DSD-Crasher: A Hybrid Analysis Tool for Bug Finding

Christoph Csallner, Yannis Smaragdakis
College of Computing
Georgia Institute of Technology
Atlanta, GA 30332, USA
{csallner,yannis}@cc.gatech.edu

ABSTRACT

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 verifying the results of the static analysis. Thereby confirmed results are never false positives, as opposed to the high false positive rate inherent in conservative static analysis.

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

Keywords

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 [4, 9, 13, 33, 35]. In this paper, we present DSD-Crasher: a tool that uses dynamic analysis to infer likely program invariants, explores the space defined by these invariants exhaustively through static analysis, and finally produces and executes test cases to confirm that the behavior is observable under some real inputs and not just due to overgeneralization in the static analysis phase. Thus, our combination has three steps: dynamic inference, static analysis, and dynamic verification (*DSD*).

More specifically, we employ the Daikon tool [14] to infer likely program invariants from an existing test suite. The results of Daikon are exported as JML annotations [22] that are used to guide our CnC tool [9]. 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). CnC employs the ESC/Java static analysis tool [15], 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.

Several past research tools follow an approach similar to ours, but omit one of the three stages of our analysis. Clearly, CnC is a representative of a static-dynamic (SD) approach. There are several DD tools with the closest ones (because of the concrete techniques used) being the Eclat tool by Pacheco and Ernst [27]. 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 Nimmer and Ernst's [25] Daikon-ESC/Java (DS) combination. Nevertheless, Nimmer and Ernst's domain was significantly different from ours and we show that their metrics are not applicable to fully automatic bug finding (it is easy to optimize the metrics without improving the quality of reports). These metrics are appropriate when Daikon-inferred invariants are inspected by humans, as in Nimmer and Ernst's evaluation.

ISSTA'06, July 17–20, 2006, Portland, Maine, USA.

The benefit of DSD-Crasher over past approaches is either in enhancing the ability to detect errors, or in limiting false positives (i.e., false error reports). For instance, compared to CnC, DSD-Crasher produces more accurate error reports with significantly fewer false positives. CnC 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 the actual problem in the calling method. Without preconditions, CnC is more likely to cause a crash in a method that is called by the tested method before the actual problematic statement is reached. Compared to the Eclat tool, DSD-Crasher is more efficient in finding more bugs because of its deeper static analysis, relative to Eclat's mostly random testing.

We evaluated our approach on medium-sized third-party applications (the Groovy scripting language and the JMS module of the JBoss application server). We show that DSD-Crasher is helpful in removing false positives relative to just using the CnC tool. Overall, we found DSD-Crasher to be a strict improvement over CnC, provided that the application has a regression test suite that exercises well the functionality under test. At the same time, the approach is more powerful than Eclat. The static analysis allows more directed generation of test cases and, thus, uncovers more errors in the same amount of time.

2. BACKGROUND

Our three-step DSD-Crasher approach is based on two existing tools: Daikon (Section 2.1) and CnC (Section 2.3), which combines ESC/Java (Section 2.2) and the JCrasher test case generator [8]. This section presents background information on these tools.

2.1 Daikon: Guessing Invariants

Daikon [14] 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) [14, 26]. 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 [22].

2.2 ESC: Guessing Invariant Violations

The Extended Static Checker for Java (ESC/Java) [15] is a compile-time program checker that detects potential invariant violations. ESC/Java recognizes invariants stated in the Java Modeling Language (JML) [22]. (We use the ESC/Java2 system [7]—an evolved version of the original ESC/Java, which supports JML specifications and recent versions of the Java language.) 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 compiles the Java source code under test to a set of predicate logic formulae [15]. 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 [11] to derive error conditions for a method.

Like many other static analysis systems, ESC/Java is imprecise, also called *unsound*: it can produce spurious error reports because of inaccurate modeling of the Java semantics. ESC/Java is also *incomplete*: it may miss some errors—for example, because ESC/Java ignores all iterations of a loop beyond a fixed limit. [1]

2.3 CnC: Confirming Guessed Violations

CnC [9] is a tool for automatic bug finding. It combines ESC/Java and the JCrasher random testing tool [8]. CnC 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 an error truly exists. Compared to ESC/Java alone, CnC's combination of ESC/Java with JCrasher eliminates spurious warnings and improves the ease of comprehension of error reports through concrete Java counterexamples.

