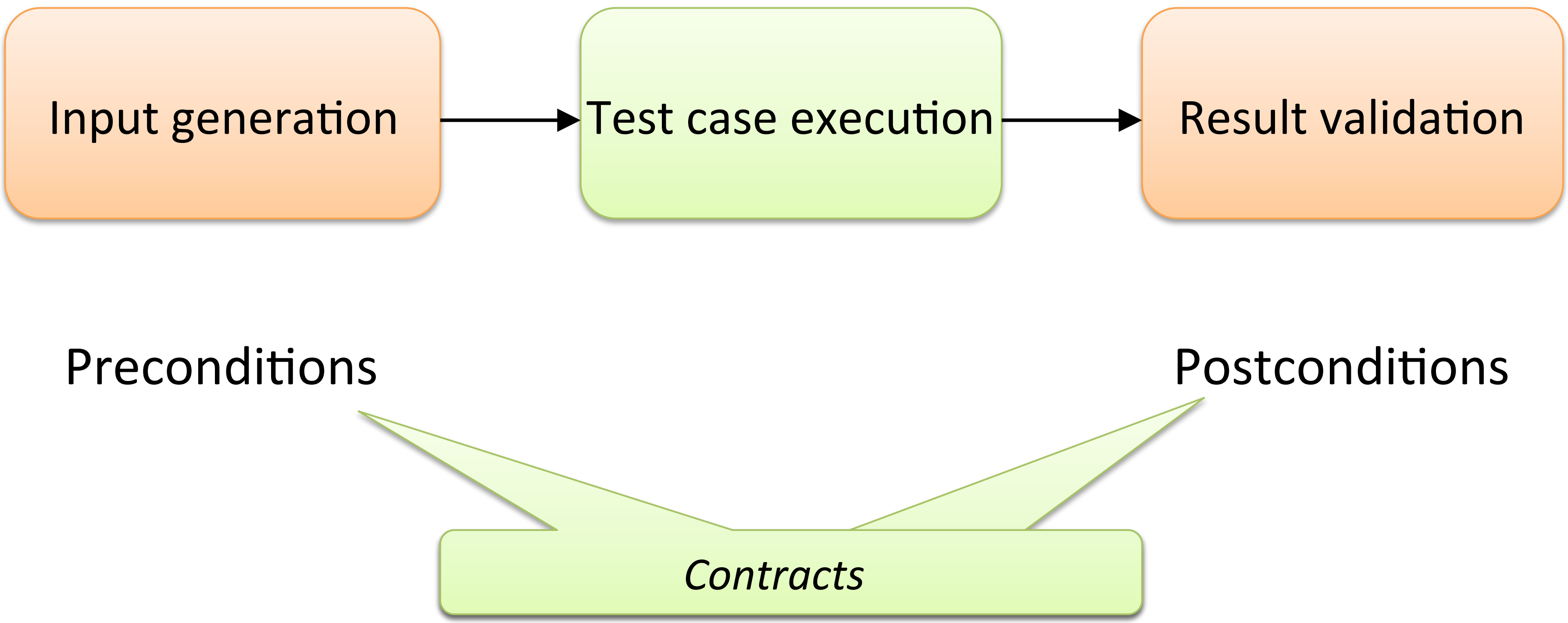
AutoTest: Contract-based testing

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

Chair of Software Engineering

ETH Zurich

Automated unit testing



Design by Contract

```
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
```

Input filter

Oracle

Contract-based random testing

Random input generation:

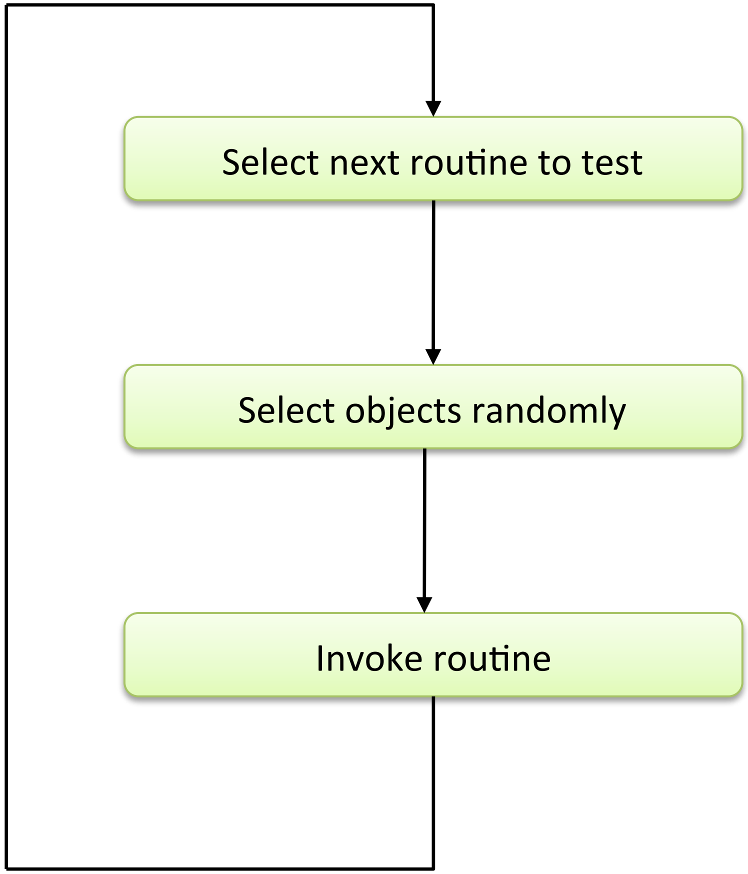
- Primitive values: random selection
- Objects: constructor calls + other (state-changing) methods

Precondition — Input filter

Routine

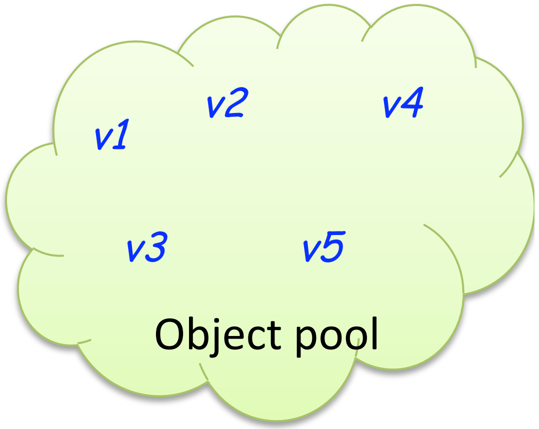
Postcondition — Oracle

Random testing strategy




```
create {LINKED_LIST [INTEGER ] } v1.make
v2 := 1
v1.extend (v2)
v3 := 125
v1.wipe_out
v4 := v1.has (v3)
v5 := v1.count
```

Sample test cases



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:

