Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

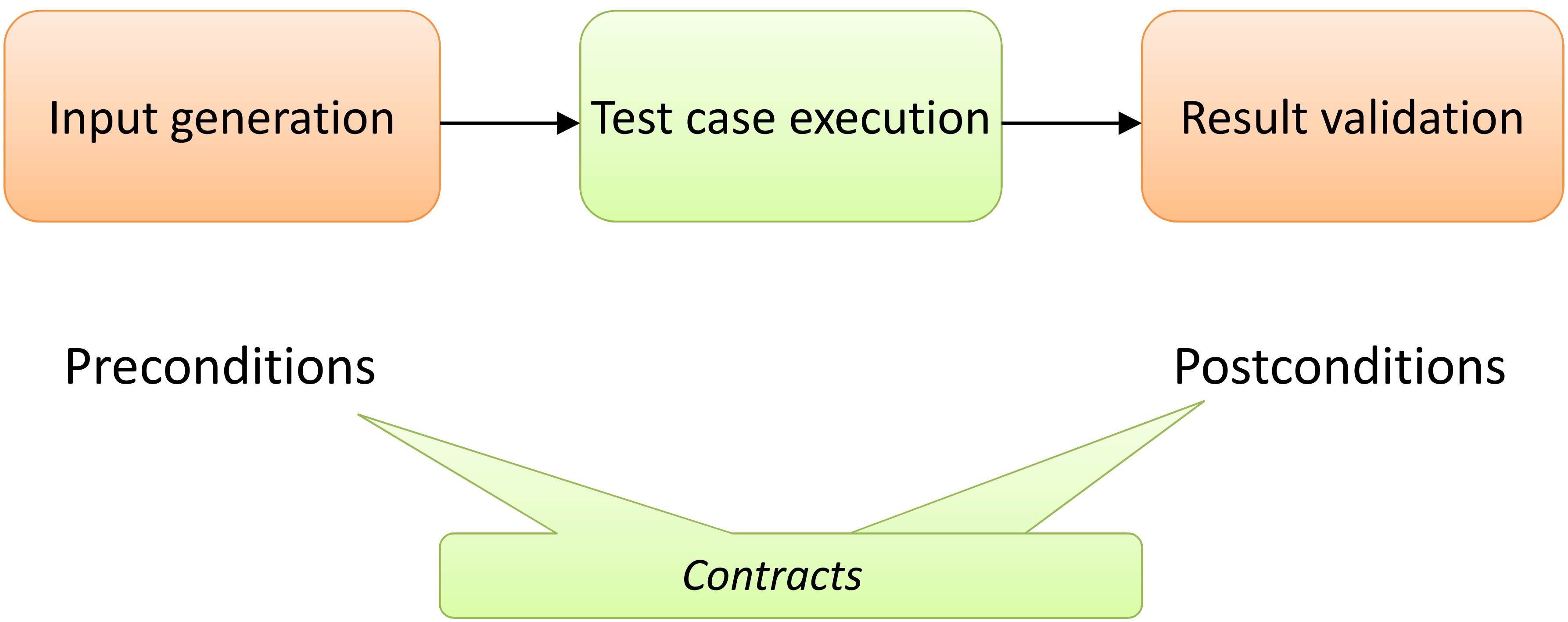
Automated contract-based testing and dynamic contract inference

