Check ’n’ Crash: Combining Static Checking and Testing

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

ABSTRACT

We present an automatic error-detection approach that combines static checking and concrete test-case generation. Our approach consists of taking the abstract error conditions inferred using theorem proving techniques by a static checker (ESC/Java), deriving specific error conditions using a constraint solver, and producing concrete test cases (with the JCrasher tool) that are executed to determine whether an error truly exists. The combined technique has advantages over both static checking and automatic testing individually. Compared to ESC/Java, we eliminate spurious warnings and improve the ease-of-comprehension of error reports through the production of Java counterexamples. Compared to JCrasher, we eliminate the blind search of the input space, thus reducing the testing time and increasing the test quality.

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*; H.5.2 [**Information Interfaces and Presentation**]: User Interfaces—*ergonomics, user-centered design*; I.2.2 [**Artificial Intelligence**]: Automatic Programming—*program verification*

General Terms

Human Factors, Reliability, Verification

Keywords

Automatic testing, dynamic analysis, extended static checking, static analysis, test case generation, usability

1. INTRODUCTION

The need to combine exhaustive and precise error checking approaches—i.e., static analysis and testing—has often been stated in the software engineering community. Testing remains the predominant way of discovering errors in real software. Nevertheless, approaches that utilize program analysis and formal reasoning [14, 19, 9, 10, 25, 27, 2, 4, 13] are gaining ground and may soon see widespread adoption.

In this paper, we present the combination of a semantic checking approach and testing. Specifically, we employ the ESC/Java [9] static checking system that analyzes a program, reasons abstractly about unknown variables, and tries to detect erroneous program operations. By default, ESC/Java checks for violations of implicit preconditions of primitive Java operations, such as accessing an array out-of-bounds, dereferencing null pointers, dividing by zero, etc. ESC/Java produces error reports that detail the abstract conditions necessary for the error to take place. Such conditions are constraints on program values—e.g., integer constraints, such as `x > 2 && y > 3*x` or object reference constraints, such as `p.next==q && q.prev==null`. We use these abstract conditions to derive concrete input values satisfying them (e.g., `x==3 && y==10 && p.next.prev == null`) through a combination of constraint solving tools. Finally, we employ our JCrasher tool [5] to produce actual Java test cases to exhibit the error. We call the combined tool CnC (Check ’n’ Crash).

The CnC approach combines static checking and automatic test generation to get the best of both. Relative to ESC/Java, the value of CnC is dual. First, the test cases are more accessible to human programmers than abstract error conditions: even in small examples ESC/Java can produce conditions many pages in length, which CnC reduces to a 10-line concise test case. Second, by only reporting failed tests, we ensure that the error is indeed a real one (with respect to the Java language semantics). ESC/Java is unsound: [1] it can produce spurious error reports due to

ICSE’05, May 15–21, 2005, St. Louis, Missouri, USA.

[1] Semantic note: 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 programs correct, then it is “sound” iff reporting no potential errors means no errors exist, and “complete” iff all correct programs result in no error. In contrast, however, if we see the static checker as a system to prove the existence of errors, then it is “sound” iff reporting an error means it is a true error (i.e., all correct programs result in no error, or what we called before “complete”) and “complete” iff all incorrect programs produce an error report (i.e., reporting no errors means no errors exist, or what we called before “sound”). In this paper, we treat static checkers as systems for finding errors. Thus, for instance,

inaccurate modeling of the Java semantics. Although there can be other ways to turn ESC/Java into a sound tool, our approach makes engineering sense: once an error condition is suspected, it is much easier to make informed guesses on input values and test them than to prove the existence of such values through abstract reasoning.

Relative to JCrasher, the value of CnC is in using the reasoning power of ESC/Java to decide which portions of the input space should be tested because they are likely to produce errors. As a result, with only a small number of test cases (e.g., a handful instead of many thousands) we find all of the bugs that JCrasher would find and occasionally bugs that JCrasher cannot find within realistic time and space bounds.