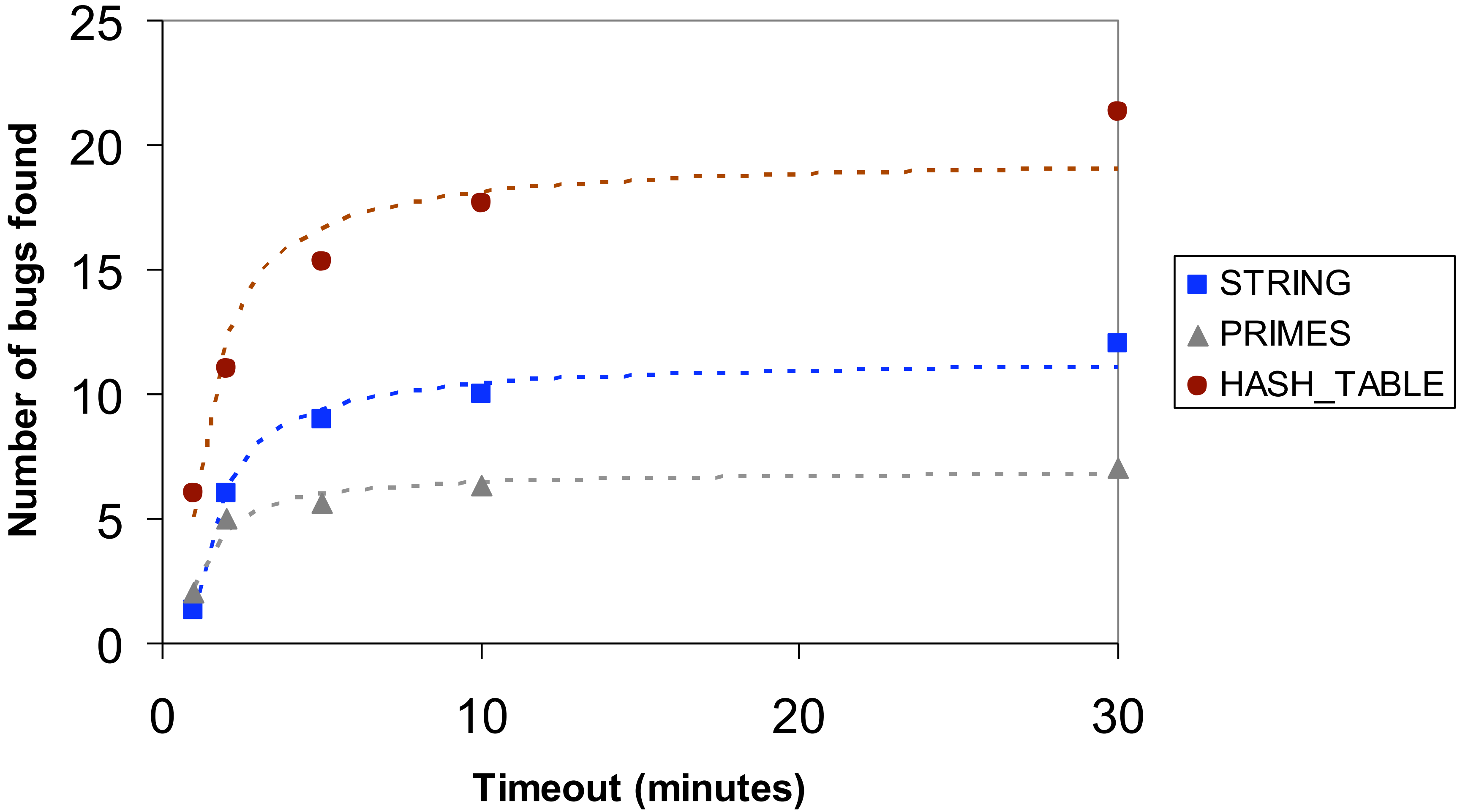
random testing is a poor strategy.

Experimental results:

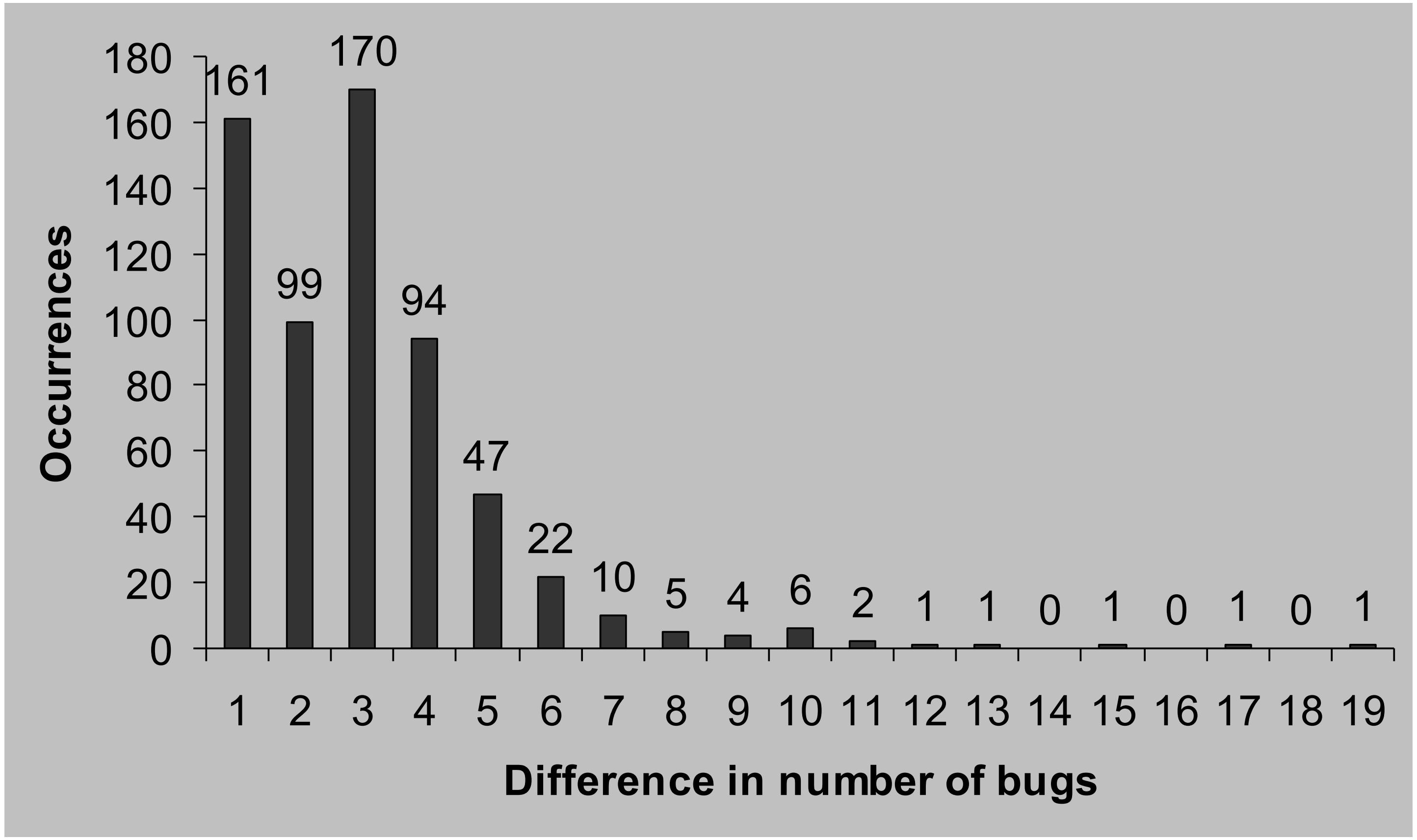
- Random testing is **effective**
- Best: **random$^+$ testing** (random + limit values)
- Relative number of found faults: predictable
- Actual found faults: unpredictable