CnC takes as inputs the names of the Java files under test. It invokes ESC/Java, which derives error conditions. CnC takes each error condition as a constraint system over a method `m`'s parameters, the object state on which `m` is ex-

[1]The meaning of the terms "sound" and "complete" depends on the intended use of the system. In mathematical logic, a "sound" system is one that can only prove true statements, while a "complete" system can prove *all* true statements. Thus, if we view a static checker as a system for proving the existence of errors (as in our work) then it is "sound" iff reporting an error means it is a true error and "complete" iff all incorrect programs produce an error report (or, equivalently, no report is produced iff the program is correct). The terminology is exactly the inverse for tools that view the static checker as a system for proving programs correct [23, 15, 20].

ecuted, and other state of the environment. CnC extends ESC by parsing and solving this constraint system. A solution is a set of variable assignments that satisfy the constraint system. [9] 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, CnC uses JCrasher to compile some of these assignments to JUnit [3] test cases. The test cases are then executed under JUnit. If the execution does not cause an exception, then the variable assignment was a false positive: no error actually exists. Similarly, some runtime exceptions do not indicate errors and JCrasher filters them out. For instance, throwing an `IllegalArgumentException` exception is the recommended Java practice for reporting illegal inputs. If the execution does result in one of the tracked exceptions, an error report is generated by CnC.

2.4 CnC Example

To see the difference between an error condition generated by ESC/Java and the concrete test cases output by CnC, 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 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 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 emits some 100 constraints. The most relevant ones are `0 < fstArray.length` and `sndArray.length == 0` (formatted for readability).

CnC parses the error condition generated by ESC and feeds the constraints to its constraint solvers. In our example, CnC creates two integer variables, `fstArray.length` and `sndArray.length`, and passes their constraints to the POOC integer constraint solver [31]. Then CnC requests a few solutions for this constraint system from its constraint solvers and compiles each solution into a JUnit [3] test case. For this example, a test cases will generate an empty and a random non-empty array. The test case will cause an exception when executed and JCrasher will process the exception according to its heuristics and conclude it is a legitimate failure and not a false positive.

3. DSD-CRASHER: INTEGRATING DAIKON AND CNC

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 CnC, 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 CnC back-end runs the generated test cases, observes their execution, and reports violations.

We next describe the motivation, design and implementation of DSD-Crasher.

3.1 Motivation and Benefits

There are good reasons why DSD-Crasher yields benefits compared to just using CnC for bug detection. CnC, when used without program annotations, lacks interprocedural knowledge. This causes the following problems:

1. CnC may produce spurious error reports that do not correspond to actual program usage. For instance, a method `forPositiveInt` under test may be throwing an exception if passed a negative number as an argument: the automatic testing part of CnC will ensure that the exception is indeed possible and the ESC warning is not just a result of the inaccuracies of ESC 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 CnC test case invalid. Precondition annotations help CnC eliminate such spurious warnings.

2. CnC 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 CnC 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 CnC target its test cases better to reach the location of interest. This increases the chance of confirming ESC 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 CnC and issue many fewer false warnings (false positives) because of the Daikon-inferred preconditions. Reducing the number of false positives is a very valuable goal for bug finding tools, as various researchers have repeatedly emphasized when evaluating the practicality of their tools. In their assessment of the applicability of ESC/Java, Flanagan et al. write [15]:

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

The same conclusion is supported by the findings of other researchers, as we discuss in Section 6. Notably, Rutar et al. [29] 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 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." Daikon-inferred annotations can help ESC/Java and, by extension, CnC focus on warnings that are more likely to represent actual bugs.

