DSD-Crasher: A Hybrid Analysis Tool for Bug Finding

CHRISTOPH CSALLNER
Georgia Institute of Technology
YANNIS SMARAGDAKIS
University of Oregon
TAO XIE
North Carolina State University

DSD-Crasher is a bug finding tool that follows a three-step approach to program analysis:

D. Capture the program's intended execution behavior with dynamic invariant detection. The derived invariants exclude many unwanted values from the program's input domain.

S. Statically analyze the program within the restricted input domain to explore many paths.

D. Automatically generate test cases that focus on reproducing the predictions of the static analysis. Thereby confirmed results are feasible.

This three-step approach yields benefits compared to past two-step combinations in the literature. In our evaluation with third-party applications, we demonstrate higher precision over tools that lack a dynamic step and higher efficiency over tools that lack a static step.

Categories and Subject Descriptors: D.2.4 [**Software Engineering**]: Software/Program Verification—*formal methods, reliability*; D.2.5 [**Software Engineering**]: Testing and Debugging—*testing tools*; I.2.2 [**Artificial Intelligence**]: Automatic Programming—*program verification*

General Terms: Reliability, Verification

Additional Key Words and Phrases: Automatic testing, bug finding, dynamic analysis, dynamic invariant detection, extended static checking, false positives, static analysis, test case generation, usability

1. INTRODUCTION

Dynamic program analysis offers the semantics and ease of concrete program execution. Static analysis lends itself to obtaining generalized properties from the program text. The need to combine the two approaches has been repeatedly stated in the software engineering community [Young 2003; Ernst 2003; Xie and Notkin 2003; Beyer et al. 2004; Csallner and Smaragdakis 2005]. In this article, we present DSD-Crasher: a bug-finding tool that uses dynamic analysis to infer likely program invariants, explores the space defined by these invariants exhaustively through static

Authors' addresses: csallner@gatech.edu, yannis@cs.uoregon.edu, xie@csc.ncsu.edu This is a revised and extended version of [Csallner and Smaragdakis 2006a], presented at ISSTA 2006 in Portland, Maine, and also contains material from [Smaragdakis and Csallner 2007].

analysis, and finally produces and executes test cases to confirm that the behavior is observable under some real inputs and not just due to overgeneralization in the static analysis phase. Thus, our combination has three steps: dynamic inference, static analysis, and dynamic verification (*DSD*).

More specifically, we employ the Daikon tool by Ernst et al. [2001] to infer likely program invariants from an existing test suite. The results of Daikon are exported as JML annotations [Leavens et al. 1998] that are used to guide our Check 'n' Crash tool [Csallner and Smaragdakis 2005]. 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). Check 'n' Crash employs the ESC/Java static analysis tool by Flanagan et al. [2002], applies constraint-solving techniques on the ESC/Java-generated error conditions, and produces and executes concrete test cases. The exceptions produced by the execution of generated test cases are processed in a way that takes into account which methods were annotated by Daikon, for more accurate error reporting. For example, a `NullPointerException` is not considered a bug if thrown by an un-annotated method, instead of an annotated method; otherwise, many false bug reports would be produced: ESC/Java produces an enormous number of warnings for potential `NullPointerException`s when used without annotations [Rutar et al. 2004].

Several past research tools follow an approach similar to ours, but omit one of the three stages of our analysis. Check 'n' Crash is a representative of a static-dynamic (SD) approach. There are several representatives of a DD approach, with the closest one (because of the concrete techniques used) being the Eclat tool by Pacheco and Ernst [2005]. Just like our DSD approach, Eclat produces program invariants from test suite executions using Daikon. Eclat also generates test cases and disqualifies the cases that violate inferred preconditions. Nevertheless, there is no static analysis phase to exhaustively attempt to explore program paths and yield a directed search through the test space. Instead, Eclat's test case generation is largely random. Finally, a DS approach is implemented by combinations of invariant detection and static analysis. A good representative, related to our work, is the Daikon-ESC/Java (DS) combination of Nimmer and Ernst [2002a].

The benefit of DSD-Crasher over past approaches is either in enhancing the ability to detect bugs, or in limiting false bug warnings.[1] For instance, compared to Check 'n' Crash, DSD-Crasher produces more accurate error reports with fewer false bug warnings. Check 'n' Crash is by nature local and intra-procedural when no program annotations are employed. As the Daikon-inferred invariants summarize actual program executions, they provide assumptions on correct code usage. Thus, DSD-Crasher can disqualify illegal inputs by using the precondition of the method under test to exclude cases that violate common usage patterns. As a secondary benefit, DSD-Crasher can concentrate on cases that satisfy called methods' preconditions. This increases the chance of returning from these method calls normally and reaching a subsequent problem in the calling method. Without preconditions, Check 'n' Crash is more likely to cause a crash in a method that is called by the

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

tested method before the subsequent problematic statement is reached. Compared to the Eclat tool, DSD-Crasher can be more efficient in finding more bugs because of its deeper static analysis, relative to Eclat's mostly random testing.

To demonstrate the potential of DSD-Crasher, we applied it to medium-size third-party applications (the Groovy scripting language and the JMS module of the JBoss application server). We show that, under controlled conditions (e.g., for specific kinds of errors that match well the candidate invariants), DSD-Crasher is helpful in removing false bug warnings relative to just using the Check 'n' Crash tool. Overall, barring engineering hurdles, we found DSD-Crasher to be an improvement over Check 'n' Crash, provided that the application has a regression test suite that exercises exhaustively the functionality under test. At the same time, the approach can be more powerful than Eclat, if we treat the latter as a bug finding tool. The static analysis can allow more directed generation of test cases and, thus, can uncover more errors in the same amount of time.

2. PHILOSOPHY AND MOTIVATION

We next discuss the main principles of our approach, which concentrates on reducing the rate of false bug warnings, at the expense of reporting fewer bugs. We then argue why a dynamic-static-dynamic combination yields benefits in a general setting, beyond our specific tools.

2.1 Terminology: 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 in terms of *soundness* and *completeness*. 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 $provable(p) \Rightarrow true(p)$ and complete iff $true(p) \Rightarrow provable(p)$.

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 view a static checker as a system for proving the existence of errors, then it is "sound" iff reporting an error means it is a true error and "complete" iff all errors in programs result in error reports. In contrast, if we view the static checker as a system for proving correctness, then it is "sound" iff passing the program means there are no errors (i.e., iff all incorrect programs produce an error report—what we called before "complete") and "complete" iff all correct programs result in no error (i.e., reporting an error means that one exists—what we called before "sound").

The interesting outcome of this duality 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 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 [Jackson and Rinard 2000]. This is unsatisfactory—e.g., it 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 [2004]). 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.

2.2 Why Prove Programs Incorrect?

Ensuring that a program is correct is the Holy Grail of program construction. Therefore analyses that are sound for correctness (e.g., static type systems) have been popular, even if limited. Nevertheless, for all interesting properties, soundness for correctness implies that the analysis has to be pessimistic and reject 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.

The conservativeness of static analysis has an impact on how it can be used in a software development cycle. 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 distinguish between a false warning and a true bug, and 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 [Flanagan et al. 2002; Engler and Musuvathi 2004; Hovemeyer and Pugh 2004] attempt to find program defects. In their assessment of the applicability of ESC/Java, Flanagan et al. [2002] write:

> [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.

This conclusion is also supported by the findings of other researchers, as we discuss in Section 8. Notably, Rutar et al. [2004] examine ESC/Java2, among other analysis tools, and conclude that it can produce many spurious warnings when used without context information (method annotations). One specific problem, which we revisit in later sections, is that of ESC/Java's numerous warnings for `NullPointerExceptions`. For five testees with a total of some 170 thousand non-commented source statements, ESC/Java warns of a possible null dereference over nine thousand times. 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 systems). Nevertheless, for analyses performed by third parties, it is more important to produce error reports in which the user can have high confidence or even certainty. This is the goal of our work. We attempt to increase the soundness of existing analyses by combining them in a way that reduces false error reports.

2.3 Research Question

The ultimate goal we would like to measure DSD-Crasher against is a *fully automated* tool for *modern object-oriented* languages that finds *bugs* but produces *no false bug warnings*. A fully automated bug finding tool should require zero interaction with the software developer using the tool. In particular, using the bug finding tool should not require any manual efforts to write additional specifications or test cases. The tool should also not require manual inspection of the produced bug warnings.