Number of faults found in time

Inversely proportional to elapsed time: $f(t) = \frac{a}{t} + b$



Number of faults found in classes



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?

```
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 *ps-strategy*

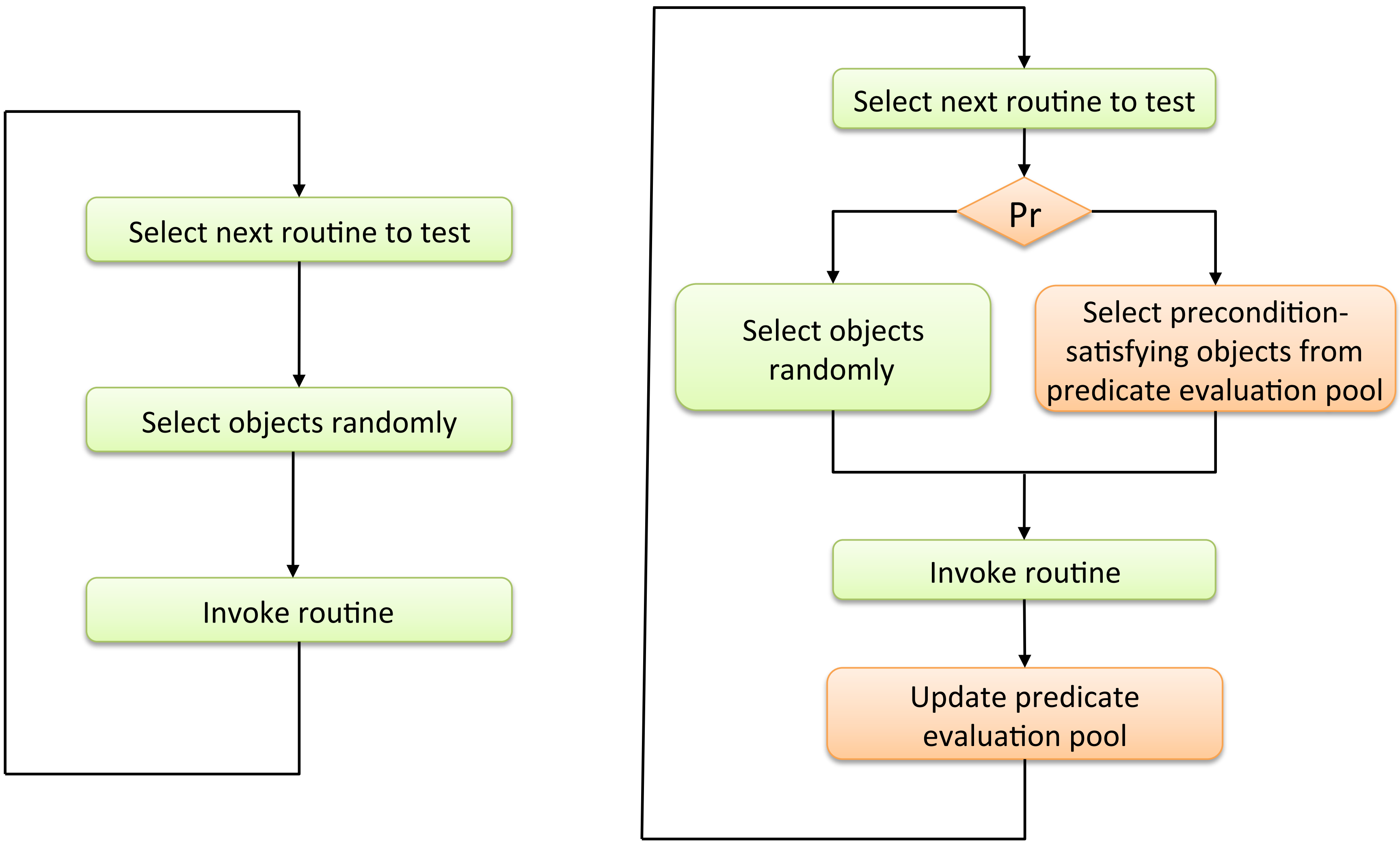
Observation

- 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

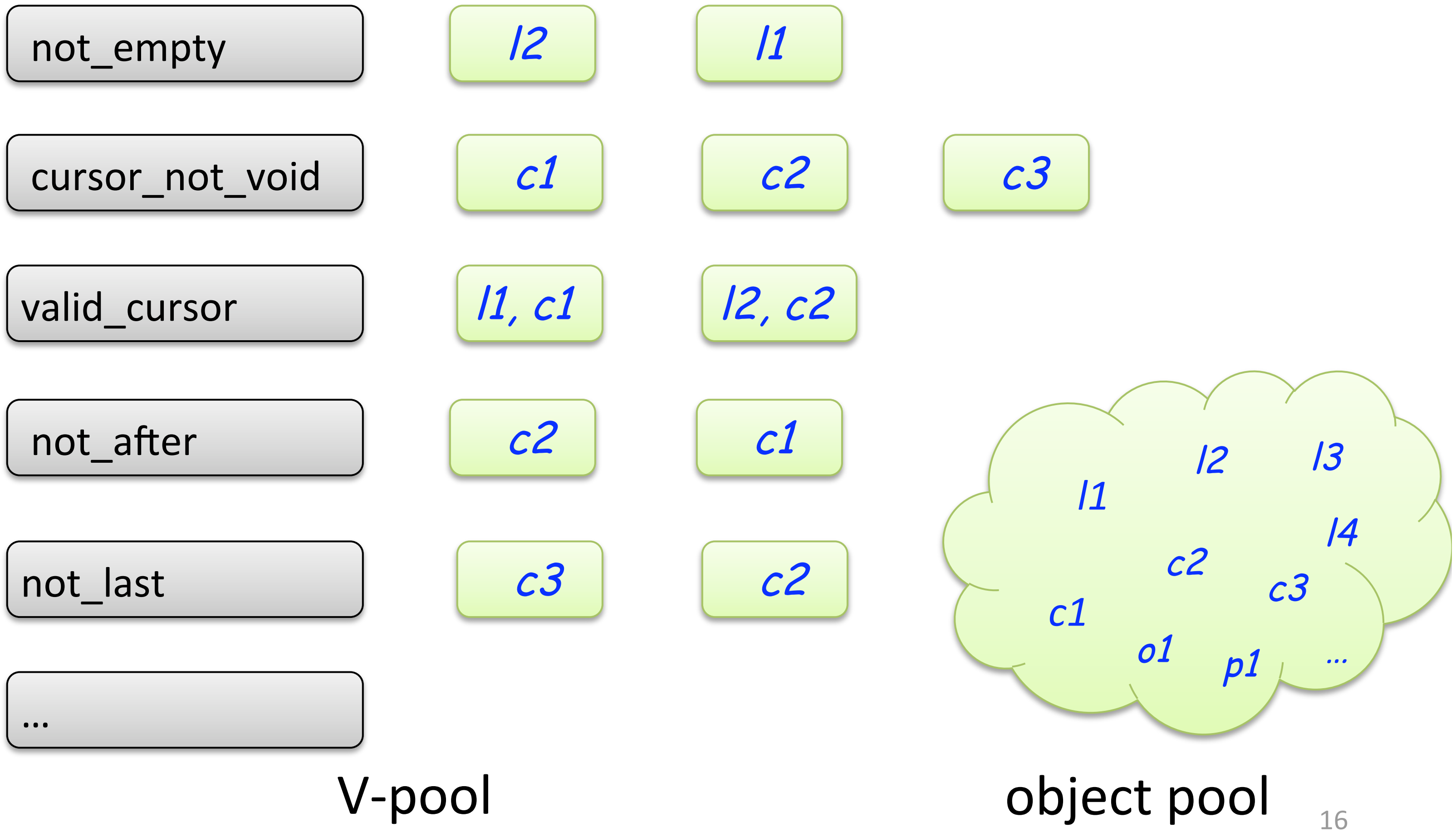


The or-strategy

The ps-strategy

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.



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
```

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

```
l.force_last (v1)
c  := l.new_cursor
c.go_i_th (1)
l.wipe_out
l.replace_at_cursor (v3, c)
```

cursor_not_void	c
valid_cursor	l, c
not_off	c
...	

