Feedback-directed Random Test Generation

(Random testing)

Select inputs at random from a program’s input space
Check that program behaves correctly on each input

An attractive error-detection technique
- Easy to implement and use
- Yields lots of test inputs
- Finds errors
  - Miller et al. 1990: Unix utilities
  - Kropp et al. 1998: OS services
  - Forrester et al. 2000: GUI applications
  - Claessen et al. 2000: functional programs
  - Csallner et al. 2005,
    Pacheco et al. 2005: object-oriented programs
  - Groce et al. 2007: flash memory, file systems
Evaluations of random testing

- Theoretical work suggests that random testing is as effective as more systematic input generation techniques (Duran 1984, Hamlet 1990)
- Some empirical studies suggest systematic is more effective than random
  - Ferguson et al. 1996: compare with chaining
  - Marinov et al. 2003: compare with bounded exhaustive
  - Visser et al. 2006: compare with model checking and symbolic execution

  Studies are performed on small benchmarks, they do not measure error revealing effectiveness, and they use completely undirected random test generation.

Contributions

- We propose feedback-directed random test generation
  - Randomized creation of new test inputs is guided by feedback about the execution of previous inputs
  - Goal is to avoid redundant and illegal inputs

- Empirical evaluation
  - Evaluate coverage and error-detection ability on a large number of widely-used, well-tested libraries (780KLOC)
  - Compare against systematic input generation
  - Compare against undirected random input generation
Outline

- Feedback-directed random test generation
- Evaluation:
  - Randoop: a tool for Java and .NET
  - Coverage
  - Error detection
- Current and future directions for Randoop

Random testing: pitfalls

1. **Useful test**
   Set t = new HashSet();
   s.add("hi");
   assertTrue(s.equals(s));

2. **Redundant test**
   Set t = new HashSet();
   s.add("hi");
   s.isEmpty();
   assertTrue(s.equals(s));

3. **Useful test**
   Date d = new Date(2006, 2, 14);
   assertTrue(d.equals(d));

4. **Illegal test**
   Date d = new Date(2006, 2, 14);
   d.setMonth(-1); // pre: argument >= 0
   assertTrue(d.equals(d));

5. **Illegal test**
   Date d = new Date(2006, 2, 14);
   d.setMonth(-1);
   d.setDay(5);
   assertTrue(d.equals(d));

*do not output*

*do not even create*
Feedback-directed random test generation

- Build test inputs **incrementally**
  - New test inputs extend previous ones
  - In our context, a test input is a method sequence
- As soon as a test input is created, execute it
- Use execution results to **guide** generation
  - away from redundant or illegal method sequences
  - towards sequences that create new object states

Technique input/output

- **Input:**
  - classes under test
  - time limit
  - set of contracts
    - Method contracts (e.g. "o.hashCode() throws no exception")
    - Object invariants (e.g. "o.equals(o) == true")
- **Output:** contract-violating test cases. Example:

```
HashMap h = new HashMap();
Collection c = h.values();
Object[] a = c.toArray();
LinkedList l = new LinkedList();
l.addFirst(a);
TreeSet t = new TreeSet(l);
Set u = Collections.unmodifiableSet(t);
assertTrue(u.equals(u));
```

Example with contract violation:
```
// no contracts violated up to last method call
fails when executed
```
1. Seed components
   \[
   \text{components} = \{ \begin{array}{l}
   \text{int } i = 0; \\
   \text{boolean } b = \text{false}; \\
   \ldots
   \end{array} \}
   \]

2. Do until time limit expires:
   a. Create a new sequence
      i. Randomly pick a method call \( m(T_1, \ldots, T_k)/T_{ret} \)
      ii. For each input parameter of type \( T_i \), randomly pick a sequence \( S_i \) from the components that constructs an object \( v_i \) of type \( T_i \)
      iii. Create new sequence \( S_{\text{new}} = S_1; \ldots; S_k; T_{ret} \ v_{\text{new}} = m(v_1, \ldots, v_k) \)
      iv. If \( S_{\text{new}} \) was previously created (lexically), go to i
   b. Classify the new sequence \( S_{\text{new}} \)
      a. May discard, output as test case, or add to components

Classifying a sequence

```
start → execute and check contracts

contract violated? yes

no

components

sequence redundant? yes

no

minimize sequence

contract-violating test case

discard sequence
```
Redundant sequences

- During generation, maintain a set of all objects created.
- A sequence is redundant if all the objects created during its execution are members of the above set (using equals to compare)
- Could also use more sophisticated state equivalence methods
  - E.g. heap canonicalization