Yi Wei

Chair of Software Engineering

ETH Zürich

Automated unit testing



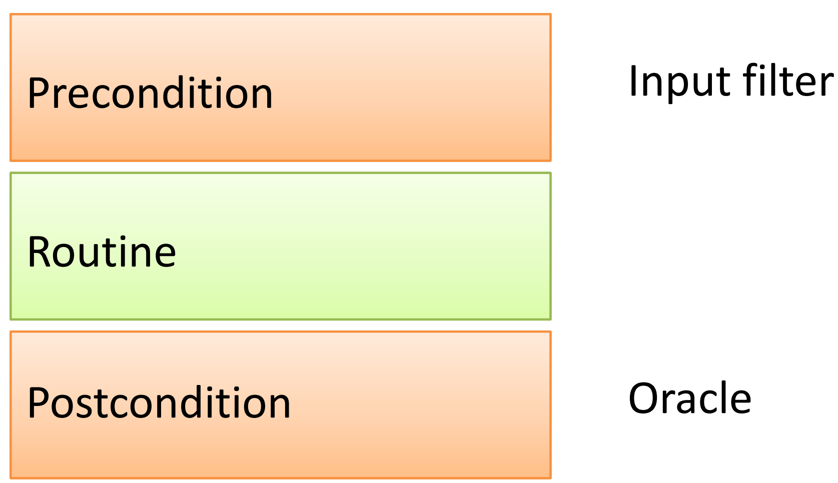
Design by Contract

```
put (v: G; i: INTEGER)
        -- From DS_ARRAYED_LIST
        -- Add `v' at `i'-th position.
require
        extendible: extendible (1)                              Input filter
        valid_index: 1 <= i and i <= (count + 1)

        -- Implementation
ensure
        one_more: count = old count + 1                         Ouput validator
        inserted: item (i) = v
```

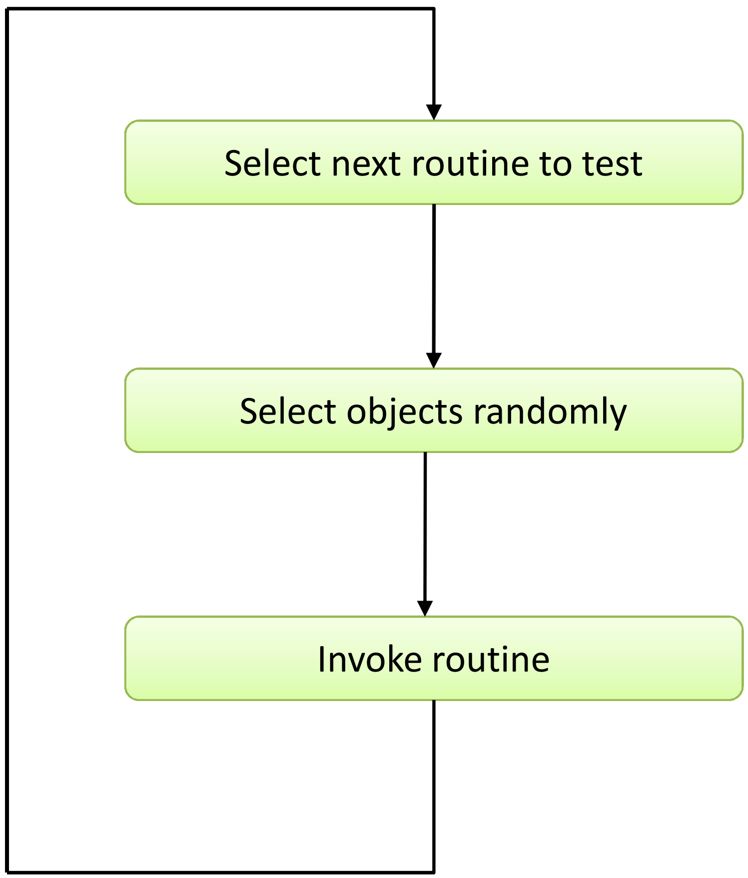
Contract-based random testing

Random input generation:

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

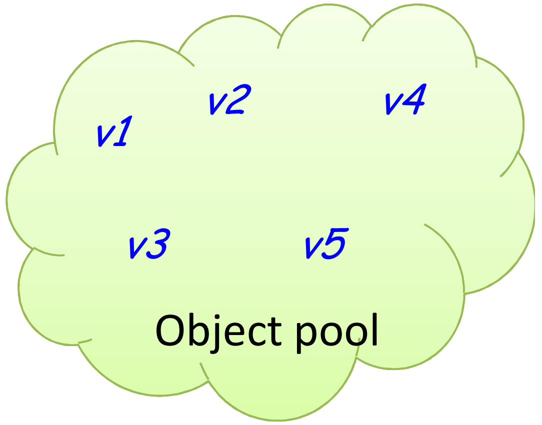


Random testing strategy




```
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:

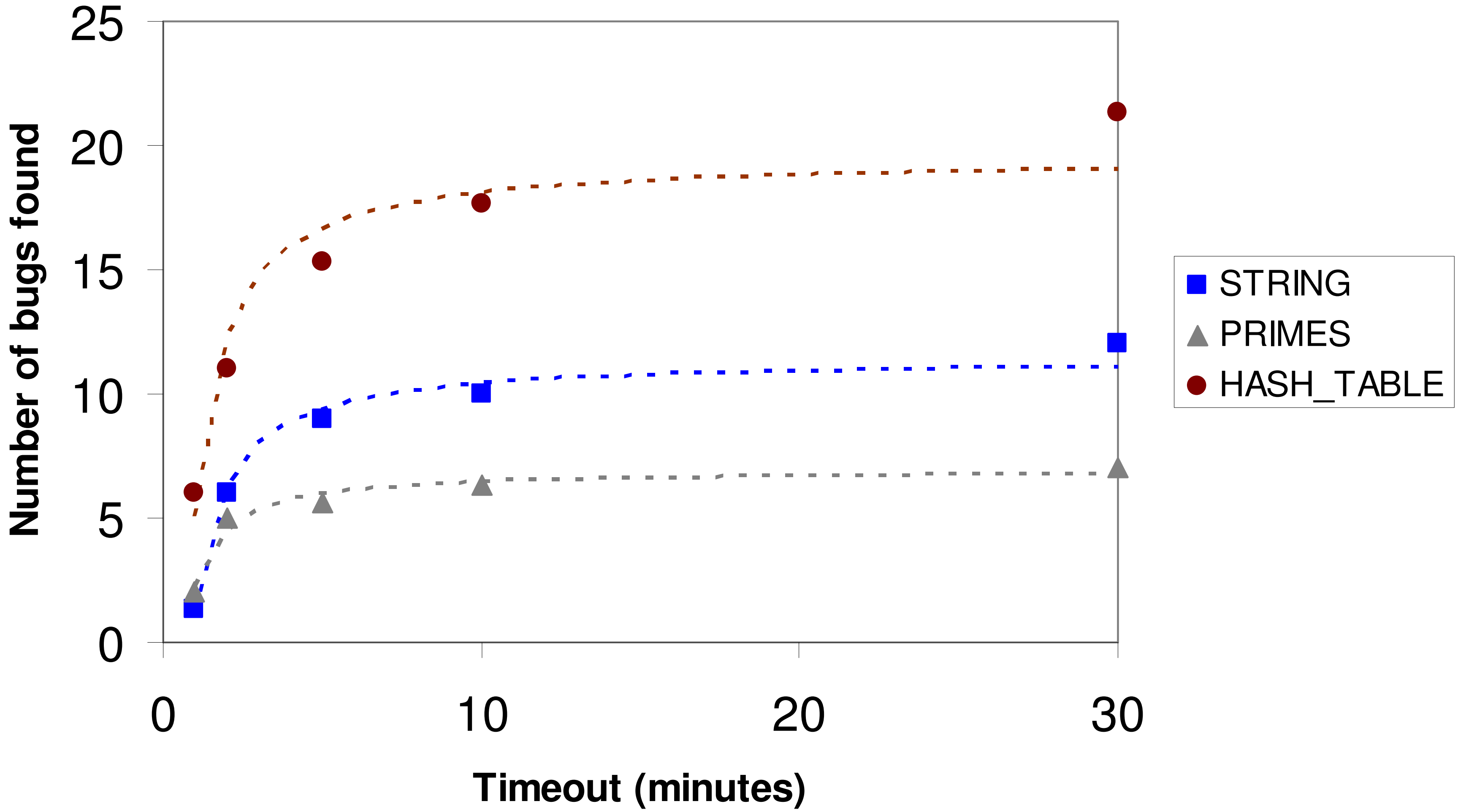
random testing is a poor strategy.

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 detected over time

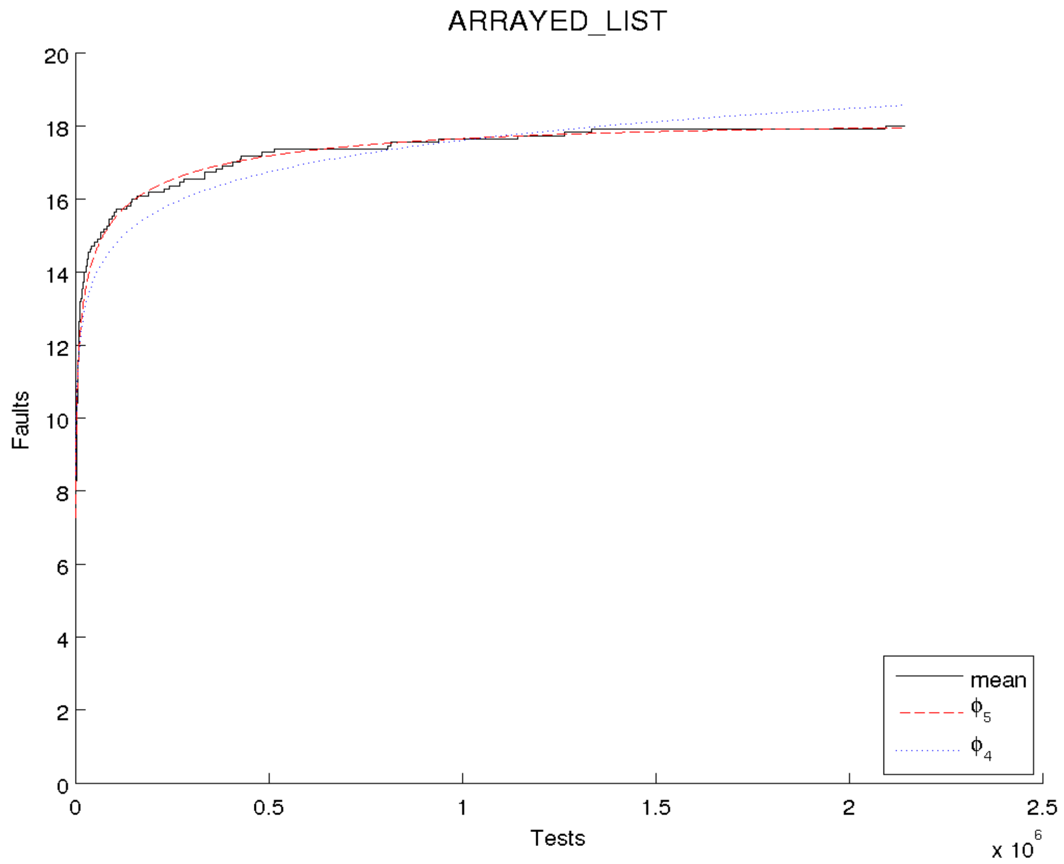
Inversely proportional to elapsed time: $f(t) = \frac{a}{t} + b$



Number of detected faults

$$\Phi(x) = alog^3(x+1) + blog^2(x+1) + clog(x+1) + d$$

where x where is the number of tests



The problem of missing contracts

