DSD-Crasher
Dynamic, Static, Dynamic

A hybrid analysis tool for bug finding

Christoph Csallner and Yannis Smaragdakis

Georgia Institute of Technology
College of Computing
Atlanta, USA
Find bugs in software

Testing widespread, for good reasons

- Requires understanding the (mostly informal) spec
- (-) Write test cases manually → Time-consuming

Generate test cases automatically (our Check ’n’ Crash)

- (-) Ignore informal spec → False positive warnings

Our goal: Generate better test cases (DSD-Crasher)

- (+) Infer informal spec → Fewer false positive warnings
False positives: Show-stopper

Flanagan et al. (ESC/Java people), 2002:

“[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.”

Rutar et al., 2004:

> 9k NPE warnings in 170k non commented source stmt

“[T]here are too many warnings to be easily useful by themselves.”
How we treat testee input

1. Dynamic
   Execute &
   Generalize

2. Static
   Exhaustive
   Search

3. Dynamic
   Confirm static results

Allowed by Java

Allowed by informal spec

Existing passing tests

Bug-finding
DSD-Crasher overview

Existing tests:
\[ m(1); \ m(2); \]

Dynamic
- Run test to infer spec:
  \[ i > 0 \]
  - Testee:
    \[ m(\text{int } i) \]
  - Testee Annotated:
    \[ \text{pre}: \ i > 0 \]

Static
- Search for bug:
  \[ i > 5 \]
  - New Test:
    \[ \{ m(6); \} \]
  - New result:
    \[ m \text{ crash} \]

Dynamic
- Run test to Confirm:
  \[ i = 6 \]
# DSD-Crasher overview

<table>
<thead>
<tr>
<th>Dynamic</th>
<th>Static</th>
<th>Dynamic</th>
</tr>
</thead>
<tbody>
<tr>
<td>Run test</td>
<td>Search for bug</td>
<td>Run test</td>
</tr>
<tr>
<td>to infer spec:</td>
<td>i &gt; 5</td>
<td>To Confirm:</td>
</tr>
<tr>
<td>i &gt; 0</td>
<td></td>
<td>i = 6</td>
</tr>
</tbody>
</table>

- **Daikon**
- **Annotate**
- **ESC**: Simplify prover
- **CnC**: POOC solver
- **JUnit**
- **JCrasher**: runtime
- **JCrasher**: test generator
ESC/Java: Extended Static Checking

★ Compile-time program checker
★ Recognizes invariants stated in Java Modeling Language
★ Knows pre- and postconditions of language operations
  Pointer dereference, class cast, array access, etc.
★ Detects potential invariant violations
  We concentrate on runtime exceptions
★ Analyzes each method in isolation
  (–) No inter-procedural analysis
ESC/Java: Analyzing a method

1. In the body, replace any call to any entity m with:
   
   check precondition(m);
   assume postcondition(m)

2. Compute weakest precondition:
   
   wp(method body, true)

   States from which execution terminates normally
   Remaining states lead execution to an error
   
   Violate some precondition (or postcondition)
   • Dereferencing null
   • Illegal type cast
   • Illegal array allocation or access
   • Division by zero
False positives (violate Java semantics)

Imprecise = Unsound: ESC produces spurious error reports

Cases that cannot occur (Java semantics approximated)

```java
public int get10() {return 10;}

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

ESC ignores implementation of `get10()`

ESC warns of a division by zero in `meth`

Our solution: Generate test, confirm behavior: DSD
DSD-Crasher overview

Existing tests:
\[ m(1); \ m(2); \]

Dynamic
- Run test to infer spec:
  - \( i > 0 \)
  - Testee: \( m(\text{int } i) \)

Static
- Search for bug
  - \( i > 5 \)
  - Testee Annotated:
    - \( \text{pre: } i > 0 \)
  - New Test:
    - \( \{m(6);\} \)

Dynamic
- Run test to Confirm:
  - \( i = 6 \)
  - New result:
    - \( m \) crash
False positives (violate user spec)  

ESC cannot access informal spec

```java
public int forPosInt(int i) {
    if (i<0) throw new MyRuntimeException();
    //..
}
```

1. User will consider report a **false positive**
   Need to infer implicit spec (first dynamic step)

2. May also terminate test execution path to real problem