3.2 Design and Implementation of DSD-Crasher

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 CnC 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. Nevertheless, we do not expect this to be a major issue in practice. When Daikon outputs an invariant, it is typically much more general than the specific values it observed during test suite execution. Thus, a wealth of behaviors is available for ESC/Java to explore exhaustively in trying to derive fault-causing conditions. In our later evaluation, we discuss how this observation has not affected DSD-Crasher's bug finding ability (relative to CnC) 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 positives in case the invariant `i != 0` is not accurate. The main goal of DSD-Crasher is to reduce false positives though. 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. Some of these violations might be of interest to the user. But we suspect that many of them are false positives and very confusing ones since the invariants are inferred without user interaction. Xie and Notkin [33] partially follow a similar approach with Daikon-inferred invariants that are used to produce test cases.

DSD-Crasher integrates Daikon and CnC through the JML language. Daikon can output JML conditions, which CnC 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 [33]. The Eclat tool by Pacheco and Ernst [27] excludes a similar list of invariants.

To make the Daikon output suitable for use in ESC/Java, we also had to provide JML annotations for Daikon's `Quant` class. Daikon expresses many invariants relative to methods of this class (e.g., `requires length == daikon.Quant.size(utf8String)`). ESC/Java needs the specifications of these methods to understand the meaning of such invariants. We thus provided JML specifications for some frequently used methods (i.e., `daikon.Quant.size`).

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* [22]. Behavioral subtyping is a standard object-oriented concept that should hold in well-designed programs (e.g., see "subcontracting" in Design by Contract [24]). 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 a superset of the preconditions of the method 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 at all 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 CnC, 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 [10] 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 CnC 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 ex-

ercised enough by the regression test suite) the standard CnC heuristics apply, whereas annotated methods are handled more strictly. Most notably, a `NullPointerException` is not considered a bug if thrown by an un-annotated method, unlike by an annotated method. This is standard CnC behavior [9] and doing otherwise would result in many false positive reports: as mentioned earlier, ESC/Java produces an enormous number of warnings for potential `NullPointerException`s when used without annotations [29]. Nevertheless, for a Daikon-annotated method, we have more information on its legal preconditions. Thus, it makes sense to report even "common" exceptions, such as `NullPointerException`, if these occur within the valid precondition space. Therefore, the CnC runtime needs to know whether a method was annotated with a Daikon-inferred precondition. To accomplish this we extended Daikon's Annotate feature to produce a list of such methods. When an exception occurs at runtime we check if the method on top of the call stack is in this list. One problem is that the call stack information at runtime omits the formal parameter types of the method that threw the exception. Thus, overloaded methods (methods with the same name but different argument types) can be a source for confusion. To disambiguate overloaded methods we use BCEL [1] to process the bytecode of classes under test. Using BCEL we retrieve the start and end line number of each method and use the line number at which the exception occurred at runtime to determine the exact method that threw it.

4. 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 [25].

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 CnC), 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 CnC) 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 two examples of metrics that we have found to be inappropriate for evaluating DSD-Crasher.

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 [4] and SLAM [2], 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 CnC 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 CnC-generated one. Nevertheless, this is a misleading metric. If CnC did not generate a test case that JCrasher would have, it is 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. Their FSE'02 paper [26] evaluates how well Daikon (and Houdini) can automatically infer program invariants to annotate a testee before checking it with ESC/Java. Their ISSTA'02 paper [25] 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 appropriate for human-controlled environments, 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.

5. EVALUATION

We evaluated DSD-Crasher and two of its closest relatives on the code of JBoss JMS and Groovy. 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.

5.1 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 [18]. 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 CnC as it was a part of CnC's past evaluation [9]. Nevertheless, the existing test suite supplied by the original authors was insufficient and we had to supplement it ourselves.

5.2 More Precise than Static-Dynamic CnC

The first benefit of DSD-Crasher is that it produces fewer false positives than the static-dynamic CnC tool.

5.2.1 JBoss JMS

CnC reported five cases, which include the errors reported earlier [9]. Two reports are false positives. We use one of them as an example on how DSD-Crasher suppresses false positives. Method `org.jboss.jms.util.JMSMap.setBytes` uses the potentially negative parameter `length` as the length in creating a new array. Calling `setBytes` with a negative `length` parameter causes a `NegativeArraySizeException`.

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

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

Table 1: Groovy results: DSD-Crasher vs. the static-dynamic CnC (SD).