2. CNC: COMBINING STATIC CHECKING AND AUTOMATIC TESTING

2.1 Background: ESC and JCrasher

The Extended Static Checker for Java (ESC/Java) [9] is a compile-time program checker that detects precondition violations. ESC/Java recognizes preconditions stated in the Java Modeling Language (JML) [17]. (We use the ESC/Java2 system [4]—an evolved version of the original ESC/Java, which supports JML specifications and recent versions of the Java language.) Typically, JML specifications will either not exist at all, or only exist for a small fraction of the methods in an application and its libraries. Thus, ESC/Java tries to be useful even with partial information. In the absence of JML specifications, ESC/Java checks for violations of implicit preconditions of primitive Java operations. Such violations include accessing an array out-of-bounds, dereferencing null pointers, mis-casting an object, dividing by zero, etc. These conditions typically result in a program crash—the corresponding exceptions are almost never caught as part of normal operation in Java programs.

JCrasher [5] is an automatic testing tool for Java code. JCrasher examines the type information of a set of Java classes and constructs code fragments that will create instances of different types to test the behavior of public methods under random data. JCrasher attempts to detect bugs by causing the program under test to "crash", i.e., to throw an undeclared runtime exception. The output of JCrasher is a set of test cases for JUnit (a popular Java unit testing tool) [1]. JCrasher defines heuristics for determining whether a Java exception should be considered a program bug or the JCrasher supplied inputs have violated the code's preconditions.

In practice, the two tools have several similarities in their usage mode. Both are used on a method-by-method basis. When an error is reported by either tool, it is not certain to be a true bug in the program, even notwithstanding the unsoundness of the analysis of ESC/Java (i.e., even if there are real inputs for which the error occurs). In the case of ESC/Java an error can be an indication of a true bug but it could also be a result of a too permissive precondition. That is, the inputs causing the error may never occur during normal application execution due to conditions established by the rest of the code and not reflected in the method's preconditions. For realistic applications, JML preconditions exist very rarely. JCrasher does not even consider JML preconditions. The tool's ideal area of application is in code currently being developed, which is highly unlikely to have preconditions specified.

2.2 CnC Structure

Our CnC tool combines ESC/Java and JCrasher. CnC takes as input the names of the Java files under test. It invokes ESC/Java 2.07a, which compiles the Java source code under test to a set of predicate logic formulae [9, 18]. ESC compiles each method `m` under test to its weakest precondition $wp(\mathtt{m}, \mathtt{true})$. This formula specifies the states from which the execution of `m` terminates normally. We use `true` as the postcondition, as we are not interested in which state the execution terminates as long as it terminates normally. The states that do not satisfy the precondition are those from which the execution "goes wrong".

We are interested in a few specific cases of the execution going wrong [19, chapter 4].

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

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

A state from which the execution of `m` goes wrong is also called a "counterexample". ESC uses the Simplify theorem prover [6] to derive counterexamples from the conjunction of $wp(\mathtt{m}, \mathtt{true})$ and additional formulae that encode the class to which `m` belongs as well as Java semantics.

We view such a counterexample as a constraint system over `m`'s parameters, the object state on which `m` is executed, and other state of the environment. CnC extends ESC by parsing and solving this constraint system. A solution is a set of variable assignments that satisfy the constraint system. Section 2.3 shows examples of constraints and discusses in detail how we process constraints over integers, arrays, and reference types in general.

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

we call a system "unsound" if it produces spurious error reports, unlike References [19, 9, 14] that call such a system "incomplete". Similarly we use the term "false positive" for a false error report, not for a lack of report for a true error. We find our convention more intuitive for the intended use of the tool.

2.3 Constraint Solving

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

2.3.1 Primitive types: integers

We first discuss our approach for integer numbers.[2] CnC uses the integer constraint solver included in the POOC platform for object-oriented constraint programming [22]. As an example, consider the following method under test.

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

ESC will return the following constraint system, which we have pruned and simplified for readability.

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

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

2.3.2 Complex types: objects

Constraints in abstract counterexamples may involve aliasing relations among object references. We solve these with a simple equivalence-class-based aliasing algorithm, similar to the well-known alias analysis by Steensgard [23].

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

For an example, consider the following method.

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

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

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

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

We use reflection to set object fields in the generated test case. In our example:

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

2.3.3 Complex types: arrays

CnC uses a simple approach to deal with arrays. Array expressions with a constant index (such as `a[1]`) are treated as regular variables. Array expressions with a symbolic index (such as `a[b]` or `a[m()]`) are replaced with a fresh variable and the resulting constraint system (with regular references and primitives) is solved. The solutions returned are used one-by-one to turn array expressions with symbolic indices into array expressions with constant indices. If the resulting constraint system is solvable (i.e., has no contradictory constraints) then the solution is appropriate for the original constraint system.