```java
public int caller(int p) {
    int j = forPosInt(p);
    //bug

    (-) May suppress true positives
```
Idea

1. Dynamic
   Execute & Generalize

2. Static
   Exhaustive Search

3. Dynamic
   Confirm static results

Allowed by Java

Allowed by informal spec

Existing passing tests

Bug-finding
Daikon: Infer invariants from execution

1. Instrument testee
2. Execute testee
   Dump variable values at each method entry & exit
3. Analyze execution traces
   (a) Instantiate invariant templates with variable values
   (b) Invariant invalidated by sample → Drop invariant
4. Invariant held for some samples and never invalidated
   → Assume: true invariant
5. Annotate testee’s source code (with JML):
   Preconditions, postconditions, class invariants
Daikon configuration

- Concentrate on simple invariants, e.g.
  - intVariable \{ ==, >=, >, <=, < \} intConstant
  - intVariable \{ ==, >=, >, <=, < \} intVariable

- Complex invariants even harder to infer correctly

- Ignore most complex invariants that involve
  - variable is one of \{ const1, const2, .. \}
  - Elements of container structures
  - float, double, String

Daikon expresses invariants relative to methods

Wrote JML spec for frequently used methods, e.g.:
daikon.Quant.size
DSD-Crasher overview

Existing tests: m(1); m(2);

Dynamic
Run test to infer spec:

Static
Search for bug

Dynamic
Run test to Confirm:

Testee:
m(int i)

Testee Annotated:
pre: i>0

New Test:
{m(6);}

New result:
m crash
Inferred invariants as assumptions

//inferred pre-condition: (i != 0)
//inferred post-condition: (\result < 0)
public int foo(int i) { .. }

• Daikon invariants generalize observed behavior

• Body of foo: Assume input to be (i != 0)
  (+) Exclude probably unwanted input
  (−) May miss bugs caused by (i == 0)

• Calling foo: Assume (\result < 0)
  (+) Exclude probably impossible output
  (−) May miss bugs when foo (\returns >= 0)
Inferred invariants not requirements

// inferred pre-condition: (i != 0)
// inferred post-condition: (\result < 0)
public int foo(int i) {...}

- Very likely that Daikon has missed valid executions
  
  Calling foo with (i == 0) may be ok
  foo producing (\result >= 0) may be ok

- Do not confuse user with violations of guessed invariants
  Do not enforce inferred invariants as requirements
What is behavioral sub-typing?

- Pre-condition(\(C.m\)): \((P \text{ or } R)\)
- Post-condition(\(C.m\)): \((P \rightarrow Q)\) and \((R \rightarrow S)\)
Example of contradicting invariants

Daikon associates values & invariants with executed body

