Contract-based Testing

Yi Wei

Material also from Ilinca Ciupa

2.12.2009
Agenda

• Contract based random testing

• Variations to the original random testing
  – Precondition satisfaction by guided object selection
  – Adaptive random testing
Automated unit testing

- Input generation
- Test case execution
- Result validation

Preconditions

Postconditions

Contracts
Contract-based testing

As long as we know what the software is supposed to do, we do not need any human intervention to test it.

When testing a certain method:

- We try to satisfy its **precondition** (so that we can execute it).
- We hope it will not fulfill its **postcondition**.
put (v: G; i: INTEGER_32 )
  -- From DS_ARRAYED_LIST
  -- Add `v' at `i'-th position.

require
  extendible: extendible (1)
  valid_index: 1 <= i and i <= (count + 1)

-- Implementation

ensure
  one_more: count = old count + 1
  inserted: item (i) = v
Contract-based random testing

Random input generation:

• Primitive values: random* selection
  • 25% probability: select from 0, ±1, ±2, ...
  • 75% probability: randomly choose a value

• Objects: constructor calls + other (state-changing) methods
  • 20% probability: create a new instance
  • 80% probability: reuse old objects
Random testing strategy

Workflow of the random testing strategy

1. Select next routine to test
2. Select objects randomly
3. Invoke routine

Sample test cases

Object pool

create {LINKED_LIST[INTEGER]} v1.make

v2 := 1
v1.extend (v2)
v3 := 125
v1.wipe_out
v4 := v1.has (v3)
v5 := v1.count
This strategy found thousands of faults in production software!
Test case execution results

Three type of results:

- Pass – all OK 😞

- Fail – contract violation / other exception => bug 😊

- Invalid – test case not executed
The issue of generating precondition satisfying tests

A random based testing tool implemented in such scheme has difficulty in generating valid test cases for precondition-equipped routines:

• Some routines are left untested.

• The testing tool may keep generating invalid test cases, instead of performing effective testing.
What kinds of preconditions are difficult to satisfy?

```
remove_right_cursor (a_cursor: DS_ARRAYED_LIST_CURSOR )
  -- Remove item to right of `a_cursor' position.
  -- Move any cursors at this position forth.

require
  not_empty: not is_empty
  cursor_not_void: a_cursor /= Void
  valid_cursor: valid_cursor (a_cursor)
  not_after: not a_cursor.after
  not_last: not a_cursor.is_last
```

At the beginning of the 50th minute, there are 356 list objects and 192 cursor objects, but only 5 out of 68,352 list-cursor combinations satisfied the precondition, the probability of a correct selection is 0.007%.
What kinds of preconditions are difficult to satisfy?

```
prune (n: INTEGER_32; i: INTEGER_32)
    -- Remove `n' items at and after `i'-th position.
require
    valid_index: 1 <= i and i <= count
    valid_n: 0 <= n and n <= (count - i + 1)
ensure
    new_count: count = old_count - n
```

This occurs often in preconditions
Guided object selection – the \textit{ps-strategy}

Observation

- The or-strategy can create objects satisfying many preconditions
- Needs to select those objects more effectively

Solution: the precondition satisfaction strategy (ps-strategy)

- Keep track of which objects satisfy certain precondition predicates
- To test a routine, select precondition-satisfying objects with a higher probability
- Use linear constraint solving
Comparison between the or-strategy and the ps-strategy

The or-strategy

Select next routine to test

Select objects randomly

Invoke routine

The ps-strategy

Select next routine to test

Pr

Select objects randomly

Select precondition-satisfying objects from predicate evaluation pool

Invoke routine

Update predicate evaluation pool
Object selection guided by predicate evaluation pool (V-pool)

The V-pool keeps track of objects satisfying certain precondition predicates; those objects can be used to construct valid test inputs.

- **not_empty**
  - l2
  - l1

- **cursor_not_void**
  - c1
  - c2
  - c3

- **valid_cursor**
  - l1, c1
  - l2, c2

- **not_after**
  - c2
  - c1

- **not_last**
  - c3
  - c2
Updating the predicate evaluation pool

After every *passing* test case:

evaluate relevant predicates, add precondition-satisfying object combinations to the V-pool.

Grow the V-pool as much as possible

After every *invalid* test case:

remove the object combination causing the precondition violation at the specific predicate from the V-pool.

Correct inconsistency lazily
After every passing test case...

```c
replace_at_cursor (v: G; a_cursor: CURSOR)
    -- Replace item at `a_cursor' position by `v'.

require
    cursor_not_void: a_cursor /= Void
    valid_cursor: valid_cursor (a_cursor)
    not_off: not a_cursor off
```

The V-pool contains *snapshots* of the relations among objects, this information may become inconsistent as testing proceeds.

```c
new_cursor (l)
    c := l.new_cursor
    c.go_i_th (1)
    l.wipe_out
    l.replace_at_cursor (v3, c)
```

...
After every invalid test case...

```plaintext
replace_at_cursor (v: G; a_cursor: CURSOR)
    -- Replace item at `a_cursor' position by `v'.
require
    cursor_not_void: a_cursor /= Void
    valid_cursor: valid_cursor (a_cursor)
    not_off: not a_cursor.off
```

> What is the success rate of test cases generated by the ps-strategy?

