AutoTest: Contract-based testing

Yi Wei, Andreas Leitner, Ilinca Ciupa, Bertrand Meyer, Manuel Oriol, Alexander Pretschner, Arno Fiva

Chair of Software Engineering
ETH Zurich
Automated unit testing

Input generation → Test case execution → Result validation

Preconditions

Contracts

Postconditions
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

• Objects: constructor calls + other (state-changing) methods
Random testing strategy

- Select next routine to test
- Select objects randomly
- Invoke routine

Sample test cases

Object pool

```plaintext
create {LINKED_LIST[INTEGER]} v1.make
v2 := 1
v1.extend (v2)
v3 := 125
v1.wipe_out
v4 := v1.has (v3)
v5 := v1.count
```
Test outcome for the feature under test

• Execution ends normally: a passing test case

• Execution fails with precondition violation: an invalid test case

• Execution fails with postcondition violation or any failure inside feature body: a detected fault
Effectiveness of contract-based random testing

Intuition:

random testing is a poor strategy.

Experimental results:

- Random testing is effective
- Best: random$^+$ testing (random + limit values)
- Relative number of found faults: predictable
- Actual found faults: unpredictable
Number of faults found in time

Inversely proportional to elapsed time: \( f(t) = \frac{a}{t} + b \)
Number of faults found in classes

<table>
<thead>
<tr>
<th>Difference in number of bugs</th>
<th>Occurrences</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>161</td>
</tr>
<tr>
<td>2</td>
<td>99</td>
</tr>
<tr>
<td>3</td>
<td>94</td>
</tr>
<tr>
<td>4</td>
<td>47</td>
</tr>
<tr>
<td>5</td>
<td>22</td>
</tr>
<tr>
<td>6</td>
<td>10</td>
</tr>
<tr>
<td>7</td>
<td>5</td>
</tr>
<tr>
<td>8</td>
<td>4</td>
</tr>
<tr>
<td>9</td>
<td>6</td>
</tr>
<tr>
<td>10</td>
<td>2</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
</tr>
<tr>
<td>12</td>
<td>1</td>
</tr>
<tr>
<td>13</td>
<td>0</td>
</tr>
<tr>
<td>14</td>
<td>1</td>
</tr>
<tr>
<td>15</td>
<td>0</td>
</tr>
<tr>
<td>16</td>
<td>1</td>
</tr>
<tr>
<td>17</td>
<td>0</td>
</tr>
<tr>
<td>18</td>
<td>1</td>
</tr>
<tr>
<td>19</td>
<td>1</td>
</tr>
</tbody>
</table>
Problems with random testing

- Some features are not tested because their precondition are not satisfied:
  
- Randomly generated test data is not diversified enough.
The issue of generating precondition-satisfying tests

A random based testing tool implemented in the original scheme (or-strategy) 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?

```haskell
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 50\textsuperscript{th} 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?

```haskell
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

\begin{itemize}
\item The or-strategy can create objects satisfying many preconditions
\item Needs to select those objects more effectively
\end{itemize}

Solution: the precondition satisfaction strategy (ps-strategy)

\begin{itemize}
\item Keep track of which objects satisfy certain precondition predicates
\item To test a routine, select precondition-satisfying objects with a higher probability
\item Use linear constraint solver
\end{itemize}
Comparison between the or-strategy and the ps-strategy

The or-strategy

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

The ps-strategy

1. Select next routine to test
2. Pr
3. Select objects randomly
   - Select precondition-satisfying objects from predicate evaluation pool
4. Invoke routine
5. Update predicate evaluation pool
The V-pool keeps track of objects satisfying certain precondition predicates; those objects can be used to generate valid test cases.
Updating the predicate evaluation pool

After every *passing* test case

evaluate relevant predicates on *last used objects*, and 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...