The goal of eliminating false bug warnings is complicated because the notion of a bug depends on interpretation. Furthermore, whether a program behavior constitutes a bug depends not only on the program state but also on the validity of inputs. Initially, no program behavior constitutes a bug. Only specifications (implicit or explicit) allow us to distinguish between an expected behavior and a bug. In practice, many implicit specifications exist, e.g., in the form of pre- and postconditions. A common precondition of object-oriented programs is that null is never a legal input value, unless explicitly stated in a code comment. A common postcondition is that a public method should not terminate by throwing a class cast exception. Beyond such general properties, most specifications are very specific, capturing the intended semantics of a given method.

Below we use *pre* and *post* when referring to satisfying pre- and postcondition, respectively.

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

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

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

In our work, we concentrate on bugs involving only primitive language operations (such as array accesses, dynamic type errors, and null dereferences). The same approach can likely generalize to violations of arbitrary user specifications. Nevertheless, our goal of full automation influences our focus: since most programs in practice do not have explicit formal specifications, we concentrate on implicit specifications in the target language.

In terms of preconditions, we analyze a program on a *per-public-method* basis and try to infer which inputs are valid with subjective *local* reasoning. This means that we consider an input to be valid if manual inspection reveals no program comments prohibiting it, if invariants of the immediately surrounding program context (e.g., class invariants) do not disqualify the input, and if program values produced during actual execution seem (to the human inspector) consistent with the input. We do not, however, try to confirm the validity of a method's input by producing whole-program inputs that give rise to it. In other words, we consider the *program as a library*: We assume that its public methods can be called for any values not specifically disallowed, as opposed to only values that can arise during whole-program executions with valid inputs to the program's `main` method. This view of "program as library" is common in modern development environments, especially in the Java or .Net world, where code can be dynamically loaded and executed in a different environment. Coding guidelines for object-oriented languages often emphasize that public methods are an interface to the world and should minimize the assumptions on their usage.[2] Furthermore, this view is convenient for experimentation, as it lets us use modules out of large software packages, without worrying about the scalability of analyzing (either automatically or by hand) the entire executable program. Finally, the per-method checking is defensive enough to withstand most changes in the assumptions of how a class is used, or what are valid whole-program inputs. Over time such assumptions are likely to change, while the actual method implementation stays the same. Examples include reusing a module as part of a new program or considering more liberal external environments: buffer overflows were promoted in the last two decades from obscure corner case to mission-critical bugs.

In an earlier paper [Smaragdakis and Csallner 2007] we introduced the terms *language-level soundness* and *user-level soundness*, which we also use in this article. A tool offers language-level sound bug warnings, if the error can be reproduced for *some* input to the method, regardless of the method's precondition—i.e., the program model used by the analysis accurately captures the language's semantics. User-level soundness is a stronger notion and means that the warning reflects a bug that arises for *valid* method inputs, as determined by the local reasoning outlined above.

2.4 DSD Combinations

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

Figure 1 illustrates the main idea of our DSD combination. The first dynamic analysis step generalizes existing executions. This is a heuristic step, as it involves inferring abstract properties from specific instances. Nevertheless, a heuristic approach is our only hope for improving soundness for incorrectness. We want to make it more likely that a reported and reproducible error will not be dismissed by the programmer as "outside the intended domain of the method". If the "intended domain" of the method (i.e., the range of inputs that constitute possible uses) were known from a formal specification, then there would be no need for this step.

[2] For instance, McConnell [2004, chapter 8] writes "The class's public methods assume the data is unsafe, and they are responsible for checking the data and sanitizing it. Once the data has been accepted by the class's public methods, the class's private methods can assume the data is safe." Similarly, Meyer [1997, chapter 23] explicitly bases his guidelines of class design on the design of libraries.

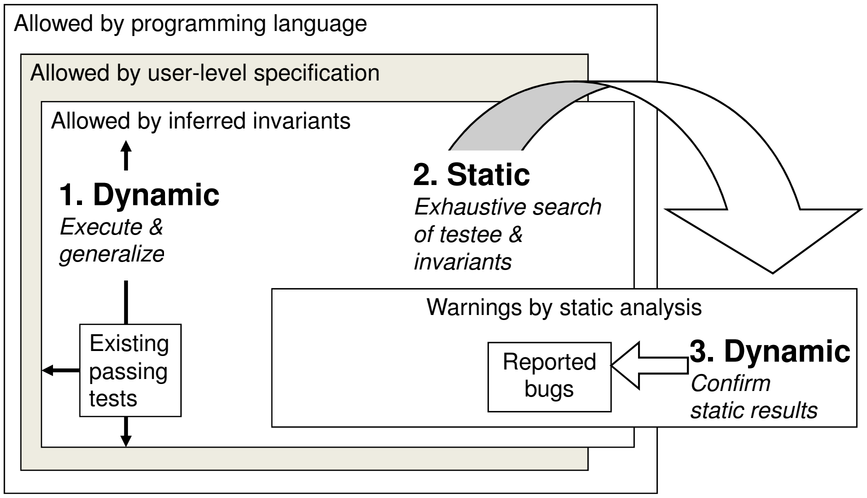
Allowed by programming language
Allowed by user-level specification
Allowed by inferred invariants
1. Dynamic
Execute & generalize
2. Static
Exhaustive search of testee & invariants
Existing passing tests
Warnings by static analysis
Reported bugs
3. Dynamic
Confirm static results

Fig. 1. The goal of the first dynamic step is to infer the testee's informal specification. The static step may generalize this specification beyond possible executions, while the final dynamic step will restrict the analysis to realizable problems. Each box represents a program domain. An arrow represents a mapping between program domains performed by the respective analysis. Shading should merely increase readability.

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

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

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

3. TOOLS BACKGROUND

Our three-step DSD-Crasher approach is based on two existing tools: Daikon (Section 3.1) and Check 'n' Crash (Section 3.3), which combines ESC/Java (Section 3.2) and the JCrasher test case generator [Csallner and Smaragdakis 2004]. This section presents background information on these tools.

3.1 Daikon: Guessing Invariants

Daikon [Ernst et al. 2001] 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) [Ernst et al. 2001; Nimmer and Ernst 2002b]. For each invariant template, Daikon tries several combinations of method parameters, method results, and object state. For example, it might propose that some method `m` never returns null. It later ignores those invariants that are refuted by an execution trace—for example, it might process a situation where `m` returned null and it will therefore ignore the above invariant. So Daikon summarizes the behavior observed in the execution traces as invariants and generalizes it by proposing that the invariants might hold in all other executions as well. Daikon can annotate the testee's source code with the inferred invariants as JML preconditions, postconditions, and class invariants [Leavens et al. 1998].

3.2 ESC/Java: Guessing Invariant Violations

The Extended Static Checker for Java (ESC/Java) by Flanagan et al. [2002] is a static program checker that detects potential invariant violations. ESC/Java recognizes invariants stated in the Java Modeling Language (JML) [Leavens et al. 1998]. (We use the ESC/Java2 system by Cok and Kiniry [2004]—an evolved version of the original ESC/Java, which supports Java 1.4 and JML specifications.) 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.

ESC/Java translates the Java source code under test to a set of predicate logic formulae [Flanagan et al. 2002]. 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 postcondition. 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 of Detlefs et al. [2003] to derive error conditions for a method.

ESC/Java is the main static analysis tool in our DSD combination. Our earlier discussion applies to ESC/Java: the tool is unsound (both for correctness and for incorrectness) yet it is powerful and exhaustive.

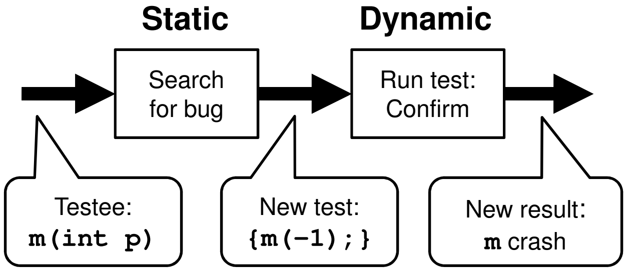

Fig. 2. Check 'n' Crash uses ESC/Java to statically check the testee for potential bugs. In this example, ESC/Java warns about a potential runtime exception in the analyzed method when passing a negative parameter (the ESC/Java warning is not shown). Check 'n' Crash then compiles ESC/Java's bug warnings to concrete test cases to eliminate those warnings that cannot be reproduced in actual executions. In this example, Check 'n' Crash produces a test case that passes -1 into the method and confirms that it throws the runtime exception ESC/Java has warned about.

