a continuum. Furthermore, the terminology implicitly assumes that static analyses always attempt to prove correctness. Yet, there are static analyses whose purpose is to detect defects (e.g., FindBugs by Hovemeyer and Pugh [11]). Another pair of terms used often are "over-" and "under-approximate". These also require qualification (e.g., "over-approximate for incorrectness" means the analysis errs on the safe side, i.e., is sound for correctness) and are often confusing.

2.3 Why Prove a Program Incorrect?

Ensuring that a program is correct is the Holy Grail of program construction. Therefore analyses that are sound for correctness have been popular, even if limited. For instance, a static type system guarantees the absence of certain kinds of bugs, such as attempting to perform an operation not defined for our data. Nevertheless, for all interesting properties, soundness for correctness implies that the analysis has to be pessimistic and reject perfectly valid programs. For some kinds of analyses this cost is acceptable. For others, it is not—for instance, no mainstream programming language includes sound static checking to ensure the lack of division-by-zero errors, exactly because of the expected high rejection rate of correct programs.

Instead, it is perfectly valid to try to be sound for incorrectness. That is, we may want to show that a program fails with full confidence. This is fairly expected for dynamic analysis tools, but it is worth noting that even static analyses have recently adopted this model. For instance, Lindahl and Sagonas's *success typings* [14] are an analogue of type systems but with the opposite trade-offs. Whereas a type system is sound for correctness and, hence, pessimistic, a success typing is sound for incorrectness and, thus, optimistic. If a success typing cannot detect a type clash, the program might work and is permitted. If the system does report a problem, then the problem is guaranteed to be real. This is a good approach for languages with a tradition of dynamic typing, where users will likely complain if a static type system limits expressiveness in the name of preventing unsafety.

Yet the most important motivation for analyses that are sound for incorrectness springs from the way analyses are used in practice. For the author of a piece of code, a sound-for-correctness analysis may make sense: if the analysis is too conservative, then the programmer probably knows how to rewrite the code to expose its correctness to the analysis. Beyond this stage of the development process, however, conservativeness stops being an asset and becomes a liability. A tester cannot distinguish between a false warning and a true bug. Reporting a non-bug to the programmer is highly counter-productive if it happens with any regularity. Given the ever-increasing separation of the roles of programmer and tester in industrial practice, high confidence in detecting errors is paramount.

This need can also be seen in the experience of authors of program analyses and other researchers. Several modern static analysis tools [10, 8, 11] attempt to find program defects. In their assessment of the applicability of ESC/Java, Flanagan et al. write [10]:

> "[T]he tool has not reached the desired level of cost effectiveness. In particular, users complain about an annotation burden that is perceived to be heavy, and about excessive warnings about non-bugs, particularly on unannotated or partially-annotated programs."

The same conclusion is supported by the findings of other researchers. Notably, Rutar et al. [19] examine ESC/Java2, among other analysis tools, and conclude

that it can produce many spurious warnings when used without context information (method annotations). For five testees with a total of some 170 thousand non commented source statements, ESC warns of a possible null dereference over nine thousand times. Rutar et al., thus, conclude that "there are too many warnings to be easily useful by themselves."

To summarize, it is most promising to use analyses that are sound for correctness at an early stage of development (e.g., static type system). Nevertheless, for analyses performed off-line, possibly by third parties, it is more important to be trying to find errors with high confidence or even certainty. This is the goal of our analysis tools. We attempt to increase the soundness of existing analyses by combining them in a way that reduces the false error reports. Just like analyses that are sound for correctness, we cannot claim full correctness, yet we can claim that our tools are sound for incorrectness with respect to specific kinds of errors. Such soundness-for-incorrectness topics are analyzed in the next section.

3 Soundness of Automatic Bug Finding Tools

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

We next examine these concepts in the context of the ESC/Java tool. Analysis with ESC/Java is an important step for our tools, and we can contrast them well by looking at what need they fill over the base ESC/Java bug finding ability.