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

5.2.2 Groovy

As discussed and motivated earlier, CnC by default suppresses most `NullPointerException`s because of the high number of false positives in 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 positives.

Table 1 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 CnC implementation, which in addition to the published description [9] only reports exceptions thrown by a method directly called by the generated test case. This restricts CnC's reports to the cases investigated by ESC/Java and removes accidental crashes inside other methods called before reaching the location of the ESC warning. CnC-classic is the current CnC 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. CnC-relaxed is CnC-classic but uses the same exception reporting as DSD-Crasher.

CnC-relaxed reports the 11 DSD-Crasher exceptions plus 8 others. (These are 15 `NullPointerException`s plus the four other exceptions reported by CnC-classic.) In 7 of the 8 additional exceptions, DSD-Crasher's ESC step could statically rule out the warning with the help of the Daikon-derived invariants. In the remaining case, ESC 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 positive reports confirms the benefits of the

Table 2: JBoss JMS results: `ClassCastException` reports by DSD-Crasher and the dynamic-dynamic Eclat. This table omits all other exception reports as well as all of Eclat's non-exception reports.

Tool	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

Daikon integration. Without it, CnC has no choice but to either ignore potential `NullPointerException`-causing bugs or to report them with a high false positive rate.

5.3 More Efficient than Dynamic-Dynamic Eclat

5.3.1 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 (`ClassCastException`s). The bugs found in JBoss JMS in the past [9] were `ClassCastException`s. (Most of the other reports concern `NullPointerException`s. Eclat produces 47 of them, with the vast majority being false positives. DSD-Crasher produces 29 reports, largely overlapping the Eclat ones. We estimate that most false positives would have been eliminated if the test suite had been thorough enough to produce reasonable Daikon invariants, as we confirmed by manually supplying appropriate unit test cases for some of the methods.)