3.3 Check 'n' Crash: Confirming Guessed Violations

Check 'n' Crash [Csallner and Smaragdakis 2005] is a tool for automatic bug finding. It combines ESC/Java and the JCrasher random testing tool [Csallner and Smaragdakis 2004]. Check 'n' Crash takes error conditions that ESC/Java infers from the testee, derives variable assignments that satisfy the error condition (using a constraint solver), and compiles them into concrete test cases that are executed with JCrasher to determine whether the error is language-level sound. Figure 2 shows the elements of Check 'n' Crash pictorially. Compared to ESC/Java alone, Check 'n' Crash's combination of ESC/Java with JCrasher eliminates spurious warnings and improves the ease of comprehension of error reports through concrete Java counterexamples.

Check 'n' Crash takes as inputs the names of the Java files under test. It invokes ESC/Java, which derives error conditions. Check 'n' Crash takes each error condition as a constraint system over a method `m`'s parameters, the object state on which `m` is executed, and other state of the environment. Check 'n' Crash extends ESC/Java by parsing and solving this constraint system. A solution is a set of variable assignments that satisfy the constraint system. [Csallner and Smaragdakis 2005] 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 [Beck and Gamma 1998] test cases. The test cases are then executed under JUnit. If the execution does not cause an exception, then the variable assignment was a false warning: no error actually exists. Similarly, some runtime exceptions do not indicate errors and JCrasher filters them out. For instance, throwing an `IllegalArgumentException` exception is the recommended Java practice for reporting illegal inputs. If the execution does result in one of the tracked exceptions, an error report is generated by Check 'n' Crash.

3.4 Check 'n' Crash Example

To see the difference between an error condition generated by ESC/Java and the concrete test cases output by Check 'n' Crash, consider the following method `swapArrays`, taken from a student homework solution.

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

The method's informal specification states that the method swaps the elements from `fstArray` to `sndArray` and vice versa. If the arrays differ in length the method should return without modifying any parameter. ESC/Java issues the following warning, which indicates that `swapArrays` might crash with an array index out-of-bounds exception.

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

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

Check 'n' Crash parses the error condition generated by ESC/Java and feeds the constraints to its constraint solvers. In our example, Check 'n' Crash creates two integer variables, `fstArray.length` and `sndArray.length`, and passes their constraints to the POOC integer constraint solver by Schlenker and Ringwelski [2002]. Then Check 'n' Crash requests a few solutions for this constraint system from its constraint solvers and compiles each solution into a JUnit [Beck and Gamma 1998] test case. For this example, the test case will create an empty and a random non-empty array. This will cause an exception when executed and JCrasher will process the exception according to its heuristics and conclude it is a language-level sound failure and not a false bug warning.

4. DSD-CRASHER: INTEGRATING DAIKON AND CHECK 'N' CRASH

We next describe the elements, scope, and ideas of DSD-Crasher.

4.1 Overview and Scope

DSD-Crasher works by first running a regression test suite over an application and deriving invariants using a modified version of Daikon. These invariants are then used to guide the reasoning process of Check 'n' Crash, by influencing the possible errors reported by ESC/Java. The constraint solving and test case generation applied to ESC/Java-reported error conditions remains unchanged. Finally, a slightly adapted Check 'n' Crash back-end runs the generated test cases, observes their execution, and reports violations. Figure 3 illustrates this process with an example.

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

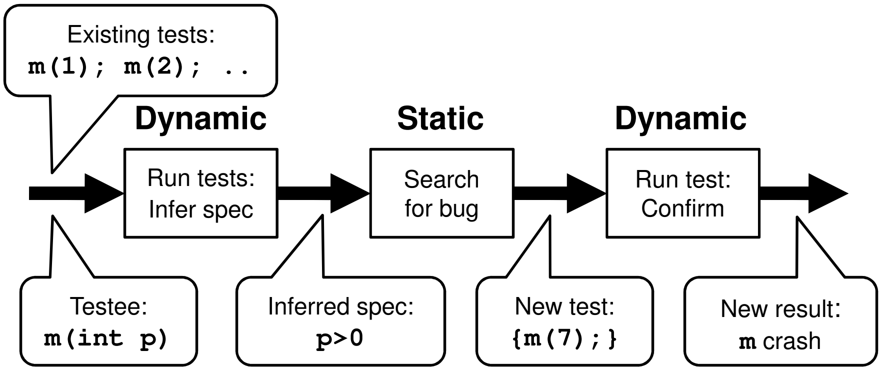

Fig. 3. DSD-Crasher adds a dynamic analysis step at the front of the pipeline, to infer the intended program behavior from existing test cases. It feeds inferred invariants to Check 'n' Crash by annotating the testee. This enables DSD-Crasher to suppress bug warnings that are not relevant to the intended uses of the program. In this example, the inferred invariant excludes negative input values. DSD-Crasher therefore does not produce a warning about -1 causing an exception as Check 'n' Crash did in figure 2.

to explore infinite loop paths. The errors that can be detected are of a few specific kinds [Csallner and Smaragdakis 2005]:

—Assigning an instance of a supertype to an array element.

—Casting to an incompatible type.

—Accessing an array outside its domain.

—Allocating an array of negative size.

—Dereferencing null.

—Division by zero.

These cases are statically detected using ESC/Java [Leino et al. 2000, chapter 4] but they also correspond to Java runtime exceptions (program crashes) that will be caught during JCrasher-initiated testing.

4.2 Benefits

The motivation of Section 2 applies to the specific features of our tools. DSD-Crasher yields the benefits of a DSD combination compared to just using its composite analysis. This can be seen with a comparison of DSD-Crasher with its predecessor and component tool, Check 'n' Crash. Check 'n' Crash, when used without program annotations, lacks interprocedural knowledge. This causes the following problems:

(1) Check 'n' Crash may produce spurious error reports that do not correspond to actual program usage. For instance, a method `forPositiveInt` under test may throw an exception if passed a negative number as an argument: the automatic testing part of Check 'n' Crash will ensure that the exception is indeed possible and the ESC/Java warning is not just a result of the inaccuracies of ESC/Java analysis and reasoning. Yet, a negative number may never be passed as input

to the method in the course of execution of the program, under any user input and circumstances. That is, an implicit precondition that the programmer has been careful to respect makes the Check 'n' Crash test case invalid. Precondition annotations help Check 'n' Crash eliminate such spurious warnings.

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

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

4.3 Design and Implementation of DSD-Crasher

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

In contrast, it is more reasonable to ignore Daikon-inferred invariants when used as requirements. In our earlier example, if we require that each caller of `foo` pass it a non-zero argument, we will produce several false bug warnings in case the invariant `i != 0` is not accurate. The main goal of DSD-Crasher, however, is to reduce false bug warnings and increase soundness for incorrectness. Thus, in DSD-Crasher, we chose to ignore Daikon-inferred invariants as requirements and only use them as assumptions. That is, we deliberately avoid searching for cases in which the method under test violates some Daikon-inferred precondition of another method it calls. Xie and Notkin [2003] partially follow a similar approach with Daikon-inferred invariants that are used to produce test cases.

4.3.2 *Inferred Invariants Excluded From Being Used.* DSD-Crasher integrates Daikon and Check 'n' Crash through the JML language. Daikon can output JML conditions, which Check 'n' Crash can use for its ESC/Java-based analysis. We exclude some classes of invariants Daikon would search for by default as we deemed them unlikely to be true invariants. Almost all of the invariants we exclude have to do with the contents of container structures viewed as sets (e.g., "the contents of array `x` are a subset of those of `y`"), conditions that apply to all elements of a container structure (e.g., "`x` is sorted", or "`x` contains no duplicates"), and ordering constraints among complex structures (e.g., "array `x` is the reverse of `y`"). Such complex invariants are very unlikely to be correctly inferred from the hand-written regression test suites of large applications, as in the setting we examine. We inherited (and slightly augmented) our list of excluded invariants from the study of the Jov tool of Xie and Notkin [2003]. The Eclat tool by Pacheco and Ernst [2005] excludes a similar list of invariants.