3.1 Background: ESC/Java

The Extended Static Checker for Java (ESC/Java) [10] is a compile-time program checker that detects potential invariant violations. ESC/Java compiles the Java source code under test to a set of predicate logic formulae [10]. ESC/Java checks each method `m` in isolation, expressing as logic formulae the properties of the class to which the method belongs, as well as Java semantics. Each method call or invocation of a primitive Java operation in `m`'s body is translated to a check of the called entity's precondition followed by assuming the entity's post-condition. ESC/Java recognizes invariants stated in the Java Modeling Language

(JML) [13]. (We consider the ESC/Java2 system [2]—an evolved version of the original ESC/Java, which supports JML specifications and recent versions of the Java language.) In addition to the explicitly stated invariants, ESC/Java knows the implicit pre- and postconditions of primitive Java operations—for example, array access, pointer dereference, class cast, or division. Violating these implicit preconditions means accessing an array out-of-bounds, dereferencing null pointers, mis-casting an object, dividing by zero, etc. ESC/Java uses the Simplify theorem prover [7] to derive error conditions for a method. We use ESC/Java to derive abstract conditions under which the execution of a method under test may terminate abnormally. Abnormal termination means that the method would throw a runtime exception because it violated the precondition of a primitive Java operation. In many cases this will lead to a program crash as few Java programs catch and recover from unexpected runtime exceptions.

Like many other static analysis based bug finding systems, ESC/Java is language-level unsound (and therefore also user-level unsound): it can produce spurious error reports because of inaccurate modeling of the Java semantics. ESC/Java is also unsound for correctness: it may miss some errors—for example, because ESC/Java ignores all iterations of a loop beyond a fixed limit.

3.2 Language-Level Soundness: Program Execution Semantics

Language-level soundness is the lower bar for automatic analysis tools. An analysis that is unsound with respect to execution semantics may flag execution paths that can never occur, under any inputs or circumstances. ESC/Java uses such an analysis. In the absence of pre-conditions and post-conditions describing the assumptions and effects of called methods, ESC/Java analyzes each method in isolation without taking the semantics of other methods into account. For instance, in the following example, ESC/Java will report potential errors for `get0() < 0` and `get0() > 0`, although neither of these conditions can be true.

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

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

In Section 4.1 we describe how our tool Check 'n' Crash eliminates language-level unsoundness from ESC/Java warnings by compiling them to test cases. This enables us to confirm ESC/Java warnings by concrete program execution and suppress warnings we could not confirm. Check 'n' Crash could never generate a test case that confirms the above warning about method `meth` and would therefore never report such an language-level unsound case to the user.

3.3 User-Level Soundness: Informal Specifications

A user-level sound analysis has to satisfy not only language semantics but also user-level specifications. Thus, user-level soundness is generally impossible to

achieve for automated tools since user-level specifications are mostly informal. Common forms of user-level specifications are code comments, emails, or web pages describing the program. Often these informal specifications only exist in the developers' minds. It is clear that user-level soundness implies language-level soundness, since the users care only about bugs that can occur in real program executions. So the user-level sound bug reports are a subset of the language-level sound bug reports.

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

In Section 4.2 we describe how our tool DSD-Crasher tries to eliminate user-level unsoundness from ESC/Java warnings by inferring the preconditions of intended program behavior from actual program executions. This allows us to exclude cases that are not of interest to the user. In the above example we might be able to infer a precondition of `i > 0` for method `forPositiveInt(int i)`, which would allow ESC/Java to suppress the user-level unsound warning.

4 Turning ESC/Java into a Sound Tool for Automatic Bug Finding

Our two tools attempt to address the two levels of unsoundness exhibited by many static analysis tools like ESC/Java. Check 'n' Crash is a *static-dynamic* (SD) tool, which post-processes ESC/Java's output with a dynamic step. DSD-Crasher is a *dynamic-static-dynamic* (DSD) tool that adds a dynamic step at the beginning, feeding the results of this first dynamic step to the static-dynamic Check 'n' Crash.

4.1 Check 'n' Crash: Making ESC/Java Language-Level Sound

Check 'n' Crash [4] addresses the problem of ESC/Java language-level unsoundness. Figure 1 illustrates the key idea. Check 'n' Crash takes error conditions that ESC/Java infers from the testee, derives variable assignments that satisfy the error condition (using a constraint solver), and compiles them into concrete test cases that are executed with our JCrasher testing tool [3], to determine whether an error truly exists. Compared to ESC/Java alone, Check 'n' Crash's combination of ESC/Java with JCrasher eliminates language-level unsound warnings and improves the ease of comprehension of error reports through concrete Java counterexamples.