After every invalid 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.off
```

l.force_last (v1)

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

l.wipe_ou

l.replace_at_cursor (v3, c)

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

cursor_not_void	*c*
valid_cursor	*l, c*
not_off	*c*
...	

For linear constraints

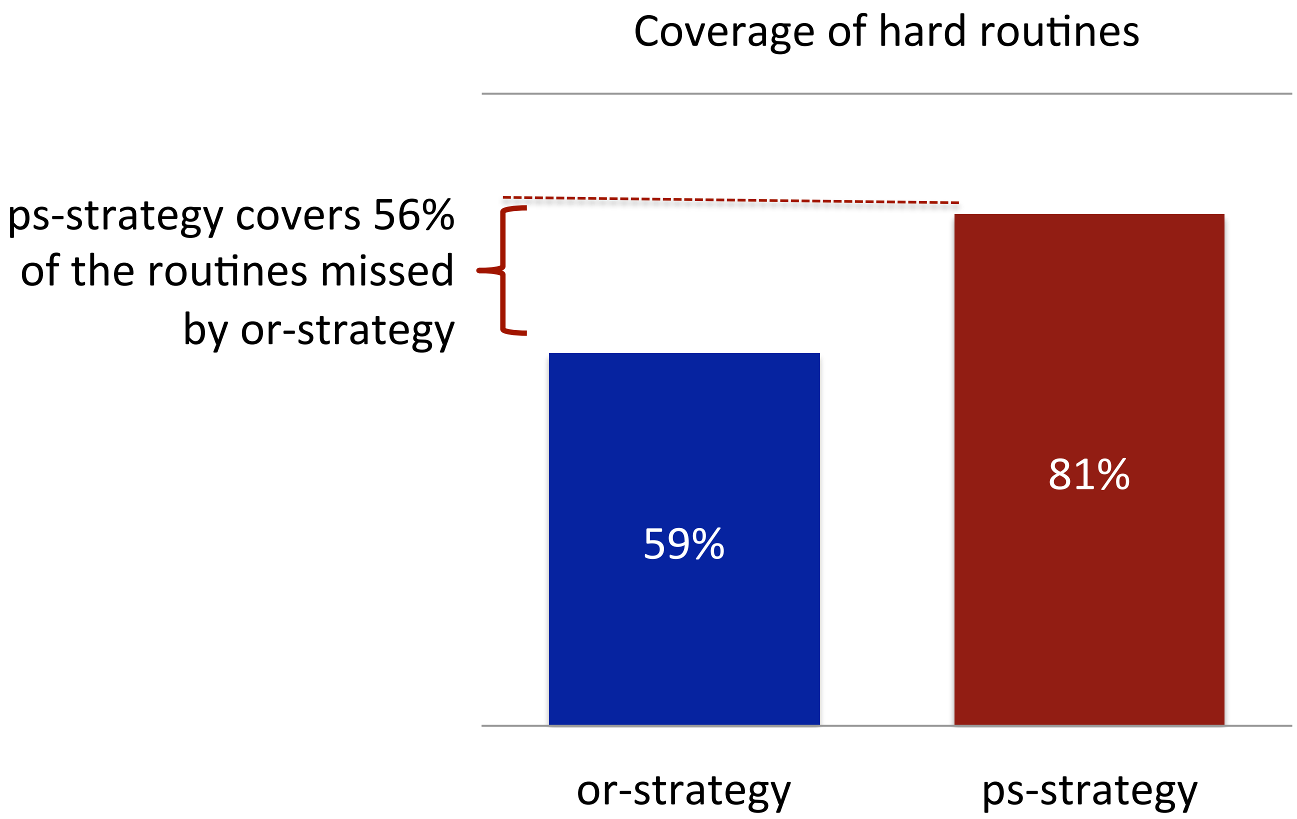
```
prune (n: INTEGER_32; i: INTEGER_32 )
        -- Remove `n' items at and after `i'-th position.
    require
        valid_index: 1 <= i and i <= count
        valid_n: 0 <= n and n <= (count - i + 1)
    ensure
        new_count: count = old count - n
```

lpsolve is used to generate a minimal and a maximal solution

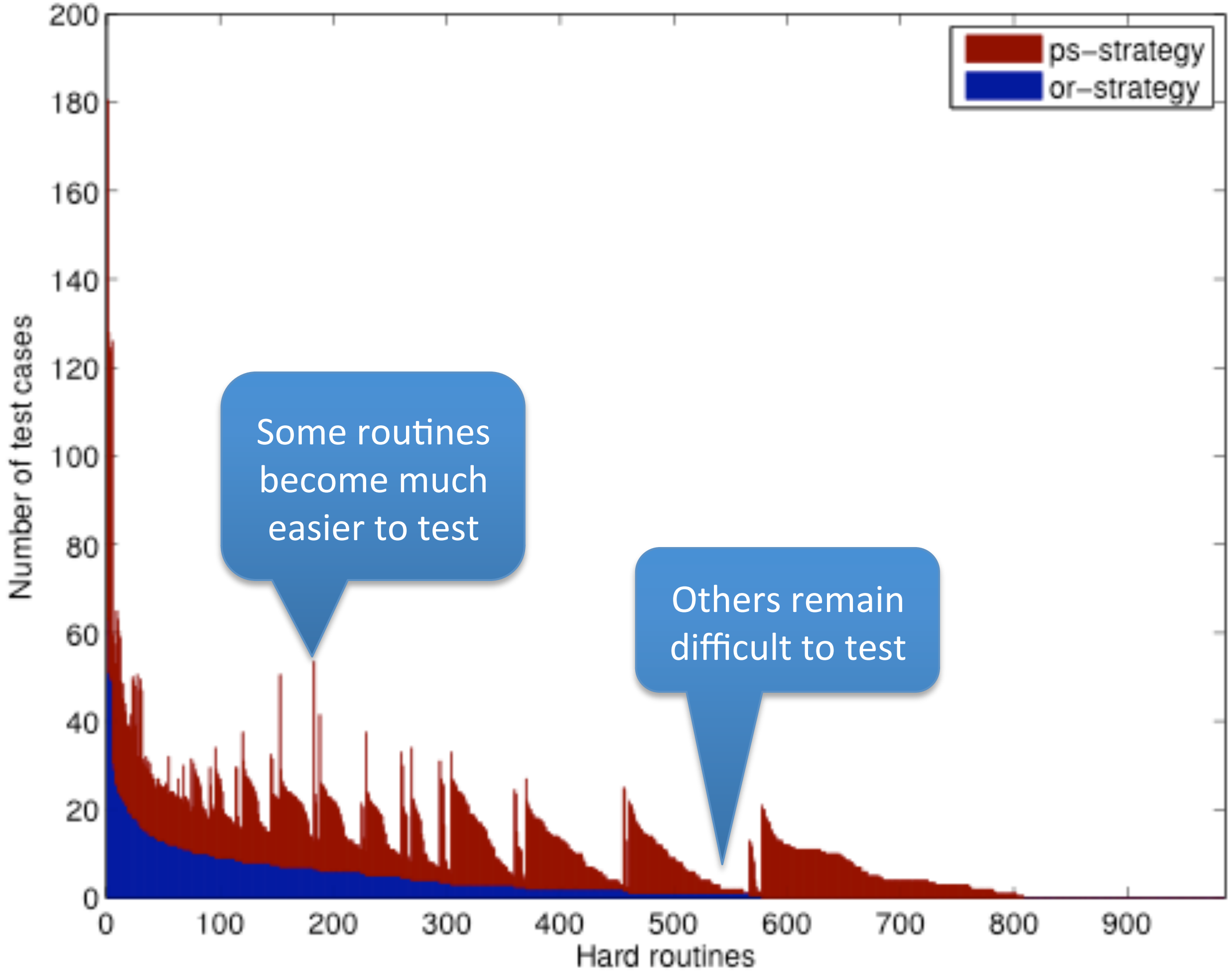
