CnC: Check ’n’ Crash

Combining static checking and testing

Christoph Csallner and Yannis Smaragdakis

Georgia Institute of Technology
College of Computing
Atlanta, USA
Goal: Generate good test cases

Testing in software development

(+) Widespread, for good reasons
(–) Expensive

Automate as much as possible

Do not replace testing with full correctness proofs
But use proof techniques to automate testing
Generate tests that have a high probability of exposing errors
CnC: Check ’n’ Crash

Method under test

m(int a) { /*...*/ }

Abstract error condition

ESC  (-->) Semantics only approximated

p1, p2, p3, ...

Possible error

CnC

Find solutions

Test case

t() { m(1); }

(+) Execution semantics

Result of test case execution

Error observed

Not observed
Outline

1. **JCrasher**: Our completely automatic robustness tester
   - (-) Undirected: Black box
   - (+) Generates and dynamically filters JUnit test cases
   - (+) Precise: Execution semantics

   - (+) Directed: White box
   - (-) Abstract error conditions hard to understand
   - (-) Spurious warnings: Java semantics only approximated

3. **Check ’n’ Crash**: JCrasher testing directed by ESC/Java
   - (+) Directed by ESC/Java
   - (+) JUnit test cases for every abstract error condition
   - (+) No spurious warnings: Filtered via test execution
JCrasher: Generate random test cases

Completely automatic

Predefined values, e.g.:
int ← 1
int ← 0
A ← null

Infer method signatures from code under test, e.g.:
A ← (B, int) from method A B.m(int)

Generate random JUnit test cases

Method parameters by recursively chained constructors
JCrasher: Example

\( f(A, \text{int}) \) is the method under test

Inferred method signatures via Java reflection:

\( A \leftarrow \text{null} \)

\( A \leftarrow (B, \text{int}) \) from method \( A \ B.m(\text{int}) \)

\( B \leftarrow \text{null} \)

\( B \leftarrow () \) from constructor \( B() \)

\( \text{int} \leftarrow 1 \)

\( \text{int} \leftarrow 0 \)

JCrasher can generate the following test cases:

\( f(\text{null}, 0) \)

\( f((\text{new } B()).m(0), 1) \)

etc.
JCrasher: Monitor test execution

JUnit extension monitors execution of generated test cases
Filter runtime exceptions—crashes
Group similar reports together

Robustness heuristic

Some exceptions are mostly thrown by the Java runtime
Indicate abnormal termination of a Java language operation
Likely to be real bugs
E.g.: Public method under test attempts an illegal type cast
→ ClassCastException
Outline

1. **JCrasher**: Our completely automatic robustness tester
   
   (-) Undirected: Black box
   (+) Generates and dynamically filters JUnit test cases
   (+) Precise: Execution semantics

   
   (+) Directed: White box
   (-) Abstract error conditions hard to understand
   (-) Spurious warnings: Java semantics only approximated

3. **Check ’n’ Crash**: JCrasher testing directed by ESC/Java
   
   (+) Directed by ESC/Java
   (+) JUnit test cases for every abstract error condition
   (+) No spurious warnings: Filtered via test execution
ESC/Java by Flanagan, Leino, et al.

We use ESC/Java 2 (supports Java 1.4 and JML)
Checks for potential invariant violations
Invariant: Precondition on entry, postcondition on exit
ESC knows pre- and postconditions of language operations
(pointer dereference, class cast, array access, etc.)

ESC compiles each method in isolation:
Postcondition  →  assertion check appended to body
Call in body  →  check precondition, assume postcondition
Method body  →  its weakest precondition
ESC/Java

Weakest precondition $\text{wp(method body, true)} = \text{States from which execution terminates normally}$

Remaining states lead execution to an error: Violate some pre- or postcondition

ESC uses Simplify to derive abstract error conditions

We are interested in those that throw a runtime exception:

- Dereferencing null
- Illegal type cast
- Illegal array allocation or access
- Division by zero
- Violating explicit (JML) invariants
Drawbacks of ESC

Incomplete: Miss some errors (limited loop unrolling)

Imprecise = Unsound: Produces spurious error reports

Cases that cannot occur (Java semantics approximated)

Example: ESC/Java ignores implementation of called methods

public int get10() {return 10;}

public int meth(int p) {return p / get10();}

ESC warns of a division by zero in meth
Need to improve soundness

Flanagan et al. conclude in PLDI 2002:

“[ESC/Java] 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.”

“[Additional] research on reducing spurious warnings and lowering the perceived annotation burden [..] could add significant value to the process of engineering software.”