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

To perform the required integration, we also needed to make a more general change to Daikon. Daikon does not automatically ensure that inferred invariants support *behavioral subtyping* [Leavens et al. 1998]. Behavioral subtyping is a standard object-oriented concept that should hold in well-designed programs (e.g., see "subcontracting" in Design by Contract [Meyer 1997]). It dictates that a subclass object should be usable wherever a superclass object is. This means that the implementation of a subclass method (overriding method) should accept at least as many inputs as the implementation of a superclass method (overridden method), and for those inputs it should return values that the superclass could also return. In other words, an overriding method should have weaker preconditions than the preconditions of the method that it overrides. Additionally, for values satisfying the (possibly narrower) preconditions of the overridden method, its postconditions should also be satisfied by the overriding method. Daikon-inferred invariants can easily violate this rule: executions of the overriding method do not affect the invariants of the overridden method and vice versa. Therefore, we extended Daikon so that all behaviors observed for a subclass correctly influence the invariants of the superclass and vice versa. This change was crucial in getting invariants of sufficient consistency for ESC/Java to process automatically—otherwise we experienced contradictions in our experiments that prevented further automatic reasoning. The change is not directly related to the integration of Daikon and Check 'n' Crash, however. It is an independent enhancement of Daikon, valid for any use of the inferred invariants. We are in the process of implementing this enhancement directly on Daikon. We describe in a separate paper [Csallner and Smaragdakis 2006b] the exact algorithm for computing the invariants so they are consistent with the observed behaviors and as general as possible, while satisfying behavioral subtyping.

DSD-Crasher also modifies the Check 'n' Crash back-end: the heuristics used during execution of the generated test cases to decide whether a thrown exception is a likely indication of a bug and should be reported to the user or not. For methods with no inferred annotations (which were not exercised enough by the regression test suite) the standard Check 'n' Crash heuristics apply, whereas annotated methods are handled more strictly. Most notably, a `NullPointerException` is only considered a bug if the throwing method is annotated with preconditions. This is standard Check 'n' Crash behavior [Csallner and Smaragdakis 2005] and doing otherwise would result in many false error reports: as mentioned earlier, ESC/Java produces an enormous number of warnings for potential `NullPointerException`s when used without annotations [Rutar et al. 2004]. Nevertheless, for a Daikon-annotated method, we have more information on its desired preconditions. Thus, it makes sense to report even "common" exceptions, such as `NullPointerException`, if these occur within the valid precondition space. Therefore, the Check 'n' Crash runtime needs to know whether or not a method was annotated with a Daikon-inferred precondition. To accomplish this we extended Daikon's Annotate feature to produce a list of such methods. When an exception occurs at runtime, we check if the method on top of the call stack is in this list. One problem is that the call stack information at runtime omits the formal parameter types of the method that threw the exception. Thus, overloaded methods (methods with the same name but different argument types) can be a source for confusion. To disambiguate overloaded methods we use BCEL [Apache Software Foundation 2003] to process the bytecode of classes under test. Using BCEL we retrieve the start and end line number of each method and use the line number at which the exception occurred at runtime to determine the exact method that threw it.

5. EVALUATING HYBRID TOOLS

An interesting question is how to evaluate hybrid dynamic-static tools. We next discuss several simple metrics and how they are often inappropriate for such evaluation. This section serves two purposes. First, we argue that the best way to evaluate DSD-Crasher is by measuring the end-to-end efficiency of the tool in automatically discovering bugs (which are confirmed by human inspection), as we do in subsequent sections. Second, we differentiate DSD-Crasher from the Daikon-ESC/Java combination of Nimmer and Ernst [2002a].

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

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

Formal Specifications and Non-Standard Test Suites. DSD-Crasher aims at finding bugs in current, medium-sized, third-party software. These testees consist of thousands of lines of code and come with the original developers' test suites. They have been developed and are used by people other than us. The open-source programs we are aware of do not contain formal specifications. So for classifying bugs we are mainly relying on our subjective judgment, source code comments, and some external prose. This approach is explicitly dissimilar from previous evaluations like the one performed on Eclat by Pacheco and Ernst [2005], which mainly uses text book examples, student homeworks, or libraries for which formal specifications were written or already existed. Some of these testees seem to have large non-standard test suites, e.g., geared towards finding programming errors in student homework submissions. In contrast, typical third-party software is not formally specified and often comes with small test suites.

Coverage. Coverage metrics (e.g., statement or branch coverage in the code) are often used to evaluate the efficiency of analysis and testing tools. Nevertheless, coverage metrics may not be appropriate when using test suites automatically generated after static analysis of the code. Although some static analysis tools, such as Blast by Beyer et al. [2004] and SLAM by Ball [2003], have been adapted to generate tests to achieve coverage, static analysis tools generally exhaustively explore statements and branches but only report those that may cause errors. ESC/Java falls in this class of tools. The only reported conditions are those that may cause an error, although all possibilities are statically examined. Several statements and paths may not be exercised at all under the conditions in an ESC/Java report, as long as they do not cause an exception.

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

Precision and Recall. Nimmer and Ernst have performed some of the research closest to ours in combining Daikon and ESC/Java. Reference [Nimmer and Ernst 2002b] evaluates how well Daikon (and Houdini) can automatically infer program invariants to annotate a testee before checking it with ESC/Java. Reference [Nimmer and Ernst 2002a] also evaluates a Daikon-ESC/Java integration, concentrating more on automatically computed metrics.

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

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

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

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

6. EVALUATION

We want to explore two questions.

(1) Can DSD-Crasher eliminate some false bug warnings Check 'n' Crash produces? Reducing false bug warnings with respect to a static-dynamic tool like Check 'n' Crash was the main goal of DSD-Crasher.

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

This evaluation will not establish that DSD-Crasher is generally better than its competition (in all dimensions). DSD-Crasher trades improvements along the above dimensions with disadvantages on other dimensions, such as the number of bugs found or execution time. Instead, we would like to find evidence that a dynamic-static-dynamic approach like DSD-Crasher can provide improved results in some scenarios. Our goal is to provide motivation to use DSD-Crasher as part of a multi-tool approach to automated bug-finding. To investigate the first question, we are looking for cases in which Daikon-inferred invariants help DSD-Crasher rule out cases that likely violate implicit user assumptions. To investigate the second question, we are looking for bugs DSD-Crasher finds that elude a lightweight static analysis such as a mostly random bug search.

6.1 JBoss JMS and Groovy

JBoss JMS is the JMS module of the JBoss open-source J2EE application server (http://www.jboss.org/). It is an implementation of Sun's Java Message Service API [Hapner et al. 2002]. We used version 4.0 RC1, which consists of some five thousand non-comment source statements (NCSS).

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

We believe that Groovy is a very representative test application for our kind of analysis: it is a medium-size, third-party application. Importantly, its test suite was developed completely independently of our evaluation by the application developers, for regression testing and not for the purpose of yielding good Daikon invariants. JBoss JMS is a good example of a third-party application, especially appropriate for comparisons with Check 'n' Crash as it was a part of Check 'n' Crash's past evaluation [Csallner and Smaragdakis 2005]. Nevertheless, the existing test suite supplied by the original authors was insufficient and we had to supplement it ourselves to increase coverage for selected examples.

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

6.2 More Precise than Static-Dynamic Check 'n' Crash

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

6.2.1 *JBoss JMS.* Check 'n' Crash reported five cases, which include the errors reported earlier [Csallner and Smaragdakis 2005]. Two reports are false bug warnings. We use one of them as an example on how DSD-Crasher suppresses false bug warnings. Method `org.jboss.jms.util.JMSMap.setBytes` uses the potentially negative parameter `length` as the length in creating a new array. Calling `setBytes`

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

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

with a negative `length` parameter causes a `NegativeArraySizeException`.

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

We used unit tests that (correctly) call `setBytes` three times with consistent parameter values. DSD-Crasher's initial dynamic step infers a precondition that includes `requires length == daikon.Quant.size(value)`. This precondition implies that the `length` parameter cannot be negative. So DSD-Crasher's static step does not warn about a potential `NegativeArraySizeException` and DSD-Crasher does not produce this false bug warning.