An example illustrates the approach. Consider the method:

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

For the case of division by zero, CnC extracts the following constraint system from the ESC/Java counterexample report:

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

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

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

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

[2] Floating point constraints are currently not supported in CnC (random values are used, just as in JCrasher) but would be handled similarly.

3. BENEFITS RELATIVE TO ESC/JAVA

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

3.1 Improving soundness

Improving the soundness of a static checker is a highly desirable goal for software engineers. In their assessment of the applicability of ESC/Java, Flanagan et al. write [9]:

> "[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."

Our combination of ESC/Java with JCrasher aims to remedy exactly this weakness, by using testing techniques to complement the ESC/Java analysis.

Every reasoning system must make a trade-off between soundness and completeness. ESC/Java produces spurious error reports because it captures the Java semantics imprecisely. Of course, the loss of accuracy could exhibit itself as incompleteness (i.e., not catching errors that are there) rather than as unsoundness, but ensuring soundness at the abstract reasoning level will make a static checker miss too many potential errors, as the ESC/Java authors argue.

In contrast, with our approach, soundness is ensured by testing the program with actual data and seeing if the predicted problem can be actually observed. From an engineering point of view, this is an appropriate approach. To verify that some condition can actually occur, it is easier to make some informed guesses on input values and test them than to try to prove the existence of such values abstractly.

To see in which cases our approach would work better, consider some representative examples of the unsoundness of ESC/Java.

3.1.1 Intra-Procedurality.

In the absence of pre-conditions and post-conditions describing the assumptions and effects of called methods, ESC/Java analyzes each method in isolation without taking the semantics of other methods into account. For instance, consider the example:

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

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

ESC/Java will report potential errors for `get0() < 0` and `get0() > 0`, although neither of these conditions can be true.

Tools like Houdini [8] can possibly be used to extract ESC/Java failure predicates and employ their inverse as a pre-condition for the current method (essentially giving ESC/Java interprocedural capabilities). Preconditions can also possibly be inferred with a tool like Daikon [7], which observes invariant properties during program executions. Nevertheless, the inferred conditions by Houdini or Daikon are likely to be too low-level, conservative and of a local nature. Hence, they are unlikely to capture elements such as the state of variables.

In general, ESC/Java has poor handling of state: it does not recognize invariants of either the current class (e.g., invariant relations between fields) or the environment. For example, ESC/Java will report a potential error in a line of code calling the `System.out.println` method, complaining that the `System.out` object may be `null`. Note that we can fix this specific problem by giving ESC/Java access to the formal JML specification of `java.lang.System` (the ESC/Java distribution includes the JML specification for this specific class).

3.1.2 Exceptions.

ESC/Java treats every exception thrown by a Java system class as an error, even if the user code catches it. Many such handled exceptions should not be considered bugs, but rather non-procedural transfers of control flow. For instance, the program may catch an exception that is thrown to indicate some valid runtime condition (e.g., a file was not found because the user entered the wrong name).

3.1.3 Floating Point Arithmetic.

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

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

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

3.1.4 Big Integers.

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

3.1.5 Multiplication.

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

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

and also for:

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

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

3.1.6 Reflection.

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

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

3.1.7 Aliasing and Data Flow Incompleteness.

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

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

3.2 Improving the Clarity of Reports

CnC is more friendly to the user than ESC/Java. Not only are fewer spurious errors reported, but also the reports are much easier to inspect. Ease of inspection is paramount in verification. Musuvathi and Engler [20] summarize their experiences (in a slightly different domain) as:

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

When an error is reported by a program analysis system, it is generally desirable to see not just where the error occurred but also an example showing the error conditions. ESC/Java can optionally emit the counterexamples produced by the Simplify theorem prover. Yet these counterexamples contain just abstract constraints instead of specific values. Furthermore, there are typically hundreds of constraints even for a small method. For instance, for a 15-line method (from one of the programs examined later in Section 4) ESC/Java emits a report that begins:

```
P1s1.java:345: Warning: Array index
possibly too large (IndexTooBig)
 isTheSame(list[iList+1],pattern[iPatte ...
               ^

Execution trace information:
 Reached top of loop after 0 iterations in
  "P1s1.java", line 339, col 4.
 Executed then branch in "P1s1.java", line
  340, col 59.
 Reached top of loop after 0 iterations in
  "P1s1.java", line 341, col 6.

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

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

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

4. BENEFITS RELATIVE TO JCRASHER

CnC is a strict improvement over using JCrasher alone for automatic testing. By employing the power of ESC/Java, CnC guides the test case generation so that only inputs likely to produce errors are tested. Thus, a small number of test cases suffice to find all problems that JCrasher would likely find. To confirm this claim, we reproduced the experiments from the JCrasher paper [5] using CnC. The programs under test include the Raytracer application from the SPEC JVM 98 benchmark suite, homework submissions for an undergraduate programming class, and the `uniqueBoundedStack` class used previously in the testing literature [24, 26]. (Xie and Notkin refer to this class as UB-Stack, a name that we adopt for brevity.) These testees are mostly small and occasionally have informal specifications. For instance, in the case of homework assignments, we review the homework handout to determine what kinds of inputs should be handled by each method and how. Thus, we can talk with some amount of certainty of bugs instead of just error reports and potential bugs.

Table 1 summarizes the results of running JCrasher 0.2.7 and CnC 0.4.10 on the set of testees. The bugs are reported as a range containing some uncertainty, as occasionally it is clear that a program fragment represents a bad practice, yet there is a possibility that some implicit precondition makes the error scenario infeasible. CnC has more flexibility than JCrasher in its settings, therefore for these tests we chose the CnC settings so that they emulate the default JCrasher behavior. Specifically, CnC parameterizes ESC so that it searches for the potential problems listed in section 2.2, other than dereferencing null. ESC unrolls each loop 1.5 times. We use no JML specifications of the Java class libraries. Using these JML specifications included in the ESC distribution does not change the reports CnC produces or the bugs it finds for this experiment. We use the runtime heuristics reported in the JCrasher paper [5] for both JCrasher and CnC. These heuristics determine which exceptions should be ignored because they probably do not constitute program errors. For identical exceptions produced with the same method call stack, only one error report is output.

As can be seen in the table, CnC detects all the errors found by JCrasher with only a fraction of the test cases (except for UB-Stack, where JCrasher found few opportunities to create random data conforming to the class's interface) and slightly fewer reports. This confirms that the search of CnC is much more directed and deeper, yet does not miss any errors uncovered by random testing. (Note: The numbers reported for JCrasher are identical to those in reference [5] with two exceptions. First, the bug count for s1139 and s3426 is increased by one, since on further review two more reports were shown to be bugs. Second, for the Binary Search Tree homework submission, we run both programs on

Table 1: Results of using JCrasher and CnC on real programs (testees). *Public methods* gives the number of public methods and constructors declared by public classes. Constructors of abstract classes are excluded. *Test cases* gives the total number of test cases generated for a testee when JCrasher searches up to a method-chaining depth of three and CnC creates up to ten test cases per ESC generated counterexample. *Crashes* denotes the number of errors or exceptions of interest thrown when executing these test cases. *Reports* denotes the number of distinct groups of failures reported to the user—all crashes in a group have an identical call-stack trace. *Bugs* denotes the number of problem reports that reveal a violation of the testee's specification.

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

the BSTNode class, which contains the error, instead of on the BSTree front-end class.)

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

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

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

In addition to JCrasher's results, CnC found one more bug in all P1 testees except s8007's. When passing arrays of different lengths (e.g., `([1.0], [])`) to the `swapArrays` method, these testees crash by accessing an array out of bounds. The students' code typically contains the pattern:

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

We classify this as a bug, since the homework specification allows arrays of different lengths as input. JCrasher did not discover this error because its creation of random array values is limited.

Due to lack of space, we cannot discuss in detail the rest of the specific bugs discovered in these programs. The interested reader should consult reference [5] instead.

