Satisfying Test Preconditions through Guided Object Selection

Yi Wei, Serge Gebhardt, Bertrand Meyer and Manuel Oriol

ETH Zürich
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
Original random testing strategy – the or-strategy

The or-strategy

Sample test cases

Object pool

create \{\texttt{LINKED\_LIST}[\texttt{INTEGER}]\} \ v1.make

\[v2 := 1\]

\[v1.extend(v2)\]

\[v3 := 125\]

\[v1.wipe\_out\]

\[v4 := v1.has(v3)\]

\[v5 := v1.count\]
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?

\[\text{remove_right_cursor (a_cursor: DS_ARRAYED_LIST_CURSOR)}\]

-- Remove item to right of `a_cursor' position.
-- Move any cursors at this position forth.

\[\text{require}\]

\[\text{not_empty: not is_empty}\]
\[\text{cursor_not_void: a_cursor /= Void}\]
\[\text{valid_cursor: valid_cursor (a_cursor)}\]
\[\text{not_after: not a_cursor.after}\]
\[\text{not_last: not a_cursor.is_last}\]

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?

```
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 *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 solver
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 generate valid test cases.

- **not_empty**
  - l2
  - l1

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

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

- **not_after**
  - c2
  - c1

- **not_last**
  - c3
  - c2

V-pool

Object pool
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...

```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)
```

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

```plaintext
force_last (v1)
```
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

\[
\text{prune} \ (n: \text{INTEGER}_32; \ i: \text{INTEGER}_32) \\
\quad \text{-- Remove `n' items at and after `i'-th position.}
\]

\[\begin{align*}
\text{require} \\
\quad \text{valid\_index}: \ 1 &\leq i \text{ and } i \leq \text{count} \\
\quad \text{valid\_n}: \ 0 &\leq n \text{ and } n \leq (\text{count} - i + 1)
\end{align*}\]

\[\text{ensure} \]
\[\text{new\_count}: \ \text{count} \ = \ \text{old count} - n\]

\textit{lp.solve} 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

ps-strategy vs. or-strategy
Evaluation overview

• How many more routines are tested by the ps-strategy?

• How often are routines tested by the ps-strategy?

• 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, and 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 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
- or-strategy covers 59%
- ps-strategy covers 81%

- 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?

- 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
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
Limitations to generalization

- The chosen classes are mostly data structures and might not be representative for all O-O programs.

- One-hour test runs might be too short, the number of faults does not reach a plateau.
Conclusion: ps-strategy vs. or-strategy

• How many more routines are tested by the ps-strategy?
  – The ps-strategy tests 56% of the routines missed by the or-strategy.

• How often are routines tested by the ps-strategy?
  – The ps-strategy tests routines over 3.5 times as often.

• How many more faults are detected by the ps-strategy?
  – The ps-strategy finds 10% more faults than the or-strategy.

• How fast is the ps-strategy?
  – The ps-strategy has negligible overhead (a mere 0.03%).
Questions
Importance of speed

• More valid test cases
  ⇒ more diversified object pool
  ⇒ greater chances of finding faults

• Tried two other variations:
  – Iterating through all objects in the pool,
    overhead >50% (even with optimizations)
  – Always enforcing precondition satisfaction,
    big overhead
Success rate of the ps-strategy

- varies from as low as 20% to as high as 99%
- mostly over 40%
- generally decreasing because hard routines are favored
Distribution of fault detection
Optimization

As a tradeoff, the precondition satisfaction is only turned on for routine $r$ from time to time:

$$P_r(t, d) = \left( 1 - \frac{t}{d} \right) \times C$$

$t$: time relative to the starting of the test run when $r$ is last tested.

$d$: duration of the test run until now.

$C$: a constant, set to 0.8 in our experiments

Benefits:

- Routines are tested often
- Routines are tested throughout the whole testing run
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
• 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

$D(f, ps) - D(f, or)$