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

Table I shows these results, as well as the runtime of the tools (confirming that DSD-Crasher has a realistic runtime). All tools are based on the current Check 'n' Crash implementation, which in addition to the published description [Csallner and Smaragdakis 2005] only reports exceptions thrown by a method directly called by the generated test case. This restricts Check 'n' Crash's reports to the cases investigated by ESC/Java and removes accidental crashes inside other methods called before reaching the location of the ESC/Java warning. Check 'n' Crash classic is the current Check 'n' Crash implementation. It suppresses all `NullPointerException`s, `IllegalArgumentException`s, etc. thrown by the method under test. DSD-Crasher is our integrated tool and reports any exception for a method that has a Daikon-inferred precondition. Check 'n' Crash relaxed is Check 'n' Crash classic but uses the same exception reporting as DSD-Crasher.

Check 'n' Crash relaxed reports the 11 DSD-Crasher exceptions plus 8 others. (These are 15 `NullPointerException`s plus the four other exceptions reported by Check 'n' Crash classic.) In 7 of the 8 additional exceptions, DSD-Crasher's ESC/Java step could statically rule out the warning with the help of the Daikon-derived invariants. In the remaining case, ESC/Java emitted the same warning, but the more complicated constraints threw off our prototype constraint solver. `(-1 - fromIndex) == size` has an expression on the left side, which is not yet supported by our solver. The elimination of the 7 false error reports confirms the

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

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

benefits of the Daikon integration. Without it, Check 'n' Crash has no choice but to either ignore potential `NullPointerException`-causing bugs or to report them, resulting in a high false bug warning rate.

6.3 More Efficient than Dynamic-Dynamic Eclat

We compare DSD-Crasher with Eclat by Pacheco and Ernst [2005], since it is the most closely related tool available to us. Specifically, Eclat also uses Daikon to observe existing correct executions and employs random test case generation to confirm testee behavior. This is not a perfect comparison, however: Eclat has a broader scope than DSD-Crasher (Section 4.1). So our comparison is limited to only one aspect of Eclat.

6.3.1 *ClassCastExceptions in JBoss JMS.* For the JBoss JMS experiment, the main difference we observed between DSD-Crasher and the dynamic-dynamic Eclat was in the reporting of potential dynamic type errors (`ClassCastExceptions`). The bugs reported by Csallner and Smaragdakis [2005] were `ClassCastExceptions`. (Most of the other reports concern `NullPointerExceptions`. Eclat produces 47 of them, with the vast majority being false bug warnings. DSD-Crasher produces 29 reports, largely overlapping the Eclat ones.)

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

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

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

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

a tool tuned to find crashes but to generate a range of tests.)

DSD-Crasher produces three distinct `ClassCastException` reports, which include the two cases presented in the past [Csallner and Smaragdakis 2005]. In the third case, class `JMSTypeConversions` throws a `ClassCastException` when the following method `getBytes` is called with a parameter of type `Byte[]` (note that the cast is to a "`byte[]`", with a lower-case "b").

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

6.3.2 *Groovy.* Table III compares DSD-Crasher with Eclat on Groovy. DSD-Crasher finds both of the Eclat reports. Both tools report several other cases, which we filtered manually to make the comparison feasible. Namely, we remove Eclat's reports of invariant violations, reports in which the exception-throwing method does not belong to the testees under test specified by the user, etc.

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

6.4 Summary of Benefits

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

Of course, inferred invariants are no substitute for human-supplied invariants. One should keep in mind that we focused on simple invariants produced by Daikon and eliminated more "ambitious" kinds of inferred invariants (e.g., ordering constraints on arrays), as discussed in Section 4.3. Even such simple invariants are sufficient for limiting the false bug warnings that Check 'n' Crash produces without any other context information.

6.5 Applicability and Limitations

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

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

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

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

Kinds of bugs caught. As discussed earlier, DSD-Crasher is a tool that aims for high degrees of automation. If we were to introduce explicit specifications, the tool could target any type of error, since it would be a violation of an explicit specification. Explicit specifications require significant human effort, however. Therefore, the intended usage mode of the tool only includes violations of implicit preconditions of language-level operations, which cause run-time exceptions, as described in Section 4.1. Thus, semantic errors that do not result in a program crash but pro-

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

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

duce incorrect results stay undetected. Furthermore, the thorough (due to the static analysis) but relatively local nature of DSD-Crasher means that it is much better for detecting violations of boundary conditions, than it is for detecting "deep" errors involving complex state and multiple methods. To be more precise, DSD-Crasher can detect bugs that hinge on prior state changes or interprocedural control- and data-flow, *only if these effects are captured well by the Daikon-inferred invariants*.

To illustrate this, we analyzed different versions of a subject (the Apache Xml Security module) from the software-artifact infrastructure repository (SIR), which is maintained by Do et al. [2005]. The repository contains several versions of a few medium-sized applications together with their respective test suites and seeded bugs. Several other research groups have used subjects from this repository to evaluate bug-finding techniques. Our results are summarized in Table IV. We found that most of the seeded bugs are too deep for DSD-Crasher to catch. Indeed, about half of the seeded bugs do not even affect ESC/Java's internal reasoning, independently of whether this reasoning leads to a bug warning or not. For instance, for version 1.0.4 of our subject, only 10 of the 20 seeded bugs affect *at all* the logical conditions computed during ESC/Java's analysis. The eventual ESC/Java warnings produced very rarely have any relevance to the seeded bug, even with a liberal "relevance" condition (local backward slice). DSD-Crasher does not manage to produce test cases for any of these warnings.

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

```
boolean _includeComments = true;
```

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

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

```
super(doc);
```

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

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

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

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

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

7. DISCUSSION: CODE-COVERAGE-ORIENTED TEST GENERATION

Several code-based test generation tools [Korel 1990; Gupta et al. 1998] aim at generating test inputs to achieve high code coverage metrics, such as statement coverage or branch coverage [Zhu et al. 1997]. Symbolic execution [King 1976; Clarke 1976] has been shown to be effective in generating test inputs to achieve high code coverage. For example, the Java PathFinder (JPF) model checker by Visser et al. [2000] has been extended to support symbolic execution [Khurshid et al. 2003; Visser et al. 2004; Anand et al. 2007]. A recent approach that has attracted attention is *concolic testing* [Godefroid et al. 2005; Sen et al. 2005; Cadar et al. 2006]: a combination of concrete and symbolic execution. Concolic testing tools explore a program path concretely for a value, while at the same time accumulating a "path condition": a set of symbolic constraints that an input needs to satisfy to follow the path. In case of a control-flow branch, a concolic execution tool attempts to solve the symbolic constraints to generate a value to also exercise the path *not* taken by the concrete execution. The power of concolic execution comes from the fact that the concrete execution has already demonstrated a way to solve many of these constraints, thus making the constraint solving problem often easier in practice.

Such symbolic and concolic execution tools can be generally described as *code-coverage-oriented* (or just *coverage-oriented*) testing tools. In principle, these tools also aim to discover bugs—the ultimate goal of all testing is to expose defects. Nevertheless, they take a different approach from tools like ESC/Java or DSD-Crasher. The domain-specific knowledge encoded in a coverage-oriented tool does not focus on what constitutes a bug, unlike an analysis that tries to find, e.g., null pointer exceptions, division by zero, array dereferences outside bounds, etc. Instead, the knowledge of a coverage-oriented tool is limited to the aspects that enhance coverage: understanding what constitutes a control-flow branch, deriving symbolic conditions, etc. This makes a coverage-oriented tool more general, in that it has no preconceived notion of a "bug", but also more limited, in that it will only discover a bug if following a control-flow path exposes the bug with high probability (i.e., for most, if not all, data values).

We believe that there are interesting insights in contrasting code-coverage-oriented tools and bug finding tools, such as DSD-Crasher and others described in Section 5. To expose them concretely, we analyzed the reports of DSD-Crasher presented in the previous section and examined which of the bugs would be found by the jCUTE [Sen and Agha 2006] concolic execution tool for Java and by JPF [Visser et al. 2004]. We next discuss our experience, as well as ways to expose bug-specific knowledge (e.g., the fact that a Java array throws an exception if referenced beyond its end) as control-flow branches. This would allow coverage-oriented tools to capture bugs that they currently do not detect.

Note that the comparison with jCUTE and JPF is qualitative rather than quantitative: although we did run jCUTE on the subject programs (Groovy and JBoss JMS) the handling was not automatic: we had to write explicit test drivers for each of the errors reported by DSD-Crasher. In several cases, our test drivers needed to be fairly contrived, in order to overcome engineering limitations in jCUTE and expose the problem in terms that jCUTE can analyze (e.g., transform `instanceOf` checks into checks on a numeric result returned by an oracle method). In essence, we tried to approximate what an ideal version of jCUTE or JPF would do in principle, beyond current specifics, and we base our discussion on that.