- Randomly select one value from the range
- Slightly biased on border values and potentially interesting values
- Solutions are cached

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.

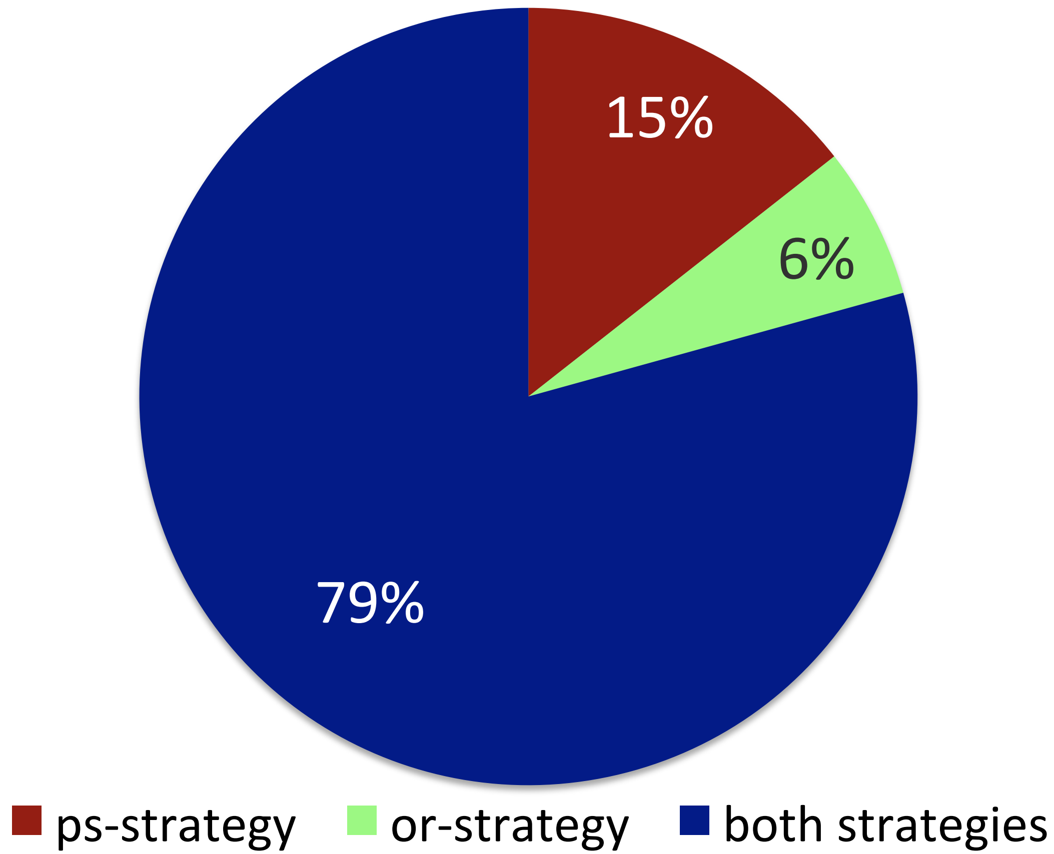


How often are routines tested by the ps-strategy?



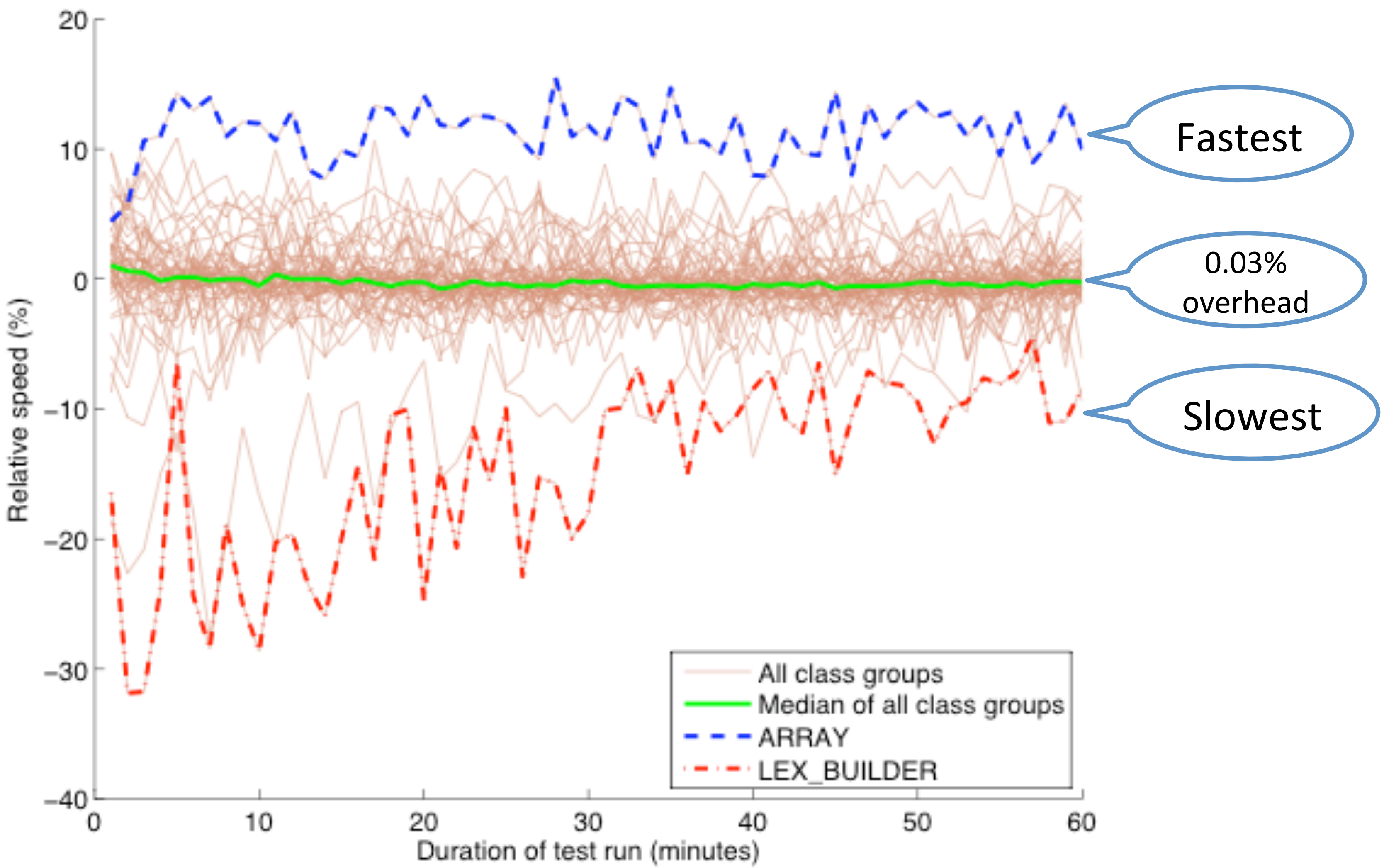
- Over 3.5 times as many valid test cases overall

Fault coverage by each strategy

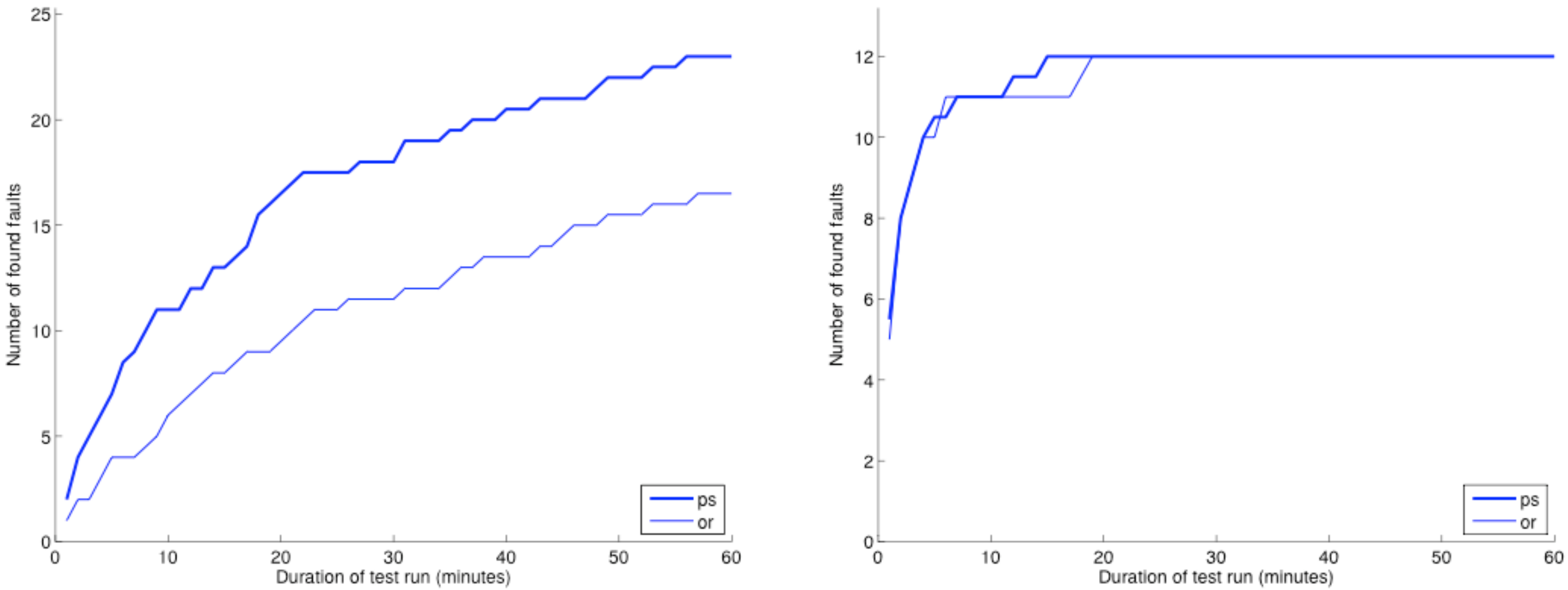


Almost 10% increase in the number of detected faults overall.

Test case generation speed

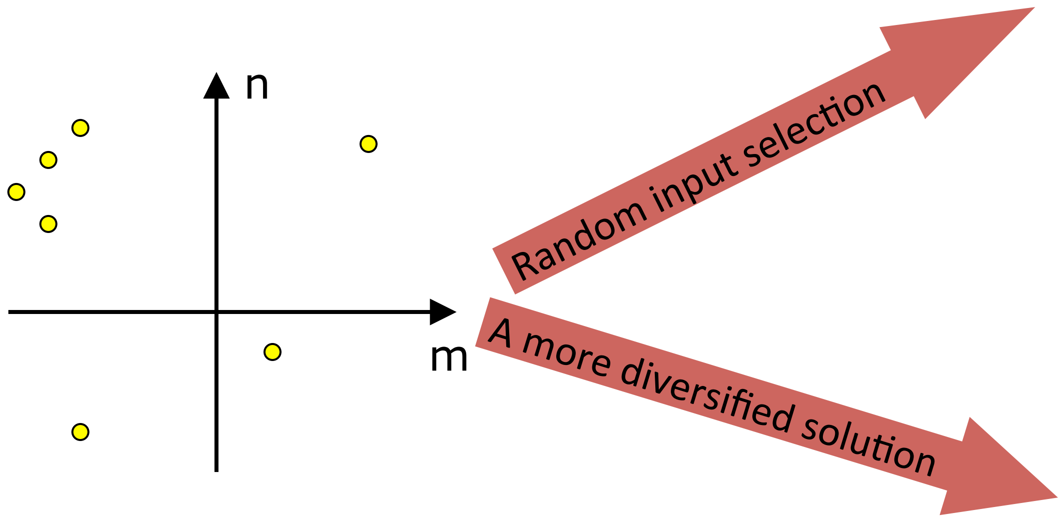


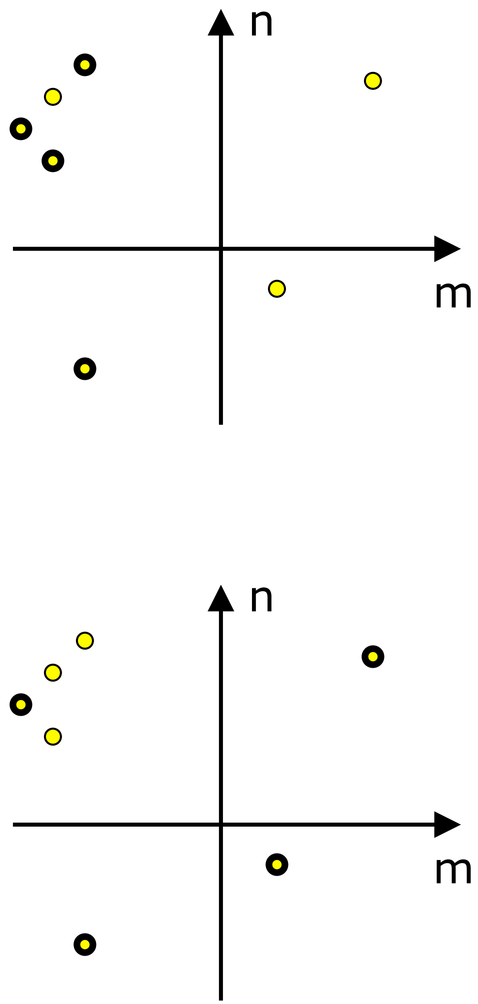
Number of faults detected in time



Randomly generated inputs are not diversified