> > 60% (cf. or-strategy: < 10%)
For linear constraints

```haskell
prune \( n: \text{INTEGER}_32; i: \text{INTEGER}_32 \)

-- Remove `n' items at and after `i'-th position.

require

valid_index: \( 1 \leq i \leq \text{count} \)
valid_n: \( 0 \leq n \leq (\text{count} - i + 1) \)

ensure

new_count: \( \text{count} = \text{old count} - n \)
```

Lpsolve is used to generate a minimal and a maximal solution

- Randomly select one value from the range
- Slightly biased on border values and potentially interesting values
- Solutions are cached
Optimization

Always enforcing precondition satisfaction slows down the test process by (50~70%), without benefits:

- did not test more routines
- found much fewer faults

Turn precondition satisfaction on only from time to time
Evaluation overview

- How many more routines are tested by the ps-strategy?
- How often can routines be tested?
- How many more faults are detected by the ps-strategy?
- How fast is the ps-strategy?
Experimental setup

• 92 classes of EiffelBase and Gobo libraries
  – widely used in production software
  – different data structures: lists, arrays, trees, stacks, even a regex lexer

• Arranged into 57 strongly-related test groups
  – based on dependency between classes
  – introduces more diversity in the object pool

• 30 test runs per group of 1 hour each

• For both the or- and ps-strategies

• 3,420 hours worth of testing
How many more routines are tested by the ps-strategy?

- A hard routine is one for which or-strategy failed to generate a valid test case for at least 90% of the time.

Coverage of hard routines

- ps-strategy covers 56% of the routines missed by or-strategy.

- But misses 1% of those tested by or-strategy.
How often are routines tested by the ps-strategy?

- Over 3.5 times as many valid test cases overall
How many more faults are detected by the ps-strategy?

3 groups with over 30% more faults

More faults detected by ps-strategy

More faults detected by or-strategy

Same number of faults detected by both strategies

28 groups

19 groups

10 groups
Fault coverage by each strategy

- Almost 10% increase in the number of detected faults overall
Fault coverage by each strategy

- Different class groups perform differently well
Fault detection probability

• What is the probability of a strategy to detect a given fault in a single test run?

• The higher the probability, the less runs are needed to detect that fault.

• Fault Detection Probability of fault $f$ using strategy $s$:

$$D(f, s) = \frac{N(f, s)}{R}$$

- $N(f, s)$: number of test runs in which $f$ was detected under strategy $s$
- $R$: number of test run per class group

$R = 30$ in our case
Fault detection probability: behavior of both strategies

- Very similar behavior between both strategies
- But does not mean that the probability is the same under both strategies
Fault detection probability: ps-strategy vs. or-strategy

- ps-strategy does a better job at finding faults systematically

\[ D(f, ps) - D(f, or) \]
Test case generation speed

- Fastest
- 0.03% overhead
- Slowest
Routines still untested by the ps-strategy

- **Strategy-unrelated (51%)**
  - Preconditions are hardcoded as unsatisfiable
  - Preconditions require a different environment (e.g. .NET)

- **Strategy-related (49%)**
  - Satisfying combinations are never created (bad luck)
  - Satisfying combinations are damaged before usage
  - Test runs are not long enough
Adaptive Random Testing

So far, we are happy if the random strategy generates one precondition satisfying input.

But if the input for a routine is always the same, the testing cannot be very effective.

Make test inputs spread out evenly over the range.
Adaptive Random Testing

Basic idea: random testing is likely to find faults faster when inputs are evenly spread over the range of possible values.

When first introduced, ART is only applicable to numeric inputs.

Reduces the number of tests generated until the first fault is found by as much as 50%.
Extending ART to O-O (ARToo)

We need to measure “how far” objects are from each other, in other words we need an object distance.

In general objects are characterized by:

• their attribute values
• their dynamic types
• recursively the primitive values of the attributes or the objects referred by attributes of reference types

[Ciupa et al., ICSE 2008]
Object distance

\[ p \leftrightarrow q = \text{combination} \]

\[
\begin{align*}
\text{elementary\_distance}(p, q), \\
\text{type\_distance}(\text{type}(p), \text{type}(q)), \\
\text{field\_distance}(\{ [p.a \leftrightarrow q.a] \mid a \in \text{Attributes(type}(p), \text{type}(q)) \})
\end{align*}
\]

In general objects are characterized by:

- their attribute values
- their dynamic types
- recursively the primitive values of the attributes or the objects referred by attributes of reference types
Elementary distance

\[ p \leftrightarrow q = \text{combination(}
\begin{align*}
&\text{elementary_distance}(p, q), \\
&\text{type_distance}(\text{type}(p), \text{type}(q)), \\
&\text{field_distance}(\{[p.a \leftrightarrow q.a] \\
&| a \in \text{Attributes}(\text{type}(p), \text{type}(q))\})
\end{align*}
\]