Table 2 compares the `ClassCastExceptions` 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 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 [9]. 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;
  } //..
}
```

Table 3: Groovy results: DSD-Crasher vs. the dynamic-dynamic Eclat. This table omits all of Eclat's non-exception reports.

Tool	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

5.3.2 Groovy

Table 3 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 method that threw the exception is outside the testees specified by the user to be tested, 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 derives the right precondition and CnC generates a satisfying test case, whereas Eclat misses it. The constraints are: `arrayLength(sources) == 1`, `sources:141.46[i] == null`, `i == 0`. CnC generates the input value `new CharStream[]{null}` that satisfies the conditions, while Eclat just performs random testing and tries the value `null`.

5.4 Summary

The main question of our evaluation is whether DSD-Crasher is an improvement over using CnC alone. The clear answer is positive, as long as there is a regression test suite sufficient for exercising big 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 positives enables DSD-Crasher (as opposed to CnC) to produce reasonable reports about `NullPointerException`s. Furthermore, we never observed cases in our experiments where false Daikon invariants over-constrained a method input domain. This would have caused DSD-Crasher to miss a bug found by CnC. 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 3.2. Even such simple invariants are sufficient for limiting the false positives that CnC produces without any other context information.

Finally, the experience with our two third-party, open source applications is instructive. The regression test suite of Groovy was sufficient for our purposes and improved the quality of DSD-Crasher error reports relative to CnC. Nevertheless, the tests supplied with JBoss JMS were not sufficient, concentrating more on end-to-end testing tasks. We believe that regression tests developed under a unit-testing philosophy should be thorough enough for our purposes. This is confirmed by our experience of adding simple and natural unit tests to several JBoss JMS methods and observing an improvement of the DSD-Crasher reports relative to CnC (similarly to the Groovy results).

6. RELATED WORK

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

There are important surveys that concur with our estimate that an important problem is not just reporting potential errors, but minimizing false positives so that inspection by humans is feasible. Rutar et al. [29] evaluate five tools for finding bugs in Java programs, including ESC/Java 2, FindBugs [19], and JLint. The number of reports differs widely between the tools. For example, ESC reported over 500 times more possible null dereferences than FindBugs, 20 times more than JLint, and six times more array bounds violations than JLint. Overall, Rutar et al. conclude: "The main difficulty in using the tools is simply the quantity of output."

The CnC and DSD-Crasher approach is explicitly dissimilar to a common class of 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 [16, 19, 34]. The approach requires domain-specific knowledge of what constitutes a bug. Bug pattern matchers do not generate concrete test cases and often result in spurious warnings, due to the unsoundness of the modeling of language semantics. Yet such tools can be quite effective in uncovering a large number of suspicious code patterns and actual bugs in important domains.

The commercial tool Jtest [28] 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.

Several dynamic tools [5, 33] generate candidate test cases and execute them to filter out false positives. Xie and Notkin [33] present an iterative process for augmenting an existing test suite with complementary test cases. They use Daikon to infer a specification of the testee when executed on a given test suite. Each iteration consists of a static and a dynamic analysis, using Jtest and Daikon. In the static phase, Jtest generates more test cases, based on the existing specification. In the dynamic phase, Daikon analyzes the execution of these additional test cases to select those which violate the existing specification—this represents previously uncovered behavior. For the following round the extended specification is used. Thus, the Xie and Notkin approach is also a DSD hybrid, but Jtest's static analysis is rather limited (and certainly provided as a black box, allowing no meaningful interaction with the rest of the tool). Therefore this approach is more useful for a less directed augmentation of an existing test suite aiming at high testee coverage—as opposed to our more directed search for fault-revealing test cases.

Korat [5] 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.

Daikon is not the only tool for invariant inference from test case execution, although it is arguably the best known. For instance, Hangal and Lam present the DIDUCE invariant inferrer [17], which is optimized for efficiency and can possibly allow bigger testees and longer-running test suites than Daikon.

Much research work on automated software testing has concentrated on checking the program against some formal specification [12, 30, 36]. Of course, the main caveat to this automation is that the formal specification needs to be available in the first place. Writing a formal specification that is sufficiently descriptive to capture interesting properties for a large piece of software is hard. Furthermore, if the specification is reasonably expressive (e.g., in full first-order logic where the quantified variables can range over dynamic values) then conformance to the specification is not automatically checkable. (That is, conformance can be checked for the values currently available but checking it for all dynamic values is equivalent to program verification.) Therefore, generation of test data is again necessary. A good example is

the recent JML+JUnit work of Cheon and Leavens on using the Java Modeling Language (JML) to construct JUnit test cases [6]. Skeletal JUnit test case code that calls the program's methods is generated. The test inputs are user-supplied, however. The integration of JML in this approach is as a rich assertion language—the JML assertion checker is used to determine whether some invariant was violated.