```
r (m, n: int)
```






```
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

ARTOO

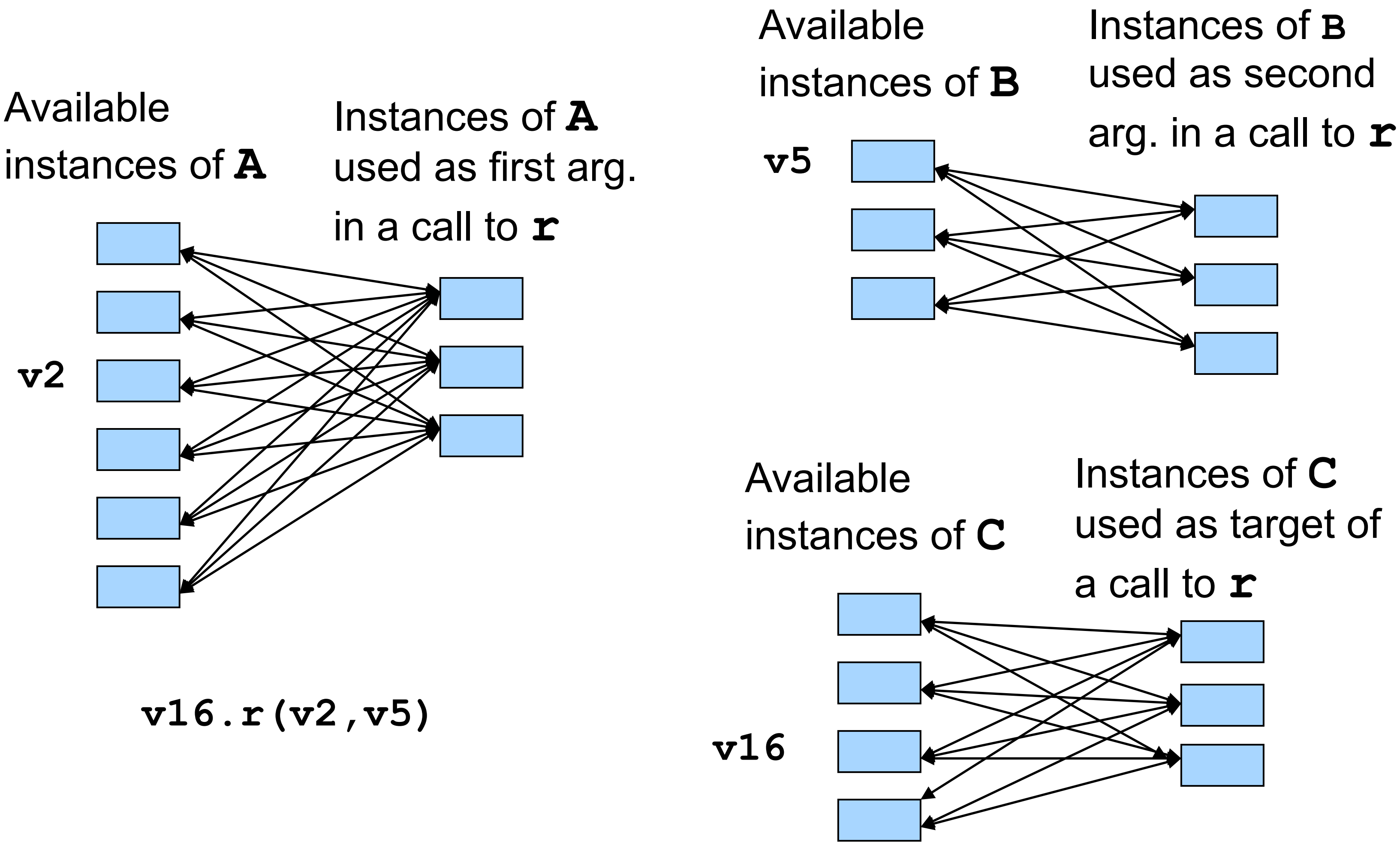
- Adaptive Random Testing for Object-Oriented Software