Our experience: Roughly half of ESC’s warnings were spurious (Spurious = cannot occur in any execution)
Outline

1. **JCrasher**: Our completely automatic robustness tester
   
   (−) Undirected: Black box
   (+) Generates and dynamically filters JUnit test cases
   (+) Precise: Execution semantics

   
   (+) Directed: White box
   (−) Abstract error conditions hard to understand
   (−) Spurious warnings: Java semantics only approximated

3. **Check ’n’ Crash**: JCrasher testing directed by ESC/Java
   
   (+) Directed by ESC/Java
   (+) JUnit test cases for every abstract error condition
   (+) No spurious warnings: Filtered via test execution
CnC: Check 'n' Crash

Method under test:

```
m(int a) { /*...*/ }
```

ESC (--) Semantics only approximated

Abstract error condition:

- $p_1, p_2, p_3, \ldots$

Possible error

CnC Find solutions

Test case:

```
t() { m(1); }
```

(+) Execution semantics

Result of test case execution:

- Error observed
- Not observed
Solving ESC’s abstract error condition

Integer: int, boolean, byte, short, and long

ESC uses Simplify [Detlefs, Nelson, and Saxe]

Does not know multiplication, except by constants
For absolute values $> 10^6$ only knows ordering
Linear reasoning precise for absolute values $\leq 10^6$

CnC uses POOC [Schlenker and Ringwelski]

Map to int, use third-party integer solver POOC
Solve constraints on integers $\leq 10^6$
Solving ESC’s abstract error condition

**float and double**
ESC very imprecise: Does not know that $1.0 \neq 2.0$
Currently ignored by CnC

**Object reference**
Own simple solver in CnC: Equivalence relation

**Array: Reference to a special object**
Integer constraints on array length and indices: POOC
Hack: Retrieve solutions until $\{a[b] \neq a[c]\} \Rightarrow \{b \neq c\}$

