Snugglebug
Work-In-Progress

Stephen Fink
Satish Chandra
Manu Sridharan

IBM T. J. Watson Research Center
March 28, 2008
SOFTWARE

Building a better bug-trap

Jun 19th 2003
From The Economist print edition

People who write it are human first and programmers only second—in short, they make mistakes, lots of them. Can software help them write better software?
What’s wrong with current bug finding tools?

1. False positives. Lots of them.

   - Mostly local pattern matching
   - Hundreds of “rules”
   - Fragile ad hoc ranking heuristics

So all we need is better analysis technology?
   - precise, scalable interprocedural analysis to move beyond local scope and eliminate false positives??

? What if God provided infinitely precise analysis?
What specifications do tools check?

Claim: If you read bug trackers, vast majority of critical defects discovered in the field are below the waterline.

```java
foo(x) {
    if (x == null) BOOM;
}
```

"The form did not resize correctly when using a Korean font."

"The ATM was not supposed to e-mail my PIN to my ex-wife."

Null Derefs

Buffer overflows

Misc. “rules”
Our goals:

1. **Eliminate FALSE ALARM**
   - Always generate concrete witnesses (JUnit tests)

2. **Attack BUGGY SPECS**
   - with analysis-driven feedback loop to acquire specifications
   - Reduce Costs
   - Increase Benefits

When a tool reports a finding, it means either:
- **BUGGY CODE**: The code is buggy.
- **BUGGY SPEC**: The specification is buggy.
- **FALSE ALARM**: The analysis is inexact.

Developer can step through with debugger, understand cause completely
Can add test to regression suite
Give up on verification

Specifications manifest as assertions in source language
Machine infers specifications.
Simple UI for developer to Accept/reject specifications
Generate counterexamples that violate accepted specifications
This sounds like ...

Agitator, Alloy, Boogie, CUTE, DART, Daikon, DIDUCE, DSD-Crasher, Dynamine, DySy, ESC, Korat, Java Pathfinder, JCrasher, jCUTE, Jex, JML, Houdini, MAPO, Metal, Miniatur, Perracotta, Pex, PreFIX, PR-Miner, Randoop, Saturn, SMART, TestEra, SPEC#, Symestra, Synergy,
Your Project (egregiously omitted) ...
Today’s workflow:

Code → Analysis → Report

Generic “rules”

NPE, OOB, etc
Snugglebug workflow:

Code → Accepted Specifications → Analysis → Suggested Specifications → Report and Tests

Accepted Specifications:

NPE, OOB, etc

Generic "rules"

Report and Tests
DEMO?
Technology Overview

Candidate identification

Identify program states (goals) we would like to reach

Witness Generation

Generate a unit test that reaches a goal state

Specification Acquisition

Acquire some formal specification of kosher and trrief (non-kosher) program states

Program Analysis
What are the risks?

Analysis Technology Inadequate

Concrete test case generation, respecting public APIs, over huge code bases, testing non-trivial properties

Can we really learn powerful specs? Can we express them in ways that a human will relate to?

Risk and reward

NPE, asserts

Typestate, contracts

Object constraints, global invariants

Functional specification
Analysis Technology

Candidate identification

Witness Generation

Specification Acquisition

Symbolic Search Via Weakest Precondition

Program Analysis
Symbolic Search via Weakest Precondition (Intro)

\[ \text{wp}(\phi) = (x-3 > 9) \land x > 7 \]

void foo (int x) {
    \phi = \text{wp}(\phi) = (x-3 > 9) \land x > 7
    if (x > 7) {
        \phi := \text{wp}(\phi) = \phi[x-3|y] = (x-3 > 9)
        int y = x - 3;
        \phi := \text{wp}(\phi) = (y > 9)
        if (y > 9) {
            \phi := \text{true}
            BOOM;
        }
    }
}
Reps-Horwitz-Sagiv POPL 95 Tabulation Solver (WALA)

- explore all paths at once, IPA with underapproximate abstraction

\[
\begin{align*}
\phi &: 1 > 3 \land 1 \leq 2 \\
x &= \min(1,2); \\
\phi &= x > 3 \\
y &= \min(x,3); \\
\phi &= x > 3 \\
z &= \min(x,4); \\
\phi &= \text{true} \\
\text{if (z > 3)} & \quad \phi = \wp(\phi) = (z > 3) \\
\text{BOOM;}
\end{align*}
\]

\[
\begin{align*}
\text{int min}(a, b) \{ \\
& \quad \phi := T \\
& \quad \phi_1 := a > 3 \land a \leq b \\
& \quad \text{if (a <= b)} \\
& \quad \quad r = a; \\
& \quad \quad \phi := a > 3 \\
& \quad \quad \text{else} \\
& \quad \quad \quad r = b; \\
& \quad \quad \quad \phi := b > 3 \\
& \quad \quad \text{return r;} \\
& \quad \phi := r > 3 \\
& \quad \phi := T
\end{align*}
\]

