Feedback-directed Random Test Generation

(ICSE '07 with minor edits)

Carlos Pacheco
Michael Ernst

MIT

Shuvendu Lahiri
Thomas Ball

Microsoft Research

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:

no contracts violated up to last method call

```
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));
```

fails when executed

Technique

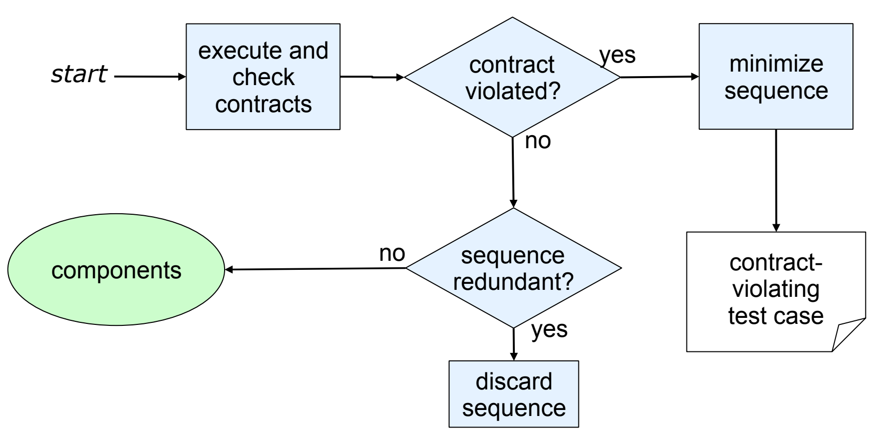
1. Seed components

   components = { `int i = 0;` `boolean b = false;` ... }

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 Ti
      iii. Create new sequence S_{new} = S_1; ... ; S_k ; T_{ret} v_{new} = m(v1...vk);
      iv. if S_{new} was previously created (lexically), go to i

   b. Classify the new sequence S_{new}
      a. May discard, output as test case, or add to components

Classifying a 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:

```
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:

```
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…

```
Object o = new Object();
LinkedList l = new LinkedList();
l.add(null);
```

…and dynamically

```
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

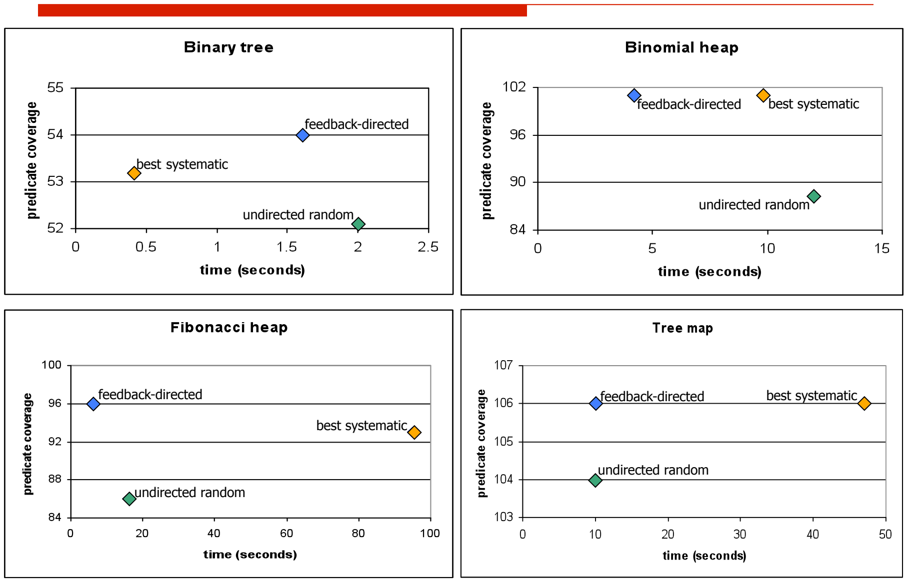
- Comparable with exhaustive/symbolic techniques

data structure	time (s)	branch cov.
Bounded stack (30 LOC)	1	100%
Unbounded stack (59 LOC)	1	100%
BS Tree (91 LOC)	1	96%
Binomial heap (309 LOC)	1	84%
Linked list (253 LOC)	1	100%
Tree map (370 LOC)	1	81%
Heap array (71 LOC)	1	100%

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



Outline

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

Subjects

	LOC	Classes
JDK (2 libraries) (java.util, javax.xml)	53K	272
Apache commons (5 libraries) (logging, primitives, chain jelly, math, collections)	114K	974
.Net framework (5 libraries)	582K	3330

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

	test cases output	error-revealing tests cases	distinct errors
JDK	32	29	8
Apache commons	187	29	6
.Net framework	192	192	192
Total	***411***	***250***	***206***

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

```
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 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

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