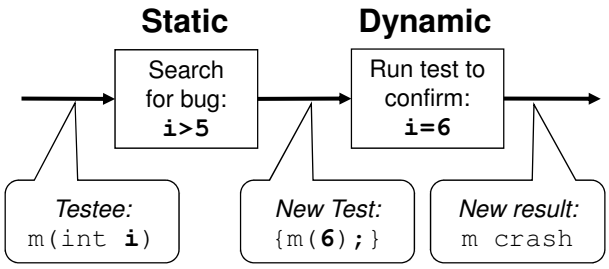

Fig. 1. Check 'n' Crash uses ESC/Java to statically check the testee for potential bugs. It then compiles ESC/Java's bug warnings to concrete test cases to eliminate those warnings that are unsound at the language level.

Check 'n' Crash takes as inputs the names of the Java files under test. It invokes ESC/Java, which derives error conditions. Check 'n' Crash takes each error condition as a constraint system over a method `m`'s parameters, the object state on which `m` is executed, and other state of the environment. Check 'n' Crash extends ESC/Java by parsing and solving this constraint system. A solution is a set of variable assignments that satisfy the constraint system. Reference [4] discusses in detail how we process constraints over integers, arrays, and reference types in general. Once the variable assignments that cause the error are computed, Check 'n' Crash uses JCrasher to compile some of these assignments to JUnit [1] test cases. The test cases are then executed under JUnit. If the execution does not cause an exception, then the variable assignment was a false positive: no error actually exists. If the execution does result in the error predicted by ESC/Java, an error report is generated by Check 'n' Crash.

4.2 DSD-Crasher: Improving ESC/Java's User-Level Soundness

DSD-Crasher [5] attempts to address the user-level unsoundness of ESC/Java and Check 'n' Crash. This requires recognizing "normal" program inputs. Such informal specifications cannot generally be derived, therefore our approach is necessarily heuristic. DSD-Crasher employs the Daikon tool [9] to infer likely program invariants from an existing test suite. The results of Daikon are exported as JML annotations [13] that are used to guide Check 'n' Crash. Figure 2 illustrates the processing steps of the tool.

Daikon [9] tracks a testee's variables during execution and generalizes their observed behavior to invariants—preconditions, postconditions, and class invariants. Daikon instruments a testee, executes it (for example, on an existing test suite or during production use), and analyzes the produced execution traces. At each method entry and exit, Daikon instantiates some three dozen invariant templates, including unary, binary, and ternary relations over scalars, and relations over arrays (relations include linear equations, orderings, implication, and disjunction) [9, 17]. For each invariant template, Daikon tries several combinations

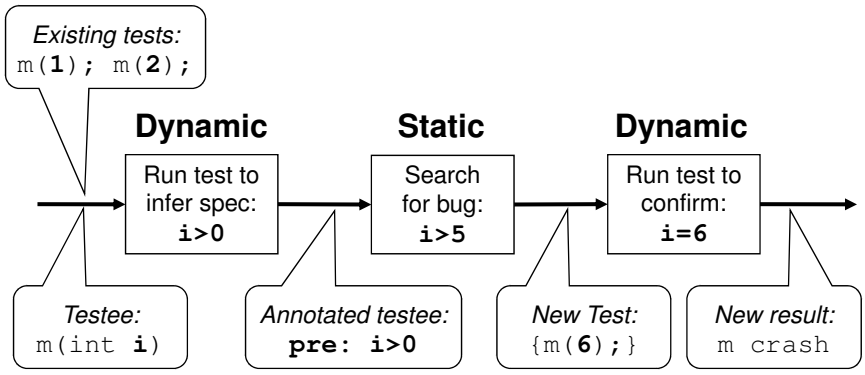

Fig. 2. DSD-Crasher adds a dynamic analysis step at the front to infer the intended program behavior from existing test cases. It feeds inferred invariants to Check 'n' Crash by annotating the testee. This enables DSD-Crasher to suppress bug warnings that are unsound at the user level.

of method parameters, method results, and object state. For example, it might propose that some method `m` never returns null. It later ignores those invariants that are refuted by an execution trace—for example, it might process a situation where `m` returned null and it will therefore ignore the above invariant. So Daikon summarizes the behavior observed in the execution traces as invariants and generalizes it by proposing that the invariants might hold in all other executions as well. Daikon can annotate the testee's source code with the inferred invariants as JML preconditions, postconditions, and class invariants. Daikon-inferred invariants are not trivially amenable to automatic processing, requiring some filtering and manipulation (e.g., for internal consistency according to the JML behavioral subtyping rules, see [6]).