Verification tools [20, 32, 4, 21] are powerful ways to discover deep program errors. Nevertheless, such tools are often limited in usability or the language features they support. Jackson and Vaziri [20, 32] enable automatic checking of complex user-defined specifications. Counterexamples are presented to the user in the formal specification language, which is less intuitive than DSD-Crasher generating a concrete test case. Their method addresses bug finding for linked data structures, as opposed to numeric properties, object casting, array indexing, etc., as in our approach.

7. 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 [14] to characterize a program's intended input domains in the form of preconditions, the static analysis step uses ESC/Java [15] to explore many paths within the intended input domain, and the dynamic verification step uses JCrasher [8] 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 positives 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 3, 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 trade soundness for completeness. In addition, some bugs do not cause the program to crash but violate real post-conditions. The current DSD-Crasher implementation does not consider inputs that satisfy inferred preconditions but violate inferred post-conditions, because a nontrivial percentage of these inputs expose no bugs, 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, trading automation for completeness.

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

Acknowledgments

We thank Tao Xie who offered extensive comments on previous versions of this paper and contributed to early discussions about the generation of test cases from Daikon invariants. We gratefully acknowledge support by the NSF under Grants CCR-0220248 and CCR-0238289.

8. REFERENCES

[1] Apache Software Foundation. Bytecode engineering library (BCEL). http://jakarta.apache.org/bcel/, Apr. 2003. Accessed May 2006.

[2] T. Ball. Abstraction-guided test generation: A case study. Technical Report MSR-TR-2003-86, Microsoft Research, Nov. 2003.

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

[4] D. Beyer, A. J. Chlipala, T. A. Henzinger, R. Jhala, and R. Majumdar. Generating tests from counterexamples. In *Proc. 26th International Conference on Software Engineering*, pages 326–335, May 2004.

[5] C. Boyapati, S. Khurshid, and D. Marinov. Korat: Automated testing based on Java predicates. In *Proc. 2002 ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 123–133. ACM Press, July 2002.

[6] Y. Cheon and G. T. Leavens. A simple and practical approach to unit testing: The JML and JUnit way. In B. Magnusson, editor, *ECOOP 2002 - Object-Oriented Programming: 16th European Conference*, volume 2374, pages 231–255. Springer, June 2002.

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

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

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

[10] C. Csallner and Y. Smaragdakis. Dynamically discovering likely interface invariants. In *Proc. International Conference on Software Engineering, Emerging Results Track*, pages 861–864, May 2006.

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

[12] S. H. Edwards. A framework for practical, automated black-box testing of component-based software. *Software Testing, Verification & Reliability*, 11(2):97–111, June 2001.

[13] M. D. Ernst. Static and dynamic analysis: Synergy and duality. In *Proc. ICSE Workshop on Dynamic Analysis*, pages 24–27, May 2003.

[14] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically discovering likely program

invariants to support program evolution. *IEEE Transactions on Software Engineering*, 27(2):99–123, Feb. 2001.

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

[16] S. Hallem, B. Chelf, Y. Xie, and D. Engler. A system and language for building system-specific, static analyses. In *Proc. ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, pages 69–82. ACM Press, June 2002.

[17] S. Hangal and M. S. Lam. Tracking down software bugs using automatic anomaly detection. In *Proc. 24th International Conference on Software Engineering*, pages 291–301, May 2002.

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

[19] D. Hovemeyer and W. Pugh. Finding bugs is easy. In *Companion to the 19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 132–136. ACM Press, Oct. 2004.

[20] D. Jackson and M. Vaziri. Finding bugs with a constraint solver. In M. J. Harrold, editor, *Proc. 2000 ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 14–25. ACM Press, Aug. 2000.

[21] D. Kroening, A. Groce, and E. M. Clarke. Counterexample guided abstraction refinement via program execution. In J. Davies, W. Schulte, and M. Barnett, editors, *Formal Methods and Software Engineering: 6th International Conference on Formal Engineering Methods*, pages 224–238. Springer, Nov. 2004.

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

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

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

[25] J. W. Nimmer and M. D. Ernst. Automatic generation of program specifications. In *Proc. 2002 ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 229–239, July 2002.

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

[27] C. Pacheco and M. D. Ernst. Eclat: Automatic generation and classification of test inputs. In *Proc. 19th European Conference on Object-Oriented Programming*, pages 504–527, July 2005.

[28] Parasoft Inc. Jtest. http://www.parasoft.com/, Oct. 2002. Accessed May 2006.

[29] N. Rutar, C. B. Almazan, and J. S. Foster. A comparison of bug finding tools for Java. In *Proc. 15th International Symposium on Software Reliability Engineering (ISSRE'04)*, pages 245–256. IEEE Computer Society Press, Nov. 2004.

[30] S. Sankar and R. Hayes. ADL—an interface definition language for specifying and testing software. In J. M. Wing and R. L. Wexelblat, editors, *Proc. workshop on Interface definition languages*, volume 29, pages 13–21. ACM Press, Aug. 1994.

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

[32] M. Vaziri and D. Jackson. Checking properties of heap-manipulating procedures with a constraint solver. In H. Garavel and J. Hatcliff, editors, *Tools and Algorithms for the Construction and Analysis of Systems: 9th International Conference*, pages 505–520. Springer, Apr. 2003.

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

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

[35] M. Young. Symbiosis of static analysis and program testing. In *Proc. 6th International Conference on Fundamental Approaches to Software Engineering*, pages 1–5, Apr. 2003.

[36] S. H. Zweben, W. D. Heym, and J. Kimmich. Systematic testing of data abstractions based on software specifications. *Software Testing, Verification & Reliability*, 1(4):39–55, Jan. 1992.