```java
class Super
  //@ ensures \result==1;
  int m() { .. }

class C
  //@ also
  //@ ensures \result==0;
  int m() { .. }
```

Derive postcondition for \texttt{C.m}:

\[(\text{result}==1) \land (\text{result}==0) \leftrightarrow \text{false}\]
Evaluation

Goal is to find bugs, not to cover code

(+ ) Static analysis already covers code

Nimmer & Ernst 2002: Precision and recall

(+ ) Great if you know perfect results (here: all bugs)

**JBoss JMS, part of JBoss 4.0 RC1**
5k non-comment source statements

**Groovy 1.0 beta 1 version, excluded low-level classes**
2k non-comment source statements
Used 603 of its unit test cases
Daikon produced 1.5 MB compressed invariants

http://groovy.codehaus.org/
Check ’n’ Crash (SD)

Dynamic
- Run test to infer spec:
  - i > 0
  - Daikon

Static
- Search for bug
  - i > 5
  - ESC: Simplify
  - POOC: solver

Dynamic
- Run test to Confirm:
  - i = 6
  - JUnit
  - JCrasher

Check ‘n’ Crash
More precise than Check ’n’ Crash (SD)  

JBoss JMS: Check ’n’ Crash reports false positive warning of NegativeArraySizeException

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

Used test case that calls `setBytes` three times
Daikon infers
requires `length == daikon.Quant.size(value)`
DSD-Crasher suppresses false positive
More precise than Check ’n’ Crash (SD)

Groovy experiments

<table>
<thead>
<tr>
<th>Tool</th>
<th>Runtime [min:s]</th>
<th>Exception reports</th>
</tr>
</thead>
<tbody>
<tr>
<td>CnC-relaxed</td>
<td>10:43</td>
<td>19</td>
</tr>
<tr>
<td>DSD-Crasher</td>
<td>30:32</td>
<td>11</td>
</tr>
</tbody>
</table>

Using Daikon-inferred invariants

12..18: ESC could statically rule out false positives

19: ESC produces more complicated error condition, threw off constraint solver (easy to fix)
Eclat (DD)

**Dynamic**
- Run test to infer spec
  - \( i > 0 \)
  - Daikon

**Static**
- Search for bug
  - \( i > 5 \)

**Dynamic**
- Run test to Confirm:
  - \( i = 6 \)
  - JUnit

Does more than we look for
Static analysis mostly random
Deeper than Eclat (DD)

**JBoss JMS:** `ClassCastException` reports

<table>
<thead>
<tr>
<th>Tool</th>
<th>Reports</th>
<th>Runtime [min:s]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Eclat-default</td>
<td>0</td>
<td>1:20</td>
</tr>
<tr>
<td>Eclat-hybrid, 4 rounds</td>
<td>0</td>
<td>2:37</td>
</tr>
<tr>
<td>Eclat-hybrid, 5 rounds</td>
<td>0</td>
<td>3:34</td>
</tr>
<tr>
<td>Eclat-hybrid, 10 rounds</td>
<td>0</td>
<td>16:39</td>
</tr>
<tr>
<td>Eclat-exhaustive, 500 s timeout</td>
<td>0</td>
<td>13:39</td>
</tr>
<tr>
<td>Eclat-exhaustive, 1000 s timeout</td>
<td>0</td>
<td>28:29</td>
</tr>
<tr>
<td>Eclat-exhaustive, 1500 s timeout</td>
<td>0</td>
<td>44:29</td>
</tr>
<tr>
<td>Eclat-exhaustive, 1750 s timeout</td>
<td>0</td>
<td>1:25:44</td>
</tr>
<tr>
<td>DSD-Crasher</td>
<td>3</td>
<td>1:59</td>
</tr>
</tbody>
</table>
Deeper than Eclat (DD)

JBoss JMS example

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

Groovy experiments

<table>
<thead>
<tr>
<th>Tool</th>
<th>Exception reports</th>
<th>Runtime [min:s]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Eclat-default</td>
<td>0</td>
<td>7:01</td>
</tr>
<tr>
<td>Eclat-hybrid, 4 rounds</td>
<td>0</td>
<td>8:24</td>
</tr>
<tr>
<td>Eclat-exhaustive, 2 rounds</td>
<td>2</td>
<td>10:02</td>
</tr>
<tr>
<td>Eclat-exhaustive, 500 s timeout</td>
<td>2</td>
<td>16:42</td>
</tr>
<tr>
<td>Eclat-exhaustive, 1200 s timeout</td>
<td>2</td>
<td>33:17</td>
</tr>
<tr>
<td>DSD-Crasher</td>
<td>4</td>
<td>30:32</td>
</tr>
</tbody>
</table>

DSD-Crasher benefits from deeper static analysis
Pointers

**JCrasher: An automatic robustness tester for Java**
http://www.cc.gatech.edu/jcrasher/

**Check ’n’ Crash: Combining static checking and testing**
27th International Conference on Software Engineering (ICSE 2005), pp. 422–431, May 2005
http://www.cc.gatech.edu/cnc/

**Dynamically discovering likely interface invariants**
28th International Conference on Software Engineering (ICSE 2006), Emerging Results Track, May 2006

**Download DSD-Crasher**
http://www.cc.gatech.edu/cnc/
Conclusions

Existing tests:
\[ m(1); \ m(2); \]

Dynamic

Run test to infer spec:
\[ i > 0 \]

Testee:
\[ m(\text{int } i) \]

Testee Annotated:
\[ \text{pre: } i > 0 \]

Static

Search for bug
\[ i > 5 \]

Dynamic

Run test to Confirm:
\[ i = 6 \]

New Test:
\[ \{ m(6); \} \]

New result:
\[ m \text{ crash} \]