5. EXPERIENCE WITH CNC

We next describe our experience in using CnC on complete applications and our observations on the strengths and weaknesses of the tool.

5.1 JABA and JBoss JMS

To demonstrate the uses of CnC in practice, we applied it to two realistic applications: the JABA bytecode analysis framework[3] and the JMS module of the JBoss[4] open source J2EE application server. The latter is an implementation of Sun's Java Message Service API [11]. Specifically, we ran CnC on all the `jaba.*` packages of JABA, which consist of some 18 thousand non-comment source statements (NCSS), and on the JMS packages of JBoss 4.0 RC1, which consist of some five thousand non-comment source statements. We should note that there is no notion of testing the scalability of CnC since the time-consuming part of its analysis is the intra-procedural ESC/Java analysis. Hence, in practice, the CnC running time scales roughly linearly with the size of the input program.

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

Table 2 presents statistics of running CnC on JABA and JBoss JMS. The analysis and test creation time was measured on a 1.2 GHz Pentium III-M with 512 MB of RAM. We use the CnC configuration from the previous experiment (section 4), but also search for null dereferences.

Since we are not familiar with the internal structure of either of these programs, we are not typically able to tell whether an error report constitutes a real bug or some implicit precondition in the code precludes the combination of

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

Table 2: Experimental results for JABA and JBoss JMS.

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

inputs that exhibit a reported crash. Exhaustive inspection of all the reports by an expert is hard due to the size of the applications (especially JABA) and, consequently, the number of CnC reports. For instance, CnC (and ESC/Java) may report that a method can fail with a null pointer exception, yet it is not always clear whether the input can occur in the normal course of execution of the application. [5] For this reason, we selected a few promising error reports and inspected them more closely to determine whether they reveal bugs. In the case of JBoss JMS, it is clear on a couple of occasions (see below) that a report corresponds to a bug. Similarly, we discussed five potential errors with JABA developers and two of them entered their list of bugs to be fixed.

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

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

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

(Note that the type of `value` in the above is `Byte[]` and not `byte[]`.) CnC finds this error because ESC/Java reports a possible illegal cast exception in the above.

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

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

5.2 CnC Usage and Critique

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

A lot of attention in the error checking community has lately focused on tools that we descriptively call "bug pattern matchers" [10, 27, 13]. These are program analysis tools that use domain-specific knowledge about incorrect program patterns and statically analyze the code to detect possible occurrences of the patterns. Example error patterns include uses of objects after they are deallocated, mutex locks without matching unlocks along all control flow paths, etc. We should emphasize that CnC is not a bug pattern matcher: it has only a basic preconceived notion of what the program text of a bug would look like. Thus, the domain of application of CnC is different from bug pattern matchers, concentrating more on numeric properties, array indexing violations, errors in class casting, etc. Furthermore, CnC is likely to find new and unusual errors, often idiomatic of a programming or design style. Thus, it is interesting to use CnC to find a few potential errors in an application and then search for occurrences of these errors with a bug pattern matcher. A practical observation is that bug pattern matchers may not need to be very sophisticated in order to be useful: the greater value is often in identifying the general bug pattern, rather than in searching the program for the pattern.

We have already mentioned the strengths of CnC. It is a sound tool: any error reported can indeed happen for some combination of method inputs. It searches for possible

[5] It is worth emphasizing this point because it is a common source of confusion. There are two kinds of false positives in automatic error checking systems. The first kind is due to unsoundness: the system may report an error condition that cannot occur in the formal reasoning framework (e.g., the Java semantics). The second kind is that of false positives due to unknown implicit conditions about the inputs. *Any* error checking system (sound or not) is susceptible to this second kind of false positives.

error-causing inputs much more efficiently than JCrasher. It gives concrete, easy-to-inspect counterexamples. Nevertheless, CnC also has shortcomings. Although it is sound with respect to program execution semantics, it still suffers from false positives when the inputs are precluded by an unstated or informal precondition (e.g., JavaDoc comments). As mentioned earlier, every automatic error checking system has this weakness. Nevertheless, CnC possibly suffers more than bug pattern matching tools in this regard because it has no domain-specific or context knowledge. In contrast, a bug pattern matcher can often discover errors that are bugs with high probability: e.g., the use of an object after it has been freed. Nevertheless, due to the complexity of common bug patterns (e.g., needing to match data values and to recognize all control flow paths), bug pattern matchers typically suffer in terms of soundness. We speculate that users may be more willing to accept false positives due to unstated preconditions than due to unsoundness in the modeling of program execution. Another weakness of CnC is that it is less complete than ESC/Java because it cannot always derive concrete test cases from the Simplify counterexamples. We have found that in practice we still prefer the higher incompleteness of CnC to the many spurious warnings of ESC/Java.