- Test inputs (objects) generated randomly: candidate set

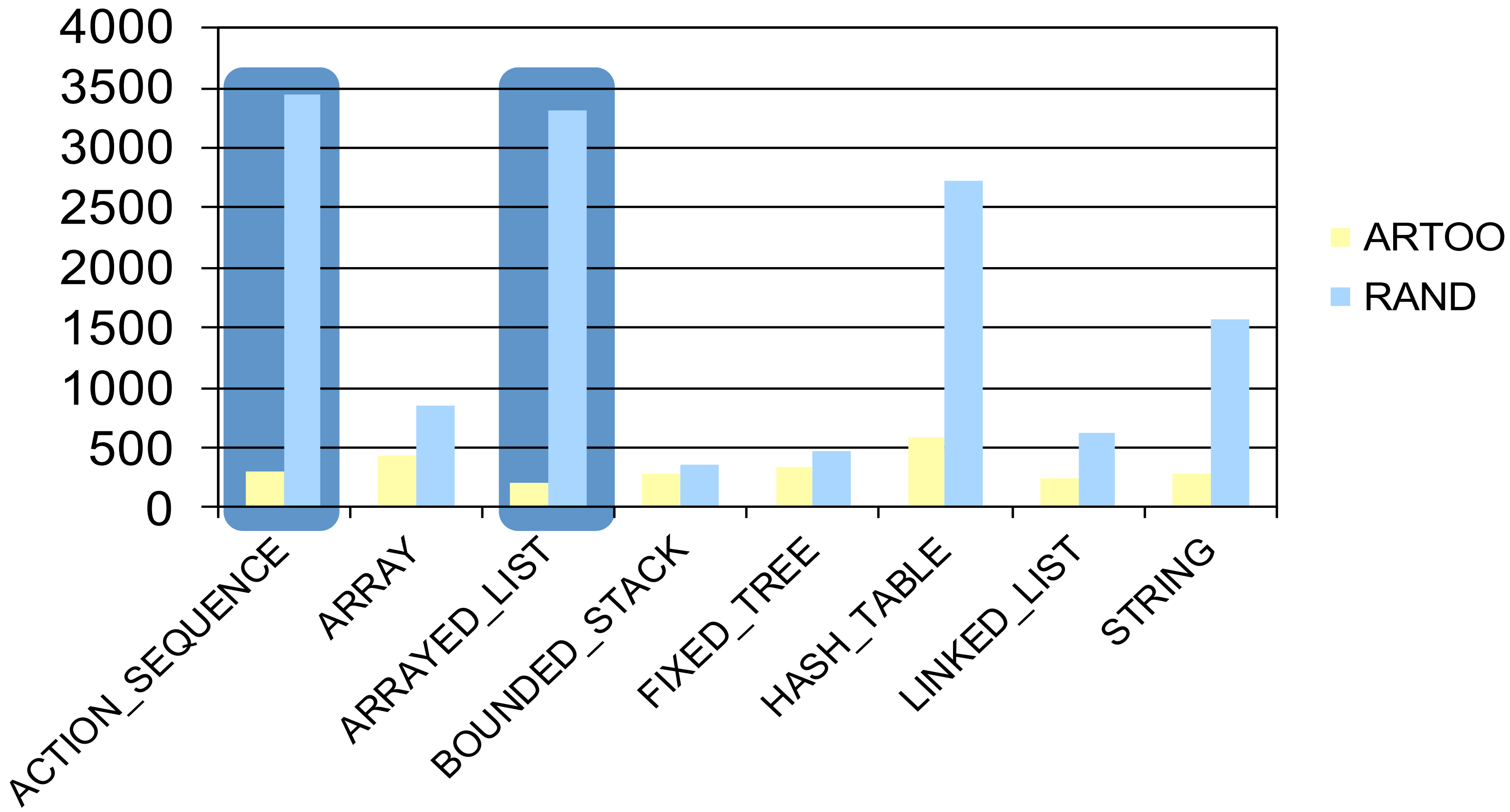
- At every step, the element from the candidate set is chosen which has the maximum average distance to the already used ones