7.1 Errors Easily Exposed via Coverage

Some errors have little data-sensitivity and would be readily exposed by just covering the potentially erroneous statement. For instance, many of the errors that DSD-Crasher reports for JBoss JMS and Groovy are due to direct null pointer dereferencing of method arguments. In principle, a possible null dereference is value-sensitive and exposed by very specific executions of the offending statements. Nevertheless, both JPF and jCUTE can generate default null references for non-primitive-type arguments, causing these null-pointer exceptions to be thrown. This case is easy, exactly because the error is reproduced with a well-identified value. Even for a tool that concentrates on control-flow, covering a single known offending value is easy enough.

Some more of the errors reported by DSD-Crasher can be readily discovered with jCUTE or JPF. For instance, consider the example code shown in Section 6.3.1. This produces a class-cast exception within the true branch of an `instanceof` conditional check. The bug is exposed every time the statement causing it is covered. Although the current versions of jCUTE or JPF cannot handle the constraints related to `instanceof`, this is a matter of constraint solving, which is orthogonal to the coverage-oriented nature of the tools—they can both be extended to handle `instanceof` constraints, thus generating test inputs to expose the bug.

Class cast exceptions are generally fairly easy to reproduce semi-accidentally with coverage-oriented tools, since they are not particularly data-sensitive: any object of the wrong type will trigger the exception. For instance, consider one of the DSD-Crasher reports for class `Container` in JBoss JMS. Method `getContainer` throws a `ClassCastException` when called with an argument of type `java.lang.Object`.

```
public static Container getContainer(Object object) throws Throwable {
  Proxy proxy = (Proxy) object;//an exception thrown here
  return (Container) Proxy.getInvocationHandler(proxy);
}
```

The error does not always occur when the statement is covered: there is no problem when the method is called with an argument of type `Proxy`. Although the current versions of jCUTE or JPF do not instrument the Java library classes such as `java.lang.Object`, this is a technicality. Both tools could potentially expose this bug since they would create an object of type `java.lang.Object` as the method argument.

7.2 Errors Not Easily Exposed via Coverage

Several errors that DSD-Crasher targets are fairly data-sensitive and, thus, not exposed easily through simple increased code coverage. Good examples are arithmetic exceptions, negative array size exceptions, or array index out-of-bounds exceptions.

We examined the negative-array-size exceptions that DSD-Crasher detects for JBoss JMS and Groovy. The statements including these errors are not within any conditionals. For a concise example, class `Tuple` in Groovy throws a `NegativeArraySizeException` when the following method `subList` is called with two integer arguments of 1 and 0.

```
public List subList(int fromIndex, int toIndex) {
  int size = toIndex - fromIndex;
  Object[] newContent = new Object[size];//exception thrown here
  System.arraycopy(contents, fromIndex, newContent, 0, size);
  return new Tuple(newContent);
}
```

This DSD-Crasher warning reveals two serious bugs, one being within the specification of `java.util`. Groovy's `Tuple` overrides the `subList` method of `java.util.AbstractList`, which in turn implements the `subList` definition in `java.util.List`. The JavaDoc specifications of `List` and `AbstractList` conflict for this case of `fromIndex > toIndex` amongst each other and the implementation of `Tuple` conflicts with both. Specifically, `List` requires to throw an `IndexOutOfBoundsException`, its redefinition in `AbstractList` requires an `IllegalArgumentException`, and `Tuple` throws a `NegativeArraySizeException` (but none of these runtime exceptions are a subtype of another).

Neither jCUTE nor JPF can detect this error because there is no branching point in the methods for either tool to collect constraints in the path condition. Consequently, the tool assigns a default value (e.g., 0) to the integer-type arguments. Of course, the exception could be thrown by randomly selecting inputs, but the essence of the problem is that the analysis in jCUTE or JPF has no insight to guide it to construct inputs where `fromIndex` is greater than `toIndex`.

7.3 Exposing Data Conditions as Branches

Combining ideas from coverage-oriented tools and DSD-Crasher-like approaches seems quite promising. Perhaps surprisingly, it seems to also be very much in the spirit of both kinds of tools. DSD-Crasher already has a symbolic engine (in the form of ESC/Java) but it can probably benefit from the idea of concolic execution to increase its constraint-solving abilities. At the same time, coverage-oriented tools can benefit in their goal to increase code coverage by integrating knowledge about errors in language-level operations, such as arithmetic exceptions or array accesses out-of-bounds.

To do this, a coverage-oriented tool needs to understand values that violate operation preconditions, in addition to understanding branches. The duality of control and data-flow is well-known in the study of compilers: standard program transformations can map a data-flow property into a control-flow property, and vice versa. Interestingly, in the case of values that violate preconditions of language-level operations (as in most of the bugs DSD-Crasher finds), the properties are really control-flow properties to begin with: the program would throw an exception during its execution, which would change the flow of control. The only reason that code-coverage-oriented tools, such as jCUTE and JPF, do not detect these errors is that they do not recognize the possible execution branch.

To provide a concrete example, consider a statement such as:

```
f = i / j;
```

The division operation constitutes an implicit branch: it can throw a division-by-zero exception (arithmetic exception). An equivalent way to view the above statement is as:

```
if (j == 0) throw new ArithmeticException();
else f = i / j;
```

This simple observation leads to a somewhat counterintuitive conclusion. Code-coverage-oriented tools need to recognize language-level exceptional conditions to achieve true high coverage of all program paths. Implicit branches are as real as explicit ones, and exploring them enhances the bug detection capabilities of a testing tool. In short, adding knowledge about illegal arguments of low-level operations to a code-coverage-oriented tool is both effective and very much in the spirit of increasing code coverage. To our knowledge, except for Pex [Anand et al. 2008; Csallner et al. 2008], no current code-coverage-oriented tool follows this approach.

8. RELATED WORK

There is an enormous amount of work on automated bug-finding tools. We discuss representative recent work below. We deliberately include approaches from multiple research communities in our discussion. Particularly, we compare DSD-Crasher with tools from the testing, program analysis, and verification communities. We believe this is a valuable approach, since many tools produced by these closely related communities have overlapping goals, i.e., to find bugs. We also discuss our choice of component analyses from which we constructed DSD-Crasher. This should highlight that other analysis combinations are possible and may provide superior properties than our concrete instantiation in the DSD-Crasher tool.

8.1 Bug-Finding Tools and False Bug Warnings

The surveys of automated bug-finding tools conducted by Zitser et al. [2004], Wagner et al. [2005], and Rutar et al. [2004] concur with our estimate that an important problem is not just reporting potential errors, but minimizing false bug warnings so that inspection by humans is feasible. Zitser et al. [2004] evaluate five static analysis tools on 14 known buffer overflow bugs. They found that the tool with the highest detection rate (PolySpace) suffered from one false alarm per twelve lines of code. They conclude "[..] that while state-of-the-art static analysis tools like Splint and PolySpace can find real buffer overflows with security implications, warning rates are unacceptably high." Wagner et al. [2005] evaluate three automatic bug finding tools for Java (FindBugs by Hovemeyer and Pugh [2004], PMD, and QJ Pro). They conclude that "as on average two thirds of the warnings are false positives, the human effort could be even higher when using bug finding tools because each warning has to be checked to decide on the relevance of the warning." Rutar et al. [2004] evaluate five tools for finding bugs in Java programs, including ESC/Java2, FindBugs, and JLint. The number of reports differs widely between the tools. For example, ESC/Java2 reported over 500 times more possible null dereferences than FindBugs, 20 times more than JLint, and six times more array bounds violations than JLint. Overall, Rutar et al. conclude: "The main difficulty in using the tools is simply the quantity of output."

8.2 Bug-Finding Tools That Reduce Language-Level And User-Level False Bug Warnings

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

Symclat by d'Amorim et al. [2006] is a closely related tool that, like DSD-Crasher, uses the Daikon invariant detector to infer a model of the testee from existing test cases. Symclat uses Java PathFinder for its symbolic reasoning, which has different tradeoffs than ESC/Java, e.g., Java PathFinder does not incorporate existing JML specifications into its reasoning. Unlike our tools, Symclat has a broader goal of discovering general invariant violations. It appears to be less tuned towards finding uncaught exceptions than our tools since it does not seem to try to cover all control flow paths implicit in primitive Java operations as we discussed in Section 7.