Contracts are good for defining semantics of programs. But most programs are not equipped with contracts or they only contain very partial contracts.

Dynamic contract inference

Infers contracts (invariants) from program execution traces

Dynamic contract inference

LINKED_LIST.extend (v: ANY) -- Add v to end.

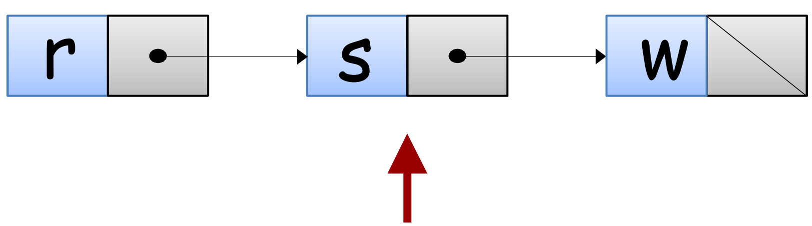
TC1: **old** *count* = 4, *count* = 5	TC1: *not* **old** *has*(*v*), *has*(*v*)
TC2: **old** *count* = 3, *count* = 4	TC2: **old** *has*(*v*), *has*(*v*)
TC3: **old** *count* = 7, *count* = 8	TC3: *not* **old** *has*(*v*), *has*(*v*)
TC4: **old** *count* = 0, *count* = 1	TC4: **old** *has*(*v*), *has*(*v*)
count = **old** *count* + 1	*has* (*v*)

Contract inference for Eiffel? Really?