ARTOO algorithm

`r (arg1: A; arg2: B)` in class **C**

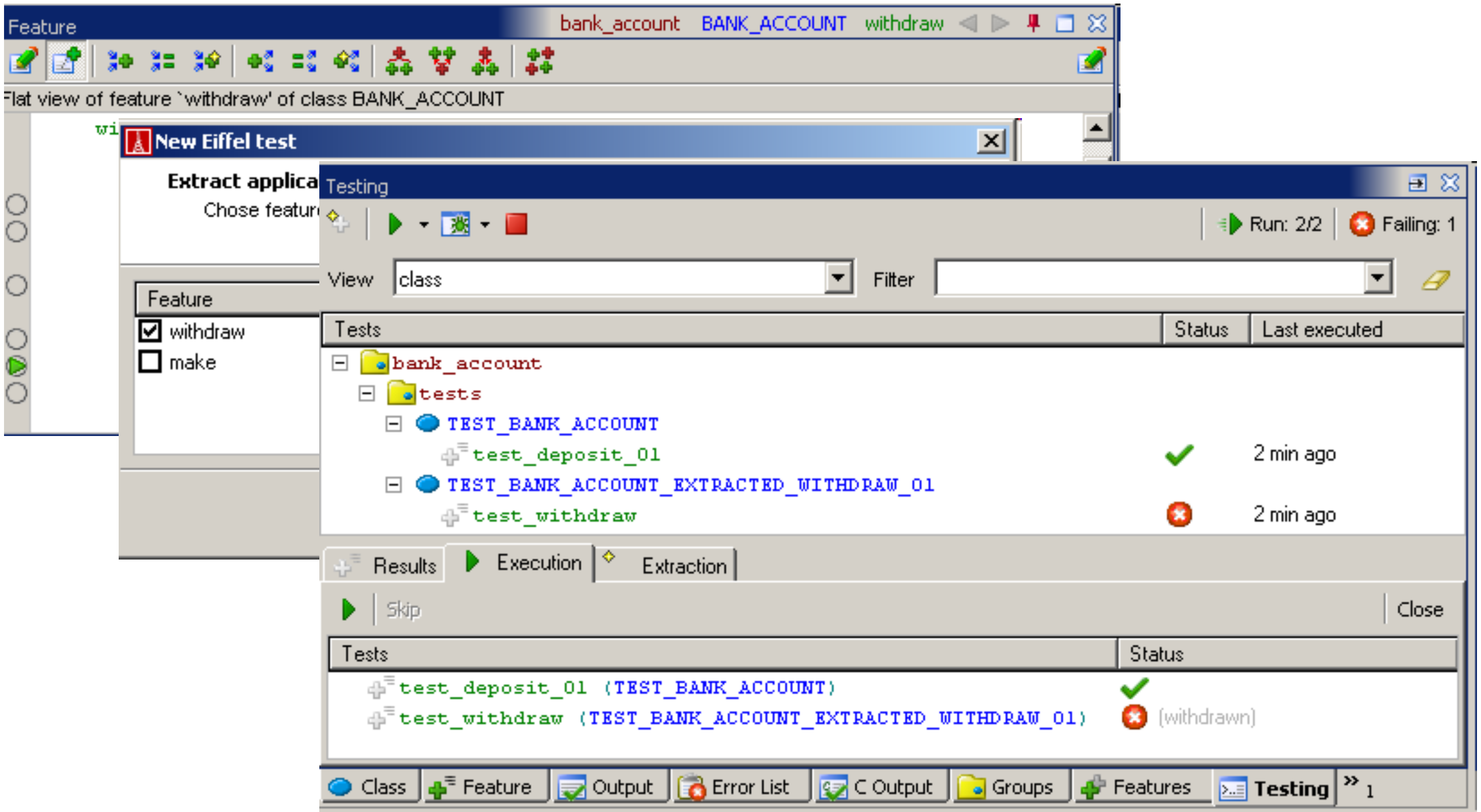


Tests to First Fault



Extracted tests when debugging

Automatically turn failed executions into tests.

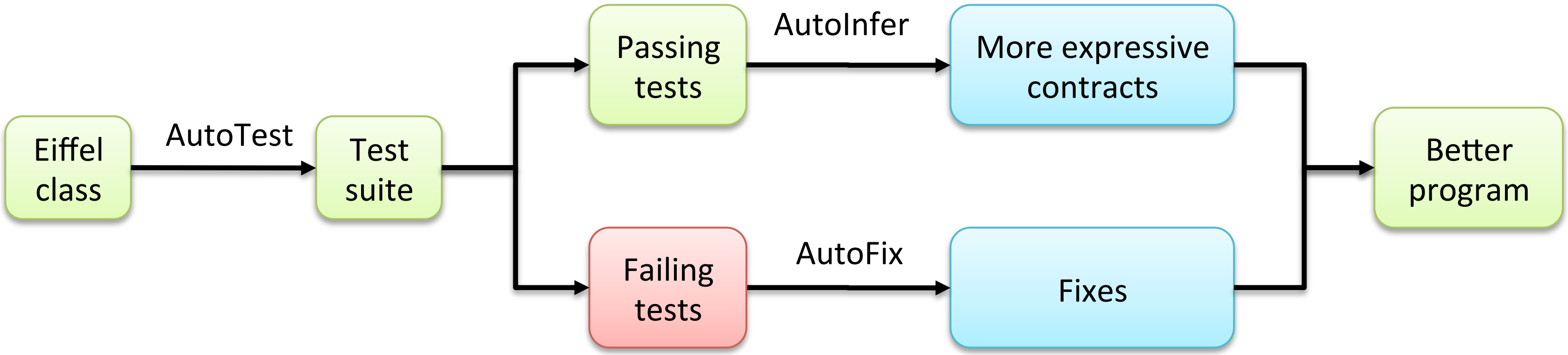


Test Generation + Test Extraction = **AutoTest**

AutoTest Demo

Research oppotunities

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

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 *occurences* (*o*) = old *ocurrences* (*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*

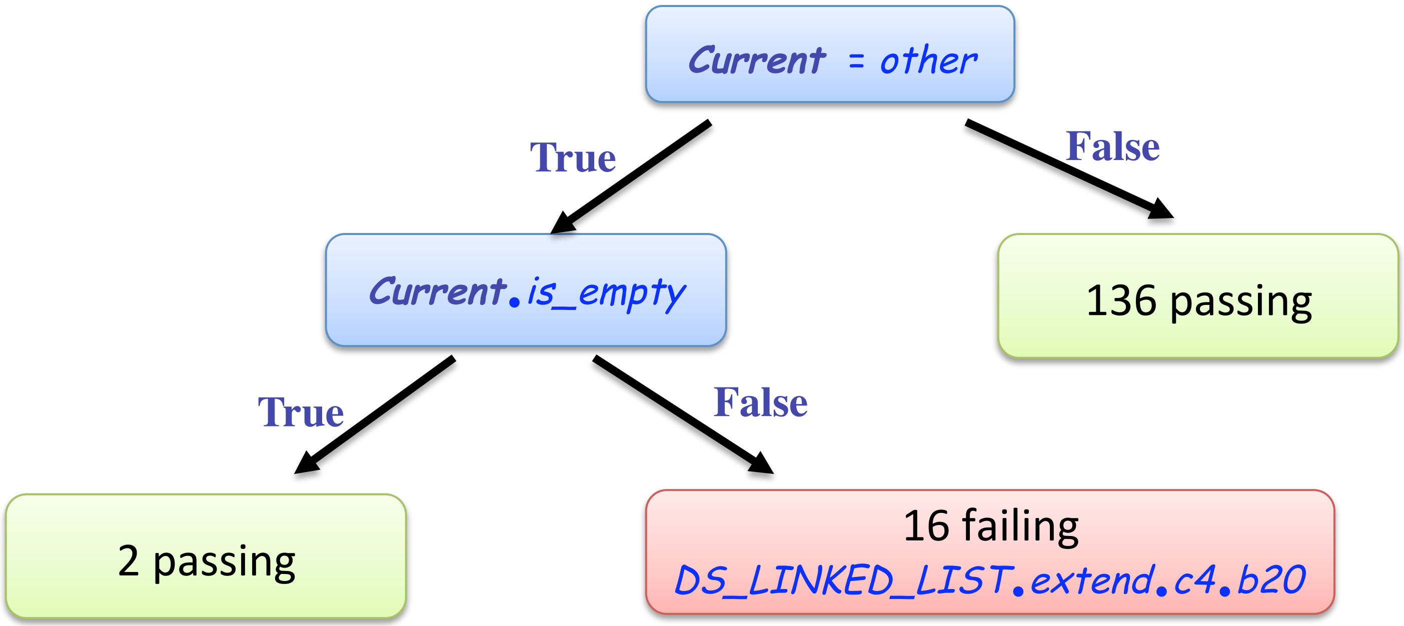
post8: *count* = old *count* + 1

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.



Fixing faults revealed in failing tests

```
duplicate (n: INTEGER ): ...
  do
     pos   := cursor
     Result := new_chain
     Result.forth
     from until  (counter =n) or after  loop
          Result.put_left (item )
          forth
          counter  := counter + 1
     end
     go_to   (pos )
  end
```

Faulty version

```
duplicate (n: INTEGER ): ...
  do
     pos   := cursor
     Result := new_chain
     Result.forth

     from until  (counter =n ) or after  loop
        if before then
           forth
        else
           Result.put_left (item )
           forth
           counter  := counter + 1
        end
     end
     go_to   (pos )
  end
```

Fixed version

Questions?