6. RELATED WORK

The areas of automatic test case generation and program analysis are inexhaustible. Below, we selectively discuss a sampling of the recent related work.

6.1 Static techniques

We have already mentioned bug pattern matchers [10, 13, 27]: tools that statically analyze programs to detect specific bugs by pattern matching the program structure to well-known error patterns. Such tools do not generate concrete test cases and often result in spurious warnings, due to the unsoundness of the modeling of language semantics. Yet the tools encode interesting knowledge of common errors and can be quite effective in uncovering a large number of suspicious code patterns.

Xie and Engler [27] use their xgcc system [10] to search for redundancies in Linux, OpenBSD, and PostgreSQL. Their assumption is that redundancy indicates programmer confusion and therefore hard errors. Their experiments support this claim for intraprocedural redundancy. It remains unclear if this extends to interprocedural redundancy—ensuring that a parameter value or the result of a function call is within the expected range. The purpose of such redundant checks is robustness—protecting the code from future changes in other functions. Robustness could indicate an experienced programmer and therefore fewer hard errors. So a more robust testee might lead a redundancy-based approach to more false negatives. CnC, on the other hand, benefits from more interprocedural redundancy as passing unexpected parameters to a more robust testee leads to fewer spurious warnings. Kremenek and Engler [15] use statistical analysis to order the warnings produced by static analyses to mitigate their unsoundness.

Rutar et al. [21] evaluate five tools for finding bugs in Java programs, including ESC/Java 2, FindBugs [13], 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. The authors' experience is similar to ours in that ESC/Java used without annotating testee code produces too many spurious warnings to be useful alone.

6.2 Dynamic techniques

Dynamic tools [3, 26] generate candidate test cases and execute them to filter out false positives. Korat [3] 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 certainly generate deeper and more interesting tests than tools like JCrasher and CnC for constraints involving object references. An approach like that of Korat is orthogonal to CnC and could be integrated as part of the CnC constraint solving/test generation.

Xie and Notkin [26] address the problem of inspecting a large number of automatically generated tests. They add a candidate to their test suite only if it behaves differently—i.e., it violates the testee's invariants, which are inferred from the suite's execution traces using the Daikon tool [7]. These approaches lead to an efficient test suite with respect to covering the entire testee. CnC generates tests automatically and focuses on those parts of the testee that are likely to contain bugs. We address the problem of inspecting many automatically generated tests by grouping together similar test results. This is easier to do in our domain, since in CnC checking we have a witness of execution failure: the throwing of a runtime exception.

6.3 Verification techniques

Verification tools [14, 25, 2, 16] are sound but often limited in usability or the language features they support. Jackson and Vaziri [14, 25] enable automatic checking of complex user-defined specifications. Counterexamples are presented to the user in the formal specification language, which is less intuitive than CnC 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 the case of CnC.

Beyer et al. [2] leverage the higher efficiency of lazy abstraction based model checking for test case generation. If the search terminates, the approach is sound and complete: either a counterexample will be produced, or the absence of a counterexample is guaranteed. Like CnC, their tool converts a constraint system into a test case (i.e., concrete input values) using a constraint solver. Unlike CnC, however, the test case is not reified as a Java program.

The CRunner model checker [16] uses program execution to reduce the number of abstraction refinement iterations. The approach works best if a program execution happens to confirm early on an abstract counterexample produced from a coarse abstraction—this saves the subsequent refinement steps needed by traditional tools. CRunner is similar to CnC as it executes a testee to confirm its conservative static reasoning.

It is hard to compare software verification approaches based on symbolic reasoning (like the Simplify theorem prover used in CnC) to those based on model checking. The difficulty is greater with model checking approaches like that of Beyer et al., which employs symbolic reasoning (also using Simplify) for counterexample-guided model refinement. The rule of thumb in verification is that model checking approaches are better for finite state properties (e.g., reachability or deadlock), while theorem proving deals better with abstract properties that capture very large or infinite domains, such as integer arithmetic.

7. CONCLUSIONS