Outline

- Feedback-directed random test generation

Evaluation:
- Randoop: a tool for Java and .NET
- Coverage
- Error detection

- Current and future directions for Randoop
Randoop

- Implements feedback-directed random test generation

- **Input:**
  - An assembly (for .NET) or a list of classes (for Java)
  - Generation time limit
  - Optional: a set of contracts to augment default contracts

- **Output:** a test suite (Junit or Nunit) containing
  - Contract-violating test cases
  - Normal-behavior test cases

Randoop outputs oracles

- **Oracle for contract-violating test case:**
  ```java
  Object o = new Object();
  LinkedList l = new LinkedList();
  l.addFirst(o);
  TreeSet t = new TreeSet(l);
  Set u = Collections.unmodifiableSet(t);
  assertTrue(u.equals(u));             // expected to fail
  ```

- **Oracle for normal-behavior test case:**
  ```java
  Object o = new Object();
  LinkedList l = new LinkedList();
  l.addFirst(o);
  l.add(o);
  assertEquals(2, l.size());            // expected to pass
  assertEquals(false, l.isEmpty()); // expected to pass
  ```

*Randoop uses observer methods to capture object state*
Some Randoop options

- Avoid use of null
  
  statically...
  
  ```java
  Object o = new Object();
  LinkedList l = new LinkedList();
  l.add(null);
  ```

  ...and dynamically
  
  ```java
  Object o = returnNull();
  LinkedList l = new LinkedList();
  l.add(o);
  ```

- Bias random selection
  
  - Favor smaller sequences
  - Favor methods that have been less covered
  - Use constants mined from source code

Outline

- Feedback-directed random test generation

- Evaluation:
  
  - Randoop: a tool for Java and .NET
  - Coverage
  - Error detection

- Current and future directions for Randoop
**Coverage**

- Seven data structures (stack, bounded stack, list, bst, heap, rbt, binomial heap)

- Used in previous research
  - Bounded exhaustive testing [Marinov 2003]
  - Symbolic execution [Xie 2005]
  - Exhaustive method sequence generation [Xie 2004]

- All above techniques achieve high coverage in seconds

- Tools not publicly available

**Coverage achieved by Randoop**

- Comparable with exhaustive/symbolic techniques

<table>
<thead>
<tr>
<th>data structure</th>
<th>time (s)</th>
<th>branch cov.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bounded stack (30 LOC)</td>
<td>1</td>
<td>100%</td>
</tr>
<tr>
<td>Unbounded stack (59 LOC)</td>
<td>1</td>
<td>100%</td>
</tr>
<tr>
<td>BS Tree (91 LOC)</td>
<td>1</td>
<td>96%</td>
</tr>
<tr>
<td>Binomial heap (309 LOC)</td>
<td>1</td>
<td>84%</td>
</tr>
<tr>
<td>Linked list (253 LOC)</td>
<td>1</td>
<td>100%</td>
</tr>
<tr>
<td>Tree map (370 LOC)</td>
<td>1</td>
<td>81%</td>
</tr>
<tr>
<td>Heap array (71 LOC)</td>
<td>1</td>
<td>100%</td>
</tr>
</tbody>
</table>
Visser containers

- Visser et al. (2006) compares several input generation techniques
  - Model checking with state matching
  - Model checking with abstract state matching
  - Symbolic execution
  - Symbolic execution with abstract state matching
  - Undirected random testing
- Comparison in terms of branch and predicate coverage
- Four nontrivial container data structures
- Experimental framework and tool available

Predicate coverage

- Binary tree
- Binomial heap
- Fibonacci heap
- Tree map
Outline

- Feedback-directed random test generation

- Evaluation:
  - Randoop: a tool for Java and .NET
  - Coverage
  - Error detection

- Current and future directions for Randoop

Subjects

<table>
<thead>
<tr>
<th>Subjects</th>
<th>LOC</th>
<th>Classes</th>
</tr>
</thead>
<tbody>
<tr>
<td>JDK (2 libraries)</td>
<td>53K</td>
<td>272</td>
</tr>
<tr>
<td>(java.util, javax.xml)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Apache commons (5 libraries)</td>
<td>114K</td>
<td>974</td>
</tr>
<tr>
<td>(logging, primitives, chain jelly, math, collections)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>.Net framework (5 libraries)</td>
<td>582K</td>
<td>3330</td>
</tr>
</tbody>
</table>
Methodology

- Ran Randoop on each library
  - Used default time limit (2 minutes)