```
class LINKED_LIST

extend (v: ANY)
        -- Add v to end.
    ensure
        occurrences (v) = old (occurrences (v)) + 1

i_th (i
        -- I
    requir
        i >= 1 and i <= count

index: INTEGER
        -- Index of current position
```

When written contracts are partial, we would like **more**.

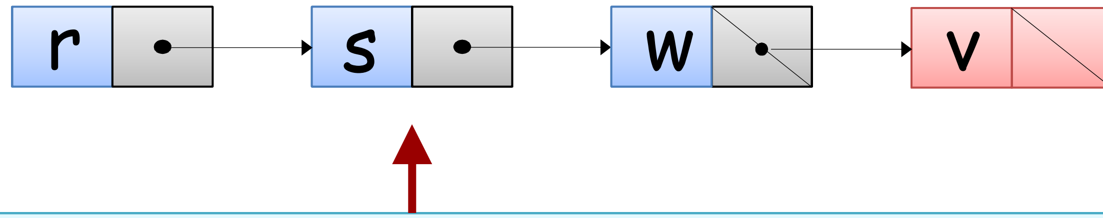
"Ask not what your contract can do for you, ask what you can do with your contract."



AutoInfer: inferring more contracts

extend (*v*: *ANY*)

-- Add v to end.



ensure

The programmer wrote

post1: *occurrences* (*v*) = old (*occurrences* (*v*)) + 1

Our tool 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*

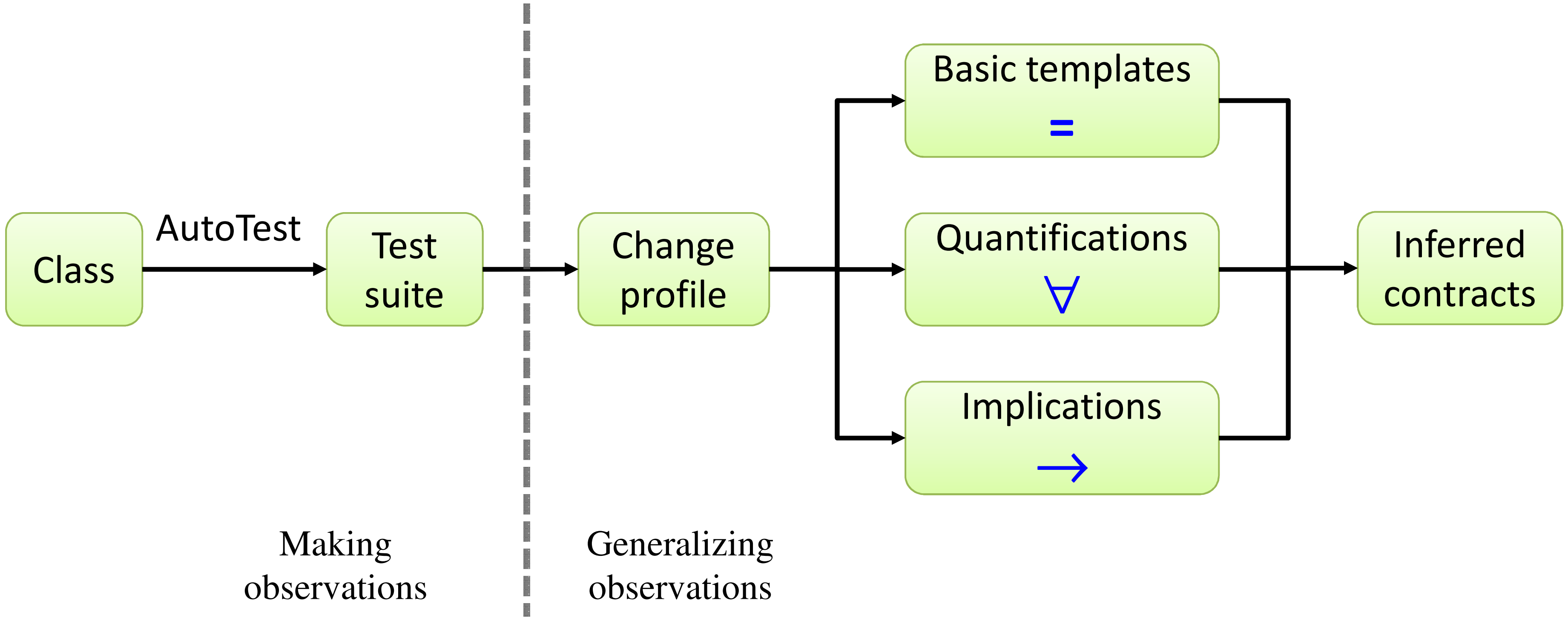
post6: old *after* implies *index* = old *index* + 1

post7: not old *after* implies *index* = old *index*

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

post9: *last* = *v*

AutoInfer: overview



Change profile

Consists of expression evaluations for each test case:

- Expressions constructed from class interface
- Evaluations in both pre and post states

Pre-state of test case 1
list.count = 1
list.has (*v*) = **False**
list.occurrences (*v*) = 0

Pre-state of test case 2
list.count = 7
list.has (*v*) = **True**
list.occurrences (*v*) = 3

list.extend (*v*)

Post-state of test case 1
list.count = 2
list.has (*v*) = **True**
list.occurrences (*v*) = 1

Post-state of test case 2
list.count = 8
list.has (*v*) = **True**
list.occurrences (*v*) = 4

Complementary techniques

extend (*v: ANY*)

ensure

1. Templates based on method signatures

post2: forall *o* . *o* /= *v* implies *occurrences* (*o*) = old *occurrences* (*o*)