Palulu by Artzi et al. [2006] derives method call sequence graphs from existing test cases. It then generates random test cases that follow the call rules encoded in the derived call graphs. Such method call graphs capture implicit API rules (e.g., first create a network session object, then send some initialization message, and only then call the testee method), which are essential in generating meaningful test cases. It would be interesting to integrate deriving such API rules into our first dynamic analysis step.

8.3 Bug-Finding Tools That Reduce Language-Level False Bug Warnings

Tools in this category use an overapproximating search to find as many bugs as possible. Additionally, they use some technique to reduce the number of false bug warnings, focusing on language-level unsound bug warnings. Our representative of this category is Check 'n' Crash [Csallner and Smaragdakis 2005].

Tomb et al. [2007] present a direct improvement over Check 'n' Crash by making their overapproximating bug search interprocedural (up to a user-defined call depth). The tool otherwise closely follows the Check 'n' Crash approach by generating test cases to confirm the warnings of their static analysis. On the other hand, their tool neither seems to incorporate pre-existing specifications (which provides an alternative source of interproceduralilty) nor address user-level unsound bug warnings.

Kiniry et al. [2006] motivate their recent extensions of ESC/Java2 similarly: "User awareness of the soundness and completeness of the tool is vitally important in the verification process, and lack of information about such is one of the most requested features from ESC/Java2 users, and a primary complaint from ESC/Java2 critics." They list several sources of unsoundness for correctness and incorrectness in ESC/Java2 including less known problems like Simplify silently converting arithmetic overflows to incorrect results. They propose a static analysis that emits warnings about potential shortcomings of the ESC/Java2 output, namely potentially missing bug reports and potentially unsound bug reports. On the bug detection side their analysis is only concerned with language-level soundness and does not worry about soundness with regard to user-level (and potentially informal) specifications like DSD-Crasher does. DSD-Crasher also provides a more extreme solution for language-level unsound bug reports as it only reports cases that are guaranteed to be language-level sound. We believe our approach is more suitable for automated bug finding since it provides the user with concrete test cases that prove the existence of offending behavior. On the other hand, DSD-Crasher only addresses the unsoundness of ESC/Java2 bug reports. On the sound-for-correctness side, DSD-Crasher would greatly benefit from such static analysis to reduce the possibility of missing real errors. DSD-Crasher needs such analysis even more than ESC/Java2 does, as it may miss sound bug reports of ESC/Java2 due to its limited constraint solving.

Several dynamic tools like the one by Xie and Notkin [2003] generate candidate test cases and execute them to filter out false error reports. Xie and Notkin [2003] 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 that violate the existing specification—this violation represents previously uncovered behavior. For the subsequent round, the extended specification is used. Thus, the approach by Xie and Notkin 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, their approach is more useful for a less directed augmentation of an existing test suite aiming at high testee coverage—as opposed to our more directed search for fault-revealing test cases.

Concolic execution (see Godefroid et al. [2005], Sen et al. [2005], Cadar et al. [2006], and Godefroid [2007]) uses concrete execution to overcome some of the limitations of symbolic execution, which goes back to King [1976] and Clarke [1976]. This makes it potentially more powerful than static-dynamic sequences like Check 'n' Crash. But unlike DSD-Crasher, concolic execution alone does not observe existing test cases and therefore does not address user-level soundness.

Systematic modular automated random testing (SMART) makes concolic execution more efficient by exploring each method in isolation [Godefroid 2007]. SMART summarizes the exploration of a method in pre- and postconditions and uses this summary information when exploring a method that calls a previously summarized method. DSD-Crasher also summarizes methods during the first dynamic analysis step in the form of invariants, which ESC/Java later uses for modular static analysis. DSD-Crasher would benefit from SMART-inferred method summaries for methods that were not covered by our initial dynamic analysis. SMART seems like a natural replacement for the SD-part (ESC/Java and JCrasher) of DSD-Crasher. Designing such a dynamic-concolic tool ("DC-Crasher") and comparing it with DSD-Crasher is part of our future work.

The commercial tool Jtest by Parasoft Inc. [2002] 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.

8.4 Alternative Component Analyses

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

8.4.1 *Inferring Specifications To Enable Reducing User-Level False Bug Warnings.* Daikon is not the only tool for invariant inference from test case execution, although it has pioneered the area and has seen the widest use in practice. For instance, Hangal and Lam [2002] present the DIDUCE invariant inference tool, which is optimized for efficiency and can possibly allow bigger testees and longer-running test suites than Daikon. Agitar Agitator [Boshernitsan et al. 2006], a commercial tool, also uses Daikon-like inference techniques to infer likely invariants (termed "observations") from test executions and suggests these observations to developers so that the developers can manually and selectively promote observations to assertions. Then Agitator further generates test cases to confirm or violate these assertions. Agitator requires manual effort in promoting observations to assertions in order to avoid false warnings of observation violations, whereas our tools concentrate on automated use.

Inferring method call sequence rules is another valuable approach for capturing implicit user assumptions. Whaley et al. [2002] present static and dynamic analyses that automatically infer over- and underapproximating finite state machines of method call sequences. Artzi et al. [2006] have used such finite state machines to generate test cases. Henkel et al. [2007] automatically infer *algebraic specifications* from program executions, which additionally include the state resulting from method call sequences. Algebraic specifications express relations between nested method calls, like *pop(push(obj)) == obj*, which makes them well-suited for specifying container classes. It is unclear, though, how this technique scales beyond container classes. Yet it would be very interesting to design an automated bug-finding tool that is able to process algebraic specifications and compare it with DSD-Crasher.

Taghdiri et al. [2006] offer a recent representative of purely static approaches that summarize methods into specifications. Such method summaries would help DSD-Crasher perform deeper interprocedural analysis in its overapproximating bug search component. Summarization approaches typically aim at inferring total specifications, though. So they do not help us in distinguishing between intended and faulty usage scenarios, which is key for a bug-finding tool as DSD-Crasher. Kremenek et al. [2006] infer partial program specifications via a combination of static analysis and expert knowledge. The static analysis is based on the assumption that the existing implementation is correct most of the time. The thereby inferred specifications helped them to correct and extend the specifications used by the commercial bug finding tool Coverity Prevent [Coverity Inc. 2003]. Expert knowledge is probably the most important source of good specifications, but also the most expensive one, because it requires manual effort. An ideal bug finding tool should combine as many specification sources as possible, including automated static and dynamic analyses.

8.4.2 *The Core Bug Search Component: Overapproximating Analysis For Bug-Finding.* The Check 'n' Crash and DSD-Crasher approach is explicitly dissimilar to a common class of static analysis tools that have received significant attention in the recent research literature. We call these tools collectively "bug pattern matchers". They are tools that statically analyze programs to detect specific bugs by pattern matching the program structure to well-known error patterns (e.g., Hallem et al. [2002], Hovemeyer and Pugh [2004], and Xie and Engler [2003]). Such tools can be quite effective in uncovering a large number of suspicious code patterns and actual bugs in important domains. But the approach requires domain-specific knowledge of what constitutes a bug. In addition, bug pattern matchers often use a lightweight static analysis, which makes it harder to integrate with automatic test case generators. For example, FindBugs does not produce rich constraint systems (like ESC/Java does) that encode the exact cause of a potential bug.

Model-checking techniques offer an alternative approach to overapproximating program exploration and therefore bug searching. Recent model-checkers directly analyze Java bytecode, which makes them comparable to our overapproximating bug search component ESC/Java. Well-known examples are Bogor/Kiasan by Deng et al. [2006] and Java PathFinder with symbolic extensions by Khurshid et al. [2003]. Building on model-checking techniques is an interesting direction for bug-finding and is being explored in the context of JML-like specification languages by Deng et al. [2007].

Verification tools such as those by Beyer et al. [2004] or Kroening et al. [2004] are powerful ways to discover deep program errors. Nevertheless, such tools are often limited in usability or the language features that they support. Jackson and Vaziri [2000] and Vaziri and Jackson [2003] enable automatic checking of complex user-defined specifications. Counterexamples are presented to the user in the formal specification language. Their method addresses bug finding for linked data structures, as opposed to numeric properties, object casting, array indexing, etc., as in our approach.