Better constraint solving would improve results

Any unconstrained variable: Use random values from JCrasher
Example: Testee

Consider the following student homework solution

```java
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.
Example: ESC’s warning

Array index possibly too large (IndexTooBig)

\[
fstArray[m] = sndArray[m];
\]

ESC’s abstract counterexample

Error condition as conjunction of constraints

5 instructions in `swapArrays` testee

100 constraints (at least) in ESC’s abstract counterexample

\[
0 < fstArray.length \\
sndArray.length == 0 \\
\ldots
\]
Example: ESC’s counterexample _______

(arrayLength(firstArray:294.43) <= intLast)
(firstArray:294.43 < longFirst)
(tmp0!new!double[]:297.27 < longFirst)
(secondArray:295.43 < longFirst)
(longFirst < intFirst)
(vAllocTime(tmp0!new!double[]:297.27) < alloc<1>)

(0 < arrayLength(firstArray:294.43))

(null <= max(LS))
(vAllocTime(firstArray:294.43) < alloc)
(alloc <= vAllocTime(tmp0!new!double[]:297.27))
(eClosedTime(elems@loopold) < alloc)
(vAllocTime(out:6..) < alloc)
(vAllocTime(secondArray:295.43) < alloc)
(intLast < longLast)
(1000001 <= intLast)
((intFirst + 1000001) <= 0)
(out:6.. < longFirst)
arrayLength(tmp0!new!double[]:297.27) ==
arrayLength(firstArray:294.43)
typeof(0) <: T_int
null.LS == @true
typeof(firstArray:294.43[0]) <: T_double
isNewArray(tmp0!new!double[]:297.27) == @true
typeof(arrayLength(firstArray:294.43)) <: T_int
T_double[] <: arrayType
typeof(firstArray:294.43) == T_double[
T_double[] <: T_double[]
elemtype(T_double[]) == T_double
T_double <: T_double
typeof(secondArray:295.43) <: T_double[
typeof(secondArray:295.43) == T_double[
arrayFresh(tmp0!new!double[]:297.27 alloc alloc<1>
   elems@loopold arrayShapeOne(arrayLength(firstArray:294.43))
   T_double[] F_0.0)
typeof(firstArray:294.43) <: T_double[
typeof(tmp0!new!double[]:297.27) == T_double[
(null <= max(LS))
tmp0!new!double[]:297.27[0] == firstArray:294.43[0]

arrayLength(secondArray:295.43) == 0

elems@loopold == elems
elems@pre == elems
state@pre == state
state@loopold == state
m@loopold:299.12 == 0
out@pre:6.. == out:6..
EC@loopold == EC
alloc@pre == alloc
!typeof(out:6..) <: T_float
!typeof(out:6..) <: T_byte
!typeof(secondArray:295.43) <: T_boolean
!typeof(firstArray:294.43) <: T_double
!typeof(secondArray:295.43) <: T_short
!typeof(tmp0!new!double[]):297.27) <: T_boolean
!typeof(tmp0!new!double[]):297.27) <: T_short
!typeof(firstArray:294.43) <: T_boolean
!typeof(out:6..) <: T_char
!typeof(secondArray:295.43) <: T_double
!typeof(firstArray:294.43) <: T_float
!typeof(out:6..) <: T_int
!isAllocated(tmp0!new!double[]):297.27 alloc)
!typeof(secondArray:295.43) <: T_long
!typeof(firstArray:294.43) <: T_char
!typeof(tmp0!new!double[]):297.27) <: T_double
!typeof(tmp0!new!double[]):297.27) <: T_long
T_double[] != T_boolean
T_double[] != T_char
T_double[] != T_byte
T_double[] != T_long
T_double[] != T_short
T_double[] != T_int
T_double[] != T_float
!typeof(firstArray:294.43) <: T_byte
!typeof(out:6..) <: T_boolean
!typeof(secondArray:295.43) <: T_float
!typeof(out:6..) <: T_short
!typeof(secondArray:295.43) <: T_byte
!typeof(tmp0!new!double[]):297.27) <: T_float
!typeof(firstArray:294.43) <: T_long
!typeof(tmp0!new!double[]):297.27) <: T_byte
!typeof(out:6..) <: T_double
!typeof(firstArray:294.43) <: T_short
!typeof(out:6..) <: T_long
typeof(out:6..) != T_boolean
typeof(out:6..) != T_char
typeof(out:6..) != T_byte
typeof(out:6..) != T_long
typeof(out:6..) != T_short
typeof(out:6..) != T_int
typeof(out:6..) != T_float
!typeof(secondArray:295.43) <: T_char
!typeof(secondArray:295.43) <: T_int
!typeof(tmp0!new!double[]):297.27) <: T_char
!typeof(firstArray:294.43) <: T_int
!typeof(tmp0!new!double[]):297.27) <: T_int
bool$false != @true
secondArray:295.43 != null
firstArray: 294.43 != null
T_double != typeof(out: 6..)
T_double != T_double[]
tmp0!new!double[]: 297.27 != null
Example: CnC processing

Parse ESC’s abstract counterexample

Feed relevant constraints to constraint solvers:
Above linear integer constraints to the POOC solver

Get some solutions and compile each into a JUnit test case:

```java
public void test0() throws Throwable {
    try {
        double[] d1 = new double[]{1.0};
        double[] d3 = new double[]{};
        P1s1.swapArrays(d1, d3);
    }
    catch (Exception e) {dispatchException(e);}
}
```
Example: Test execution

Test case passes any exception to the JCrasher filtering heuristic

Here, test0 causes:

1) test0(P1s1Test1)
java.lang.ArrayIndexOutOfBoundsException: 0
   at P1s1.swapArrays(P1s1.java:301)
   at P1s1Test1.test0(P1s1Test1.java:49)
   ...

The user classifies this as a bug
## Benefits relative to JCrasher

<table>
<thead>
<tr>
<th>Public method</th>
<th>Test Cases</th>
<th>Crashes</th>
<th>Reports</th>
<th>Bugs</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>JCrasher</td>
<td>CnC</td>
<td>JCr.</td>
<td>CnC</td>
</tr>
<tr>
<td>1</td>
<td>6</td>
<td>14382</td>
<td>12667</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>16</td>
<td>104</td>
<td>8</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>15</td>
<td>95</td>
<td>27</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>16</td>
<td>239</td>
<td>44</td>
<td>3</td>
</tr>
<tr>
<td>5</td>
<td>18</td>
<td>116</td>
<td>26</td>
<td>4</td>
</tr>
<tr>
<td>6</td>
<td>15</td>
<td>95</td>
<td>22</td>
<td>2</td>
</tr>
<tr>
<td>7</td>
<td>24</td>
<td>3872</td>
<td>1547</td>
<td>4</td>
</tr>
<tr>
<td>8</td>
<td>11</td>
<td>16</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
| 1..Canvas [SPEC JVM 98], 2–7..Student homework, 8..UB-stack [Stotts et al.]

2–8 available at CnC website

CnC found a superset of the bugs using fewer test cases
Experience with realistic applications

<table>
<thead>
<tr>
<th>Testee</th>
<th>Package Size [NCSS]</th>
<th>CnC Creation [min:s]</th>
<th>CnC Test Cases</th>
<th>CnC Crashes</th>
<th>CnC Reports</th>
</tr>
</thead>
<tbody>
<tr>
<td>jaba</td>
<td>17.9 k</td>
<td>25:58</td>
<td>18.3 k</td>
<td>4.4 k</td>
<td>56</td>
</tr>
<tr>
<td>jboss.jms</td>
<td>5.1 k</td>
<td>1:35</td>
<td>3.9 k</td>
<td>0.6 k</td>
<td>95</td>
</tr>
</tbody>
</table>

JBoss 4.0 RC1, JABA bytecode analysis framework [Harrold, Orso, et al.]

NCSS..Non-comment source statements

Found bugs in both applications

No formal specification: Completely automatic

Little informal specification: Some cases hard to decide
Conclusions

1. **JCrasher**: Our completely automatic robustness tester
   - (-) Undirected: Black box
   - (+) Generates and dynamically filters JUnit test cases
   - (+) Precise: Execution semantics

   - (+) Directed: White box
   - (-) Abstract error conditions hard to understand
   - (-) Spurious warnings: Java semantics only approximated

3. **Check ’n’ Crash**: JCrasher testing directed by ESC/Java
   - (+) Directed by ESC/Java
   - (+) JUnit test cases for every abstract error condition
   - (+) No spurious warnings: Filtered via test execution
Pointers

JCrasher: An automatic robustness tester for Java
Christoph Csallner and Yannis Smaragdakis.

Download JCrasher
http://www.cc.gatech.edu/jcrasher/

Download Check ’n’ Crash
http://www.cc.gatech.edu/cnc/
Need to improve the clarity of reports

Musuvathi and Engler write in SoftMC 2003 about a slightly different domain:

“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).”
ESC: Aliasing Example

public int alias(HeapNode n) {
    for (int i=0; i<1; i++) {n = n.next;}
    return 1/n.i;
}

Relevant constraints, reordered for comprehension:

\((n-165.2#0:166.4).i:7.15.12) == 0\)
\(n-165.2#0:166.4 == n:164.27<1>\)
\((n@loopold:164.27).next:7.14.17) == n:164.27<1>\)
\(n@loopold:164.27 == n:164.27\)

More constraints that look similar: Hard to filter
CnC: Constraints on complex types 

Maintain an equivalence relation

Constraint \( a = b \)

\( \rightarrow \) a and b point to the same object

\( \rightarrow \) Merge equivalence classes of a and b

Example:

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

Constraints \( a.x=0, \ a=b \)
CnC: Constraints on complex types

Constraints $a.x=0$, $a=b$

Set fields with reflection:

```java
Point p1 = new Point();  // JCrasher
Point.class.getDeclaredField("x"). // Reflection
  set(p1, new Integer(0));  // a.x=0
Point p2 = p1;  // a=b
Testee t = new Testee();  // JCrasher
  t.m2(p1, p2);
```
CnC: Arrays

Additional variables: length and indices

Example:

```java
public int m3(int[] a, int b) {
    if (a[a[b]] = 5) {return 1/0;}  // This line should be corrected to: if (a[a[b]] == 5) {
    else {return 0;}
}
```

Constraints for division by zero:

- $0 \leq b$
- $a[b] < \text{arrayLength}(a)$
- $0 \leq a[b]$
- $b < \text{arrayLength}(a)$
- $a[a[b]] = 5$
CnC: Array example

Solver cannot model that \( a[b] \neq a[c] \) implies \( b \neq c \)

\[
0 \leq b < \text{arrayLength}(a) \\
0 \leq a[b] < \text{arrayLength}(a) \\
a[a[b]] = 5
\]

First solution: \( b := 0, a[b] := 0, a[a[b]] := 5 \) illegal

Second solution: \( b := 0, a[b] := 1, a[a[b]] := 5 \)
CnC: Example bug found in JBoss

From `org.jboss.jms.BytesMessageImpl`

```java
public void writeObject(Object value)
    throws JMSException
{
    //..
    if (value instanceof Byte[])
        this.writeBytes((byte[]) value);
    //..
}
```
CnC: Example 2 found in JBoss

From `org.jboss.jms.container.Container`

Bad coding pattern or bug?

```java
public static Container getContainer
    (Object object) throws Throwable
{
    Proxy proxy = (Proxy) object;  //..
}
```
CnC: Automate with ANT (Java `make`)  

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

CnC distribution contains a generic Ant build file `use.xml`:

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