- For numbers: \( F(|p-q|) \)
- For characters: 0 if identical, C otherwise
- For booleans: 0 if identical, B otherwise
- For strings: the Levenshtein distance
- For references: 0 if identical, R if different but none is void, V if only one of them is void
Type distance

\[ p \leftrightarrow q = \text{combination(} \]
\[ \text{elementary_distance}(p, q), \]
\[ \text{type_distance}(\text{type}(p), \text{type}(q)), \]
\[ \text{field_distance}([p.a \leftrightarrow q.a] \]
\[ | a \in \text{Attributes}(\text{type}(p), \text{type}(q))) \]
\[ ) \]

\[ \text{type_distance}(t, u) = \lambda \times \text{path_length}(t, u) + \nu \times \sum_{a \in \text{non_shared}(t, u)} \text{weight}_a \]

- \text{path_length}: the minimum path length to a closest common ancestor.
- \text{non_shared}: the set of non-shared features.
- \text{weight}_a: denotes the weight associated with attribute a.
Field distance

\[ p \leftrightarrow q = \text{combination(} \]
\[ \text{elementary_distance}(p, q), \]
\[ \text{type_distance}(\text{type}(p), \text{type}(q)), \]
\[ \text{field_distance}([[p.a \leftrightarrow q.a] \]
\[ | a \in \text{Attributes(}\text{type}(p), \text{type}(q))\text{]})) \]

\[
\text{field_distance}(p, q) = \sum_{a} \text{weight}_{a} \times (p.a \leftrightarrow q.a)
\]

Terminates after a fixed number of steps (2 in our experiments)
ARTOO idea

- Test inputs (objects) generated randomly: candidate set.

- At every step, the element from the candidate set is chosen which has the maximum average distance to the already used ones.
m (arg1: A; arg2: B) in class C

Available instances of A
Instances of A used as first arg. in a call to m

v2

v16.m(v2, v5)

Available instances of B
Instances of B used as second arg. in a call to m

v5

Available instances of C
Instances of C used as target of a call to m

v16
<table>
<thead>
<tr>
<th>Class</th>
<th>LOC</th>
<th>#methods</th>
<th>#attributes</th>
<th>#parents</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACTION_SEQ</td>
<td>2477</td>
<td>156</td>
<td>16</td>
<td>24</td>
</tr>
<tr>
<td>ARRAY</td>
<td>1208</td>
<td>86</td>
<td>4</td>
<td>11</td>
</tr>
<tr>
<td>ARRAYED_LIST</td>
<td>2164</td>
<td>39</td>
<td>6</td>
<td>23</td>
</tr>
<tr>
<td>BOUNDED_STACK</td>
<td>779</td>
<td>62</td>
<td>4</td>
<td>10</td>
</tr>
<tr>
<td>FIXED_TREE</td>
<td>1623</td>
<td>125</td>
<td>6</td>
<td>4</td>
</tr>
<tr>
<td>HASH_TABLE</td>
<td>1791</td>
<td>122</td>
<td>13</td>
<td>9</td>
</tr>
<tr>
<td>LINKED_LIST</td>
<td>1893</td>
<td>106</td>
<td>6</td>
<td>19</td>
</tr>
<tr>
<td>STRING</td>
<td>2980</td>
<td>171</td>
<td>4</td>
<td>16</td>
</tr>
<tr>
<td>Average</td>
<td>1874</td>
<td>108</td>
<td>7</td>
<td>14</td>
</tr>
</tbody>
</table>
Test cases to first fault
Time to first fault

![Bar chart showing time to first fault for different data structures.

- **ACTION_SEQUENCE**: ARTOO 130, RAND 120
- **ARRAY**: ARTOO 150, RAND 140
- **ARRAYED_LIST**: ARTOO 160, RAND 150
- **BOUNDDED_STACK**: ARTOO 140, RAND 130
- **FIXED_TREE**: ARTOO 160, RAND 150
- **HASH_TABLE**: ARTOO 170, RAND 160
- **LINKED_LIST**: ARTOO 120, RAND 110
- **STRING**: ARTOO 140, RAND 130

Legend:
- **ARTOO**
- **RAND**


Conclusions

• Random testing is effective.

• Many variations are possible:
  – Precondition satisfaction by guided object selection
  – Adaptive random testing

Many researches are going on in the Chair of Software Engineering.
Questions?
Final formula to calculate object distance

\[ p \leftrightarrow q = \frac{1}{3} \times ( \]
\[ \text{norm(elementary\_distance}(p, q)) + \]
\[ \text{norm(path\_length}(\text{type}(p), \text{type}(q)) + \]
\[ \sum_{a \in \text{non\_shared}(\text{type}(p), \text{type}(q))} \text{weight}_a ) + \]
\[ \text{norm}(\frac{1}{2} \times \sum_{a} \text{weight}_a \times (p.a \leftrightarrow q.a))) \]
\[ \text{norm}(x) = (1 - \frac{1}{1 + x}) \times \text{max\_distance} \]