post3: forall *o* . *o* /= *v* implies *has* (*o*) = old *has* (*o*)

2. Templates based on sequences

post4: forall *i* . *i* >= 1 and *i* <= old *count* implies *i_th* (*i*) = old *i_th* (*i*)

post5: *i_th* (old *count* + 1) = *v*

3. Decision tree learning

post6: old *after* implies *index* = old *index* + 1

post7: not old *after* implies *index* = old *index*

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

post9: *last* = *v*

Technique 1: signature based templates

extend (*v*: *ANY*)

ensure

1. Templates based on method signatures

post2: **forall** *o* . *o* /= *v* **implies** *occurrences* (*o*) = **old** *occurrences* (*o*)

post3: **forall** *o* . *o* /= *v* **implies** *has* (*o*) = **old** *has* (*o*)

2. Templates based on sequences

post4: **forall** *i* . *i* >= 1 **and** *i* <= **old** *count* **implies** *i_th* (*i*)=**old** *i_th* (*i*)

post5: *i_th* (**old** *count* + 1) = *v*

3. Decision tree learning

post6: **old** *after* **implies** *index* = **old** *index* + 1

post7: *not* **old** *after* **implies** *index* = **old** *index*

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

post9: *last* = *v*

Methods with similar signature

extend (*v: ANY*)

ensure

1. Templates based on method signatures

post2: **forall** *o* . *o* /= *v* **implies** *occurrences* (*o*) = **old** *occurrences* (*o*)

post3: **forall** *o* . *o* /= *v* **implies** *has* (*o*) = **old** *has* (*o*)

Queries available in the same class:

Quantifications based on argument types

occurrences (*v: ANY*): *INTEGER*

-- Number of times v appears.

has (*v: ANY*): *BOOLEAN*