8.4.3 *Finding Feasible Executions.* AutoTest by Meyer et al. [2007] is a closely related automatic bug finding tool. It targets the Eiffel programming language, which supports invariants at the language level in the form of contracts [Meyer 1997]. AutoTest generates random test cases, like JCrasher, but uses more sophisticated test selection heuristics and makes sure that generated test cases satisfy given testee invariants. It can also use the given invariants as its test oracle. Our tools do not assume existing invariants since, unlike Eiffel programmers, Java programmers usually do not annotate their code with formal specifications.

Korat by Boyapati et al. [2002] generates all (up to a small bound) non-isomorphic method parameter values that satisfy a method's explicit precondition. Korat executes a candidate and monitors which part of the testee state it accesses to decide whether it satisfies the precondition and to guide the generation of the next candidate. The primary domain of application for Korat is that of complex linked data structures. Given explicit preconditions, Korat will generate deep random tests very efficiently. Thus, Korat will be better than DSD-Crasher for the cases when our constraint solving does not manage to produce values for the abstract constraints output by ESC/Java and we resort to random testing. In fact, the Korat approach is orthogonal to DSD-Crasher and could be used as our random test generator for reference constraints that we cannot solve. Nevertheless, when DSD-Crasher produces actual solutions to constraints, these are much more directed than Korat. ESC/Java analyzes the method to determine which path we want to execute in order to throw a runtime exception. Then we infer the appropriate constraints in order to force execution along this specific path (taking into account the meaning of standard Java language constructs) instead of just trying to cover all paths.

9. CONCLUSIONS AND FUTURE WORK

We have presented DSD-Crasher: a tool based on a hybrid analysis approach to program analysis, particularly for automatic bug finding. The approach combines three steps: dynamic inference, static analysis, and dynamic verification. The dynamic inference step uses Daikon [Ernst et al. 2001] to characterize a program's intended input domains in the form of preconditions, the static analysis step uses ESC/Java [Flanagan et al. 2002] to explore many paths within the intended input domain, and the dynamic verification step uses JCrasher [Csallner and Smaragdakis

2004] to automatically generate tests to verify the results of the static analysis. The three-step approach provides several benefits over existing approaches. The preconditions derived in the dynamic inference step reduce the false bug warnings produced by the static analysis and dynamic verification steps alone. The derived preconditions can also help the static analysis to reach a problematic statement in a method by bypassing unintended input domains of the method's callees. In addition, the static analysis step provides more systematic exploration of input domains than the dynamic inference and dynamic verification alone.

The current DSD-Crasher implementation focuses on finding crash-inducing bugs, which are exposed by inputs falling into intended input domains. As we discussed in Section 4, intended input domains inferred by Daikon could be narrower than the real ones; therefore, a crash-inducing bug could be exposed by an input falling outside inferred input domains but inside the (real) intended input domain. In the future, we plan to develop heuristics to relax inferred input domains to possibly detect more bugs. In addition, some bugs do not cause the program to crash but violate real postconditions. The current DSD-Crasher implementation does not consider inputs that satisfy inferred preconditions but violate inferred postconditions, because this may lead to additional bug warnings, requiring much inspection effort. We plan to develop heuristics (based on constraints generated by ESC/Java for violating a certain postcondition) to select for inspection a small number of inferred-postcondition-violating test inputs.

Our DSD-Crasher implementation and testees are available at: `http://www.cc.gatech.edu/cnc/`

Acknowledgments

We thank Koushik Sen who offered help in using jCUTE and Willem Visser and Saswat Anand who offered help in using Java Pathfinder's symbolic execution. We are especially grateful to the anonymous referees whose detailed responses greatly improved this paper. We gratefully acknowledge support by the NSF under Grants CCR-0735267 and CCR-0238289.

REFERENCES

Anand, S., Godefroid, P., and Tillmann, N. 2008. Demand-driven compositional symbolic execution. In *Proc. 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. Springer, To appear.

Anand, S., Pasareanu, C., and Visser, W. 2007. JPF-SE: A symbolic execution extension to Java Pathfinder. In *Proc. 13th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. Springer, 134–138.

Apache Software Foundation. 2003. Bytecode engineering library (BCEL). http://jakarta.apache.org/bcel/. Accessed Dec. 2007.

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

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

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

Beyer, D., Chlipala, A. J., Henzinger, T. A., Jhala, R., and Majumdar, R. 2004. Generating tests from counterexamples. In *Proc. 26th International Conference on Software Engineering (ICSE)*. IEEE, 326–335.

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

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

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

Centonze, P., Flynn, R. J., and Pistoia, M. 2007. Combining static and dynamic analysis for automatic identification of precise access-control policies. In *Proc. 23rd Annual Computer Security Applications Conference (ACSAC)*. IEEE, 292–303.

Clarke, L. A. 1976. A system to generate test data and symbolically execute programs. *IEEE Transactions on Software Engineering 2,* 3, 215–222.

Cok, D. R. and Kiniry, J. R. 2004. 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.

Coverity Inc. 2003. Coverity Prevent. http://www.coverity.com/. Accessed Dec. 2007.

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

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

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

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

Csallner, C., Tillmann, N., and Smaragdakis, Y. 2008. DySy: Dynamic symbolic execution for invariant inference. In *Proc. 30th ACM/IEEE International Conference on Software Engineering (ICSE)*. ACM, To appear.

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

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

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

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

Do, H., Elbaum, S. G., and Rothermel, G. 2005. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *Empirical Software Engineering 10,* 4 (Oct.), 405–435.

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

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

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

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

Godefroid, P. 2007. Compositional dynamic test generation. In *Proc. 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. ACM, 47–54.

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

Gupta, N., Mathur, A. P., and Soffa, M. L. 1998. Automated test data generation using an iterative relaxation method. In *Proc. 6th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*. ACM, 231–244.

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

Hangal, S. and Lam, M. S. 2002. Tracking down software bugs using automatic anomaly detection. In *Proc. 24th International Conference on Software Engineering (ICSE)*. ACM, 291–301.

Hapner, M., Burridge, R., Sharma, R., and Fialli, J. 2002. *Java message service: Version 1.1.* Sun Microsystems, Inc.

Henkel, J., Reichenbach, C., and Diwan, A. 2007. Discovering documentation for Java container classes. *IEEE Transactions on Software Engineering 33,* 8 (Aug.), 526–543.

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

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

Jackson, D. and Vaziri, M. 2000. Finding bugs with a constraint solver. In *Proc. ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*. ACM, 14–25.

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

King, J. C. 1976. Symbolic execution and program testing. *Communications of the ACM 19,* 7, 385–394.

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

Korel, B. 1990. Automated software test data generation. *IEEE Transactions on Software Engineering 16,* 8, 870–879.

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

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

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

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

McConnell, S. 2004. *Code Complete*, Second ed. Microsoft Press.

Meyer, B. 1997. *Object-Oriented Software Construction*, Second ed. Prentice Hall PTR.

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

Nimmer, J. W. and Ernst, M. D. 2002a. Automatic generation of program specifications. In *Proc. ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*. ACM, 229–239.

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

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

Parasoft Inc. 2002. Jtest. http://www.parasoft.com/. Accessed Dec. 2007.

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

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

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

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

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

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

Tomb, A., Brat, G. P., and Visser, W. 2007. Variably interprocedural program analysis for runtime error detection. In *Proc. ACM/SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*. ACM, 97–107.

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

Visser, W., Havelund, K., Brat, G., and Park, S. 2000. Model checking programs. In *Proc. 15th IEEE International Conference on Automated Software Engineering (ASE)*. IEEE, 3–12.

Visser, W., Pasareanu, C. S., and Khurshid, S. 2004. Test input generation with Java PathFinder. In *Proc. ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*. ACM, 97–107.

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

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

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

Xie, Y. and Engler, D. 2003. Using redundancies to find errors. *IEEE Transactions on Software Engineering 29,* 10 (Oct.), 915–928.

Young, M. 2003. Symbiosis of static analysis and program testing. In *Proc. 6th International Conference on Fundamental Approaches to Software Engineering (FASE)*. Springer, 1–5.

Zhu, H., Hall, P. A. V., and May, J. H. R. 1997. Software unit test coverage and adequacy. *ACM Computing Surveys 29,* 4, 366–427.

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