<table>
<thead>
<tr>
<th>(\phi)</th>
<th>(\text{Wp}(\text{min},\phi))</th>
</tr>
</thead>
<tbody>
<tr>
<td>(r &gt; 3)</td>
<td>(a &gt; 3 \land a \leq b)</td>
</tr>
<tr>
<td>(r &gt; 3)</td>
<td>(b &gt; 3 \land a &gt; b)</td>
</tr>
<tr>
<td>(T)</td>
<td>(T)</td>
</tr>
</tbody>
</table>
Effective Modular Analysis?

Tabulation is fully automatic

Maintain (large?) database of partial transfer functions
  Precompute partial predicate transformers for standard libraries
    • WP(true), WP(throws an exception)
    • WP(other common conditions?)

Key issue: **Separation.** What is the frame condition?
  “logical mod/ref”
  abstract interpretation

Open question: degree of reuse?
Dealing with exponential explosion
(Without even worrying about loops …)

Paths
if (c₁)
S₁; T₁;
if (c₂)
S₂; T₂;
if (c₃)
S₃; T₃;

Substitution run amuck (FS POPL02)
x = ...
y = x + x
z = y + y
w = z + z
v = w + w

Dynamic Dispatch
s = x.toString();
s += y.toString();
s += z.toString();
s += w.toString();

Aliasing and Destructive Updates
y.f = x;
z.f = y;
w.g = z;
Dealing with exponential explosion
Merge Functions & Search Heuristics

\begin{align*}
&y \geq 3 \land P \\
&x \geq 3 \land P \land \neg P \\
&y \geq 3 \\
&y \geq 3 \land P \land \neg P \\
&y \geq 3 \land \neg P \\
&\text{if (P)} \\
&y \geq 3 \land \neg P \\
&\text{if (P)} \\
&x \geq 3 \\
&x = y \\
&z = x \\
&z = y \\
&z \geq 3 \\
&z \geq 3 \land c_3 \\
&z \geq 3 \land \neg c_3 \\
&\text{if (c_3)} \\
&z \geq 3 \\
&g = 4; \\
&h = 5; \\
&\text{assert } z < 3
\end{align*}
Generating API-conformant test cases.

```java
wp(\phi) = (b.f == 1)
static void foo(Bar b) {
    if (b.getF() == 1) {
        BOOM;
    }
}
class Bar {
    private int f; // f == 0 or 2
    public int getF() { return f; }
    private Bar(int f) {
        this.f = f;
    }
    public static Bar make0() {
        return new Bar(0);
    }
    public static Bar make2() {
        return new Bar(2);
    }
}
```

Solution: Universal Driver

Encodes all reasonable ways of driving the method under test.

Parameterized in a way to facilitate search by an SMT solver.

Partial evaluation of universal driver w.r.t. a satisfying assignment gives a unit test.
Generating API-conformant test cases.

\[ wp(\phi) = (b.f \equiv 1) \]

```
static void foo(Bar b) {
  if (b.getF() \equiv 1) {
    BOOM;
  }
}
```

```
class Bar {
  private int f; // f \equiv 0 or 2
  public int getF() { return f; }
  private Bar(int f) {
    this.f = f;
  }
  public static Bar make0() {
    return new Bar(0);
  }
  public static Bar make2() {
    return new Bar(2);
  }
}
```

```
Universal Driver

public static void driveFoo(int[] x) {
  int length = x[0];
  int[] y = x[1:length];
  Bar b = makeBar(y);
  foo(b);
}
```

```
public static Bar makeBar(int[] y) {
  switch(y[0]) {
    case 0: return Bar.make0();
    case 1: return Bar.make2();
  }
}
```

SMT: no satisfying assignment for driveFoo().
Generating API-conformant test cases.

\[
wp(\phi) = (b.f == 1)
\]

```java
static void foo(Bar b) {
    if (b.getF() == 1) {
        BOOM;
    }
}

class Bar {
    private int f;
    public int getF() { return f; }
    private Bar(int f) {
        this.f = f;
    }
    public static Bar make0() {
        return new Bar(0);
    }
    public static Bar make2() {
        return new Bar(2);
    }
    public static Bar make(int y) {
        return new Bar(y);
    }
}
```

**Universal Driver**

```java
public static void driveFoo(int[] x) {
    int length = x[0];
    int[] y = x[1 ... length];
    Bar b = makeBar(y);
    foo(b);
}

public static Bar makeBar(int[] y) {
    switch(y[0]) {
        case 0: return Bar.make0();
        case 1: return Bar.make2();
        case 2: return Bar.make(y[1]);
    }
}
```

*SMT: satisfying assignment for driveFoo(): [2, 2, 1]*
Generating API-conformant test cases.