In DSD-Crasher we chose to ignore Daikon-inferred invariants as requirements and only use them as assumptions. That is, we deliberately avoid searching for cases in which the method under test violates some Daikon-inferred precondition of another method it calls. (This would be against the spirit of the tool, as it would increase its user-level unsoundness, by producing extra error reports for violations of preconditions that were only heuristically derived.) Instead, we use Daikon-inferred invariants as assumptions. This restricts the number of legal program executions.

5 A Small Case Study

We next discuss in detail a small case study and present examples that illustrate the capabilities of each of our tools. Our test subject is Groovy, an open source scripting language that compiles to Java bytecode.

5.1 Setting

Table 1. Groovy testees.

	Total	Analyzed
Top-level classes	171	105
Top-level interfaces	11	8
Non-commented source statements (NCSS)	11k	5k
Public non-abstract methods and constructors	1240	757
Other methods and constructors	441	195

We used the Groovy 1.0 beta 1 version. Table 1 gives an overview of Groovy's main application classes and the subset used in our experiments. Its main application sources contain some eleven thousand non-commented source statements (NCSS) in 182 top-level classes and interfaces. We excluded any testees that led to processing problems in our tools. These were mainly low-level Groovy AST classes. The biggest processing problem was the Daikon component of DSD-Crasher running out of heap space. (We were using Daikon version 4.1.6, which is not the latest version.) The resulting set of testees consisted of 113 top-level types with a total of some five thousand NCSS. These declare a total of 952 methods and constructors, of which our testing tools analyze 757. (We analyze all public non-abstract methods declared by the testees as well as public constructors declared by non-abstract testees.) We used 603 of the unit test cases that came with the tested Groovy version. (The source code of the application and the unit tests are available from http://groovy.codehaus.org/ .) All experiments were conducted on a 2 GHz AMD Athlon 64 X2 dual core 3800+ with 4 GB of RAM, of which 1.5 GB were available for each experiment.

We believe that Groovy is a very representative test application for our kind of analysis: it is a medium-size, third party application. Importantly, its test suite was developed completely independently of our evaluation by the application developers, for regression testing and not for the purpose of yielding good Daikon invariants.

5.2 Baseline: JCrasher

We include JCrasher in the experiment in order to provide a baseline with a dynamic tool. This serves to highlight the advantages of sophisticated static analyses. JCrasher picks its test cases at random, without analyzing the bodies of the methods under test. It examines the type information of a set of Java classes and constructs code fragments that will create instances of different types to test the behavior of public methods under random data. For instance, to test a method, JCrasher will attempt to create sample objects of the receiver type and of each of the argument types. JCrasher begins with a set of types for which

it knows how to create instances—e.g., primitive types or types with public no-argument constructors. This set is expanded by finding constructors and methods that accept as arguments only types that JCrasher knows how to create, and the process continues until the space is exhaustively explored up to a certain invocation depth. Once the size of the space is known, test cases are selected at random, up to a user-defined number.

JCrasher attempts to detect bugs by causing the program under test to "crash", i.e., to throw an undeclared runtime exception indicating an illegal class cast, division by zero, or an illegal array expression. The output of JCrasher is a set of test cases for JUnit (a popular Java unit testing tool) [1]. We only include those JCrasher reports where a method under test throws such an exception directly (by performing an illegal Java language operation such as an illegal class cast).

Table 2 shows the total runtime, which includes the steps of compiling the testee classes (three seconds), generating test cases (about 20 seconds), compiling these test cases (about 40 seconds), and running them with our JUnit extensions (about 30 seconds). We let JCrasher generate test cases that randomly combine methods and constructors up to a depth of two. We also limited JCrasher to generate 100 thousand test cases per run. JCrasher picked these test cases from some $7 * 10^8$ available cases. (This is not an unusually large number: on the entire Groovy testee, JCrasher has $4 * 10^{12}$ test cases to pick from.) In three out of five runs JCrasher got lucky and reported an array index out of bounds exception in the last statement of the following parser look-ahead method `org.codehaus.groovy.syntax.lexer.AbstractCharStream.la(int)` when passed -1.

```
public char la(int k) throws IOException {
  if (k > buf.length)
    throw new LookAheadExhaustionException(k);
  int pos = this.cur + k - 1;
  pos %= buf.length;
  if (pos == this.limit) {
    this.buf[pos] = nextChar();
    ++this.limit;
    this.limit %= buf.length;
  }
  return this.buf[pos];
}
```

Clearly this report is language-level sound (like all JCrasher reports), as we observed an actual runtime exception. On the other hand it is likely that this report is not user-level sound since look-ahead functions are usually meant to be called with a non-negative value.