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.

l := new_cursor

l.go_i_th (1)

l.wipe_out

l.replace_at_cursor (v3, c)

The V-pool contains snapshots of the relations among objects, this information may become inconsistent as testing proceeds.
After every invalid test case...

\[
\text{replace\_at\_cursor} \ (v: G; \ a\_cursor: \ \text{CURSOR})
\]

-- Replace item at `a\_cursor' position by `v'.

\[
\text{require}
\]

\[
\text{cursor\_not\_void: } a\_cursor \neq \text{Void}
\]
\[
\text{valid\_cursor: valid\_cursor \ (a\_cursor)}
\]
\[
\text{not\_off: } \text{not} \ a\_cursor\_off
\]

\[
\text{\_force\_last} \ (v1)
\]

\[
\text{What is the success rate of test cases generated by the ps-strategy?}
\]

\[
\text{\_wipe\_out}
\]

\[
\text{\_replace\_at\_cursor} \ (v3, c)
\]

\[
> 60\% \ (\text{cf. or-strategy: } < 10\%)
\]
For linear constraints

```haskell
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
```

**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
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
- **or-strategy** covers 59%
- **ps-strategy** covers 81%
How often are routines tested by the ps-strategy?

• Over 3.5 times as many valid test cases overall
Fault coverage by each strategy

Almost 10% increase in the number of detected faults overall.
Test case generation speed

- Fastest
- 0.03% overhead
- Slowest
Number of faults detected in time

![Graph showing the number of faults detected over time for two different test runs.](image)
Randomly generated inputs are not diversified

\[ r \left( m, n: \text{int} \right) \]

PERSON
  - name: STRING
  - age: INTEGER
  - spouse: PERSON
  - children: ARRAY [PERSON]
  - bank_account: BANK_ACCOUNT ...
Object distance

Objects characterized by:

- their values
- their dynamic types
- recursively the basic values of the attributes or the objects referred by the attributes
• Adaptive Random Testing for Object-Oriented Software

- 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
ARTOO algorithm

\( r \ (\text{arg1} : A; \ \text{arg2} : B) \) in class C

Available instances of A

Instances of A used as first arg. in a call to \( r \)

\( v_16 \cdot r(v_2,v_5) \)

Available instances of B

Instances of B used as second arg. in a call to \( r \)

v5

Available instances of C

Instances of C used as target of a call to \( r \)

v16
Tests to First Fault

![Graph showing tests to first fault with categories such as ACTION_SEQUENCE, ARRAY, BOUNDED_LIST, FIXED_TREE, HASH_TABLE, LINKED_LIST, STRING. The graph compares ARTOO and RAND.]
Extracted tests when debugging

Automatically turn failed executions into tests.
Test Generation + Test Extraction = AutoTest
AutoTest Demo
Research opportunities

Contract-based testing is an active research area, with many applications.

Master thesis, semester thesis projects are open. Check http://se.inf.ethz.ch/projects
Inferring contracts from passing tests

```plaintext
LINKED_LIST.extend (v: ANY)
    -- Add `v' to end. Do not move cursor.

ensure

Programmer written
    post1: occurrences (v) = old (occurrences (v)) + 1

Automatically inferred
    post2: forall o . o /= v implies occurrences (o) = old occurrences (o)
    post3: forall o . o /= v implies has (o) = old has (o)
    post4: forall i . i >= 1 and i <= old count implies i_th (i) = old i_th (i)
    post5: i_th (old count + 1) = v
    post6: old after implies index = old index + 1
    post7: not old after implies index = old index
    post8: count = old count + 1
    post9: last = v
```
Explaining faults revealed in failing tests

`extend (other: DS_LINEAR [G]; i: INTEGER_32)`

-- Add items of `other` at `i`-th position. Keep items of `other`
-- in the same order. Do not move cursors.

In several hours, AutoTest generated 144 test cases, most of them are passing, some of them are failing.

```
Current = other

Current.is_empty

True

2 passing

False

136 passing

16 failing
```

`DS_LINKED_LIST.extend.c4.b20`
Fixing faults revealed in failing tests

\[ \text{duplicate } (n: \text{INTEGER}): \ldots \]
\[
\text{do}
\text{pos} := \text{cursor}
\text{Result} := \text{new\_chain}
\text{Result.forth}
\text{from until } (\text{counter}=n) \text{ or after } \text{loop}
\text{Result.put\_left } (\text{item})
\text{forth}
\text{counter} := \text{counter} + 1
\text{end}
\text{go\_to } (\text{pos})
\text{end}
\]

\[ \text{Faulty version} \]

\[ \text{Fixed version} \]
Questions?