We presented an automatic error detection approach that combines abstract reasoning and automatic test case generation. Our CnC tool combines the advantages of both the ESC/Java static checker and the JCrasher automating testing tool. The result is a completely automatic error detection system, suitable for catching many low-level errors in arbitrary Java code. CnC emphasizes practicality by dealing with general-purpose Java programs through abstract reasoning and constraint solving, while producing concrete JUnit test cases as output. We have demonstrated the usefulness of CnC relative to ESC/Java and JCrasher, as well as by finding bugs in real world applications.

Acknowledgments

This research was supported by the NSF through grants CCR-0238289 and CCR-0220248, and by the Georgia Electronic Design Center.

We thank JABA developers Jim Jones and Alex Orso for the access to their code and expertise. We thank Tao Xie and the anonymous reviewers for their comments that helped improve the paper.

8. REFERENCES

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

[2] D. Beyer, A. J. Chlipala, T. A. Henzinger, R. Jhala, and R. Majumdar. Generating tests from counterexamples. In *Proceedings of the 26th International Conference on Software Engineering*, pages 326–335. IEEE Computer Society Press, May 2004.

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

[4] 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.

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

[6] 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.

[7] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. In *Proceedings of the 21st International Conference on Software Engineering*, pages 213–224. IEEE Computer Society Press, May 1999.

[8] C. Flanagan and K. R. M. Leino. Houdini, an annotation assistant for ESC/Java. In J. N. Oliveira and P. Zave, editors, *FME 2001: Formal Methods for Increasing Software Productivity: International Symposium of Formal Methods Europe*, pages 500–517. Springer, Mar. 2001.

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

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

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

[12] E. Hatcher and S. Loughran. *Java development with Ant.* Manning Publications, Aug. 2002.

[13] 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.

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

[15] T. Kremenek and D. Engler. Z-ranking: Using statistical analysis to counter the impact of static analysis approximations. In R. Cousot, editor, *Static Analysis: 10th Annual International Static Analysis Symposium*, pages 295–315. Springer, June 2003.

[16] 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.

[17] 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.

[18] K. R. M. Leino. Efficient weakest preconditions. Technical Report MSR-TR-2004-34, Microsoft Research, Apr. 2004.

[19] 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.

[20] M. Musuvathi and D. Engler. Some lessons from using static analysis and software model checking for bug

finding. In B. Cook, S. Stoller, and W. Visser, editors, *SoftMC 2003: Workshop on Software Model Checking.* Elsevier, July 2003.

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

[22] H. Schlenker and G. Ringwelski. POOC: A platform for object-oriented constraint programming. In B. O'Sullivan, editor, *Recent Advances in Constraints: Joint ERCIM/CologNet International Workshop on Constraint Solving and Constraint Logic Programming*, pages 159–170. Springer, June 2002.

[23] B. Steensgaard. Points-to analysis in almost linear time. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 32–41. ACM Press, Jan. 1996.

[24] D. Stotts, M. Lindsey, and A. Antley. An informal formal method for systematic JUnit test case generation. In D. Wells and L. A. Williams, editors, *Proceedings of the Second XP Universe and First Agile Universe Conference on Extreme Programming and Agile Methods - XP/Agile Universe 2002*, pages 131–143. Springer, Aug. 2002.

[25] 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.

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

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

APPENDIX

A. CNC USAGE

CnC is available at:
`http://www.cc.gatech.edu/cnc/`

A user project using CnC can integrate the following steps into its build process.

1. Execute CnC on selected testee sources and binaries.
2. Compile the test cases CnC generated.
3. Execute JUnit on the generated test cases.
4. Archive the test results.

We have tried to ease the integration of CnC into a user project's build process, yet separate the above general steps from the specifics of the project's build process.

Ant [12] (`http://ant.apache.org/`) is a popular open source tool to automate the build process of Java projects. Ant takes an XML formatted text file as input, which is also called the build file. An Ant build file is similar to a make file—it declares build targets and their dependencies. Ant interprets a build file and solves its dependencies to execute the build targets in a correct order.

The CnC distribution contains a generic Ant build file `use.xml` that invokes the CnC computation steps. The user project's build file `build.xml` then only needs to compile the testee and call CnC's `use.xml`, passing project specific parameters to `use.xml`. The following code fragments illustrate the idea.

```
<!-- build.xml -->
<project name="MyProject" [..]>
  <!-- project properties -->
  <!-- environment properties -->
  <!-- compile -->
  <!-- test, call use.xml -->
</project>
```

The CnC web site provides complete `build.xml` examples, which were used for the evaluation of CnC in this paper.

```
<!-- use.xml -->
<project name="CnC" [..]>
  <!-- test.generate -->
  <!-- test.compile -->
  <!-- test.run -->
  <!-- test.archive -->
</project>
```

Some user projects may need to adapt `use.xml` to integrate the execution of CnC generated test cases with other testing activities or to adapt the archival of test results to the user project requirements.