5.3 Check 'n' Crash

For this and the DSD-Crasher experiment we used ESC/Java2 version 2.08a, set the Simplify timeout to one minute, limited ESC/Java to generate ten warnings

Table 2. Experience with running different automatic bug finding tools on Groovy. ESC/Java warnings may be language-level unsound. Each report generated by JCrasher, Check 'n' Crash, and DSD-Crasher is backed by an actual test case execution and therefore guaranteed language-level sound.

	Runtime [min:s]	ESC/Java warnings	Generated test cases	Reports confirmed by test cases
JCrasher	1:40	n/a	100,000	0.6
Check 'n' Crash	2:17	51	439	7.0
DSD-Crasher	10:31	47	434	4.0

per method under test, and configured ESC/Java to only search for potential runtime exceptions in public methods and constructors, stemming from illegal class cast, array creation and access, and division by zero. Table 2 shows that ESC/Java produced 51 reports. By manual inspection we classified 14 as language-level unsound and 32 as language-level sound (we hand-wrote eight test cases to convince ourselves of non-trivial sound cases.) We could not classify the remaining five warnings within three minutes each due to their complex control flow. The latter cases are the most frustrating to inspect since several minutes of investigation might only prove that the bug finding tool produced a spurious report. Of the 32 language-level sound warnings Check 'n' Crash could confirm seven. The remaining 24 warnings would require to generate more sophisticated test cases than currently implemented by Check 'n' Crash, supporting method call sequences and generating custom sub-classes that produce bug inducing behavior not found in existing sub-classes.

To our surprise ESC/Java did not produce a warning that would correspond to the runtime exception discovered by JCrasher. Instead it warned about a potential division by zero in the earlier statement `pos %= buf.length`. This warning is language-level unsound, though, since `buf.length` is never zero. `buf` is a private field, all constructors set it to an array of length five, and there are no other assignments to this field. This case is representative of the language-level unsound ESC/Java warnings we observed: a few methods access a private field or local variable and all of these accesses maintain a simple invariant. ESC/Java misses the invariant since it analyzes each method in isolation. When commenting out this line, ESC/Java's analysis reaches the final statement of the method and generates a warning corresponding to JCrasher's finding and Check 'n' Crash confirms this warning as language-level sound.

5.4 DSD-Crasher

For the 603 Groovy test cases Daikon gathers some 600 MB of execution traces, which it distills to 3.6 MB of compressed invariants. Of the total runtime, Daikon took 88 seconds to monitor the existing test suite, 204 seconds to infer invariants from the execution traces, and 130 seconds to annotate the testee sources with

the derived invariants. The Check 'n' Crash component of DSD-Crasher used the remaining time.

In our working example, Daikon derived several preconditions and class invariants, including `k >= 1` and `this.cur >= 0`, for the look-ahead method described above. This supports our initial estimate that JCrasher and Check 'n' Crash reported a user-level unsound warning about passing a negative value to this method. The remainder of this example requires a modified version of Daikon since the method under test implements an interface method. Daikon can produce a contradictory invariant in this case—see [6] for a detailed discussion of dealing correctly with JML behavioral subtyping. For this example we manually added a precondition of `false` to the interface method declaration. When we again comment out the line `pos %= buf.length;` (but re-using the previously derived invariants, including `k >= 1` and `this.cur >= 0`) ESC/Java reaches the offending statement but uses the derived precondition to rule out the case. Thus, ESC/Java no longer produces the user-level unsound warning and DSD-Crasher does not produce a corresponding report.

6 Related Work

There is clearly an enormous amount of work in the general areas of test case generation and program analysis. We discuss representative recent work below.

There are important surveys that concur with our estimate that an important problem is not just reporting potential errors, but minimizing false positives so that inspection by humans is feasible. Rutar et al. [19] evaluate five tools for finding bugs in Java programs, including ESC/Java 2, FindBugs [11], and JLint. The number of reports differs widely between the tools. For example, ESC reported over 500 times more possible null dereferences than FindBugs, 20 times more than JLint, and six times more array bounds violations than JLint. Overall, Rutar et al. conclude: "The main difficulty in using the tools is simply the quantity of output."