- Contracts:
  - o.equals(o)==true
  - o.equals(o) throws no exception
  - o.hashCode() throws no exception
  - o.toString() throw no exception
  - No null inputs and:
    - Java: No NPEs
    - .NET: No NPEs, out-of-bounds, of illegal state exceptions

Results

<table>
<thead>
<tr>
<th>Library</th>
<th>Test Cases output</th>
<th>Error-revealing tests cases</th>
<th>Distinct errors</th>
</tr>
</thead>
<tbody>
<tr>
<td>JDK</td>
<td>32</td>
<td>29</td>
<td>8</td>
</tr>
<tr>
<td>Apache commons</td>
<td>187</td>
<td>29</td>
<td>6</td>
</tr>
<tr>
<td>.Net framework</td>
<td>192</td>
<td>192</td>
<td>192</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>411</strong></td>
<td><strong>250</strong></td>
<td><strong>206</strong></td>
</tr>
</tbody>
</table>
Errors found: examples

- JDK Collections classes have 4 methods that create objects violating `o.equals(o)` contract
- Javax.xml creates objects that cause `hashCode` and `toString` to crash, even though objects are well-formed XML constructs
- Apache libraries have constructors that leave fields unset, leading to NPE on calls of `equals`, `hashCode` and `toString` (this only counts as one bug)
- Many Apache classes require a call of an `init()` method before object is legal—led to many false positives
- .Net framework has at least 175 methods that throw an exception forbidden by the library specification (NPE, out-of-bounds, or illegal state exception)
- .Net framework has 8 methods that violate `o.equals(o)`
- .Net framework loops forever on a legal but unexpected input

JPF

- Used JPF to generate test inputs for the Java libraries (JDK and Apache)
  - Breadth-first search (suggested strategy)
  - max sequence length of 10
- JPF ran out of memory without finding any errors
  - Out of memory after 32 seconds on average
  - Spent most of its time systematically exploring a very localized portion of the space
- For large libraries, random, sparse sampling seems to be more effective
Undirected random testing

- JCrasher implements undirected random test generation
- Creates random method call sequences
  - Does not use feedback from execution
- Reports sequences that throw exceptions
- Found 1 error on Java libraries
  - Reported 595 false positives

Regression testing

- Randoop can create regression oracles
- Generated test cases using JDK 1.5
  - Randoop generated 41K regression test cases
- Ran resulting test cases on
  - JDK 1.6 Beta
    - 25 test cases failed
  - Sun’s implementation of the JDK
    - 73 test cases failed
  - Failing test cases pointed to 12 distinct errors
  - These errors were not found by the extensive compliance test suite that Sun provides to JDK developers

```java
Object o = new Object();
LinkedList l = new LinkedList();
l.addFirst(o);
l.add(o);
assertEquals(2, l.size());            // expected to pass
assertEquals(false, l.isEmpty()); // expected to pass
```
Evaluation: summary

- Feedback-directed random test generation:
  - Is effective at finding errors
    - Discovered several errors in real code (e.g. JDK, .NET framework core libraries)
  - Can outperform systematic input generation
    - On previous benchmarks and metrics (coverage), and
    - On a new, larger corpus of subjects, measuring error detection
  - Can outperform undirected random test generation

Outline

- Feedback-directed random test generation
- Evaluation:
  - Randoop: a tool for Java and .NET
  - Coverage
  - Error detection
- Current and future directions for Randoop
Tech transfer

- Randoop is currently being maintained by a product group at Microsoft
  - Spent an internship doing the tech transfer
    - How would a test team actually use the tool?
      - Push-button at first, desire more control later
    - Would the tool be cost-effective?
      - Yes
      - Immediately found a few errors
      - With more control, found more errors
      - Pointed to blind spots in
        - existing test suites
        - Existing automated testing tools
  - Which heuristics would be most useful?
    - The simplest ones (e.g. uniform selection)
    - More sophisticated guidance was best left to the users of the tool

Future directions

- Combining random and systematic generation
  - DART (Godefroid 2005) combines random and systematic generation of test data
  - How to combine random and systematic generation of sequences?
- Using Randoop for reliability estimation
  - Random sampling amenable to statistical analysis
  - Are programs that Randoop finds more problems with more error-prone?
- Better oracles
  - To date, we have used a very basic set of contracts
  - Will better contracts lead to more errors?
  - Incorporate techniques that create oracles automatically
Conclusion

- Feedback-directed random test generation
  - Finds errors in widely-used, well-tested libraries
  - Can outperform systematic test generation
  - Can outperform undirected test generation
- Randoop:
  - Easy to use—just point at a set of classes
  - Has real clients: used by product groups at Microsoft
- A mid-point in the systematic-random space of input generation techniques