Partially evaluate `driveFoo()` w.r.t. [2, 2, 1]:

```java
public void testFoo() {
    Bar b = Bar.make(1);
    foo(b);
}
```

```
Universal Driver

public static void driveFoo(int[] x) {
    int length = x[0];
    int[] y = x[1 ... length];
    Bar b = makeBar(y);
    foo(b);
}

public static Bar makeBar(int[] y) {
    switch(y[0]) {
        case 0: return Bar.make0();
        case 1: return Bar.make2();
        case 2: return Bar.make(y[1]);
    }
}

SMT: satisfying assignment for driveFoo(): [2, 2, 1]
```
Other technologies of interest

Abstraction to guide search, skip loops/recursion
Speculation and dynamic checking
From WP to specifications
  Requires effective formulae simplification, not just satisfying assignments
  “lifting” predicates from points to larger scopes (e.g. invariants)
Lots of ways to improve specification acquisition
  Tests as specifications
  Mining client codes for example specifications
  Mining the web for specifications
  Other stuff to be invented
Milestone 1: We judge the snugglebug tool useful enough for us to adopt it into our own daily development.

Milestone 2: Somebody else judges the snugglebug tool useful enough us to adopt it.

Milestone n: Total world domination. Retire to Tahiti.
BACKUP SLIDES
public Object[] toArray(Object[] a)

Specified by:
toArray in interface Collection

Specifications:
also
public normal_behavior
old int cosize = this.theCollection.int_size();
old int arsize = a.length;
requires 0 <= null&cosize = 2147483647;
requires this.elementType == returnType(typeof(a));
requires [forall java.lang.Object o; this.contains(o); returnType(typeof(o))];
{ }

requires cosize <= arsize;
assignable a[];
ensures result = a;
ensures [forall int k; 0 <= NAK < cosize; this.theList.get(k) == result[k]];
ensures [forall int k; cosize <= NAK < arsize; result[k] == null];
also
requires cosize > arsize;
assignable Nothing;
ensures fresh(result) && result_length == cosize;
ensures [forall int k; 0 <= NAK < cosize; this.theList.get(k) == result[k]];
}

Specifications inherited from overridden method toArray(Object[]) in interface Collection:
also
public normal_behavior
old int cosize = this.theCollection.int_size();
old int arsize = a.length;
requires 0 <= null&cosize = 2147483647;
requires this.elementType == returnType(typeof(a));
requires [forall java.lang.Object o; this.contains(o); returnType(typeof(o))];
{ }

requires cosize <= arsize;
assignable a[];
ensures result = a;
ensures [forall int k; 0 <= NAK < cosize; this.theCollection.count[result[k]] == org.testng.models.JMArrayOps.valueEqualCount(result[k],cosize)];
ensures [forall int k; cosize <= NAK < arsize; result[k] == null];
ensures_redundantly returnType(result) == returnType();
also
requires cosize > arsize;
assignable Nothing;
ensures fresh(result) && result_length == cosize;
ensures [forall int k; 0 <= NAK < cosize; this.theCollection.count[result[k]] == org.testng.models.JMArrayOps.valueEqualCount(result[k],cosize)];
ensures [forall int k; 0 <= NAK < cosize; result[k] == null];
ensures returnType(result) == returnType();
also
public exceptional_behavior
requires 1 == null;
assignable Nothing;
signals_only java.lang.NullPointerException;
also
public behavior
requires 1 == null;
requires [forall java.lang.Object o; 1 != null&&this.contains(o); returnType(typeof(o))];
assignable a[];
signals_only java.lang.ArrayStoreException;
Everyone wants a piece of the pie ... and “Finding Bugs is Easy” ...
Typical Interaction between Analysis Tools and Developers

Your method `foo` can throw a null pointer exception at line 25

Oh really?

Yes, really, when the parameter `p` is such that `p.next == null`

Oh yeah?

Really. Here is a JUnit test case that exercises this bug

I know for sure that `p.next != null`
Have we changed the world yet?

Maturity is a bitter disappointment for which no remedy exists, unless laughter can be said to remedy anything.
- Vonnegut

These tools report a lot of things I don’t care about and few things I do care about.