AutoTest by Meyer et al. is a closely related automatic bug finding tool [16]. It targets the Eiffel programming language, which supports invariants at the language level in the form of contracts [15]. AutoTest generates random test cases like JCrasher, but uses more sophisticated test selection heuristics and makes sure that generated test cases satisfy given testee invariants. It can also use the given invariants as its test oracle. Our tools do not assume existing invariants since, unlike Eiffel programmers, Java programmers usually do not annotate their code with formal specifications.

The commercial tool Jtest [18] has an automatic white-box testing mode that generates test cases. Jtest generates chains of values, constructors, and methods in an effort to cause runtime exceptions, just like our approach. The maximal supported depth of chaining seems to be three, though. Since there is little technical documentation, it is not clear to us how Jtest deals with issues of representing and managing the parameter-space, classifying exceptions as errors or invalid tests, etc. Jtest does, however, seem to have a test planning approach,

employing static analysis to identify what kinds of test inputs are likely to cause problems.

Xie and Notkin [20] present an iterative process for augmenting an existing test suite with complementary test cases. They use Daikon to infer a specification of the testee when executed on a given test suite. Each iteration consists of a static and a dynamic analysis, using Jtest and Daikon. In the static phase, Jtest generates more test cases, based on the existing specification. In the dynamic phase, Daikon analyzes the execution of these additional test cases to select those which violate the existing specification—this represents previously uncovered behavior. For the following round the extended specification is used. Thus, the Xie and Notkin approach is also a DSD hybrid, but Jtest's static analysis is rather limited (and certainly provided as a black box, allowing no meaningful interaction with the rest of the tool). Therefore this approach is more useful for a less directed augmentation of an existing test suite aiming at high testee coverage—as opposed to our more directed search for fault-revealing test cases.

7 Conclusions

We discussed our thoughts on combinations of static and dynamic reasoning for bug detection, and presented our experience with our tools, JCrasher, Check 'n' Crash, and DSD-Crasher. We argued that static and dynamic analyses form a continuum and that a "sound for correctness"/"sound for incorrectness" terminology is more illuminating than other conventions in the area. We believe that tools that are sound for incorrectness (i.e., complete for correctness) will gain ground in the future, in the entire range of static and dynamic analyses.

Our DSD-Crasher, Check 'n' Crash, and JCrasher implementations are available in source and binary form at `http://code.google.com/p/check-n-crash/` and `http://code.google.com/p/jcrasher/`

Acknowledgments

We thank Tao Xie who offered extensive comments and contributed to early discussions about the generation of test cases from Daikon invariants. We gratefully acknowledge support by the NSF under grant CCR-0238289.

References

[1] K. Beck and E. Gamma. Test infected: Programmers love writing tests. *Java Report*, 3(7):37–50, July 1998.

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

[3] C. Csallner and Y. Smaragdakis. JCrasher: An automatic robustness tester for Java. *Software—Practice & Experience*, 34(11):1025–1050, Sept. 2004.

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

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

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

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

[8] D. Engler and M. Musuvathi. Static analysis versus software model checking for bug finding. In *Proc. 5th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI)*, pages 191–210. Springer, Jan. 2004.

[9] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE Transactions on Software Engineering*, 27(2):99–123, Feb. 2001.

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

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

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

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

[14] T. Lindahl and K. Sagonas. Practical type inference based on success typings. In *Proc. 8th ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming (PPDP)*, pages 167–178. ACM, 2006.

[15] B. Meyer. *Object-Oriented Software Construction*. Prentice Hall PTR, 2nd edition, 1997.

[16] B. Meyer, I. Ciupa, A. Leitner, and L. Liu. Automatic testing of object-oriented software. In *Proc. 33rd Conference on Current Trends in Theory and Practice of Computer Science (SOFSEM)*. Springer, Jan. 2007. To appear.

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

[18] Parasoft Inc. Jtest. http://www.parasoft.com/, Oct. 2002. Accessed Mar. 2007.

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

[20] T. Xie and D. Notkin. Tool-assisted unit test selection based on operational violations. In *Proc. 18th IEEE International Conference on Automated Software Engineering (ASE)*, pages 40–48. IEEE Computer Society, Oct. 2003.