-- Does current list include v?

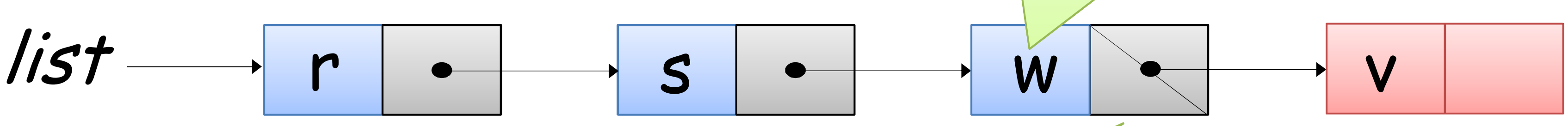
How to evaluate **forall** *o* . *p* (*o*) *efficiently?*

The ones you don’t know, you don’t care

Only consider object *o* in **forall** *o . p(o)* if it is known to the inference tool.

In a test case for: *list.extend(v*

2 to 100 relevant objects in a test case, evaluated in a short time



LINKABLE objects unseen to AutoInfer

Technique 2: templates based on sequences

extend (*v: ANY*)

ensure

1. Templates based on feature signatures

post2: **forall** *o* . *o* /= *v* **implies** *occurrences* (*o*) = **old** *occurrences* (*o*)
post3: **forall** *o* . *o* /= *v* **implies** *has* (*o*) = **old** *has* (*o*)

2. Templates based on sequences

post4: **forall** *i* . *i* >= 1 **and** *i* <= **old** *count* **implies** *i_th* (*i*)=**old** *i_th* (*i*)
post5: *i_th* (**old** *count* + 1) = *v*

3. Decision tree learning

post6: **old** *after* **implies** *index* = **old** *index* + 1
post7: *not* **old** *after* **implies** *index* = **old** *index*

post8: *count* = **old** *count* + 1
post9: *last* = *v*

Templates based on sequences

extend (*v: ANY*)

ensure

2. Templates based on sequences

post4: **forall** *i* . *i* >= 1 **and** *i* <= **old** *count* **implies** *i_th* (*i*)=**old** *i_th* (*i*)

post5: *i_th* (**old** *count* + 1) = *v*

Query in the same class:

i_th (*i* : *INTEGER*): *ANY*

require

i >= 1 **and** *i* <= *count*

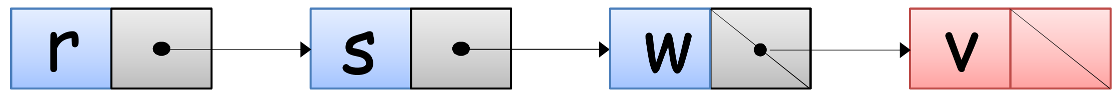
1. Full range of valid integers as indexes

2. Indexes have an order

i_th (i) provides a façade to extract an element sequence

Translating sequence-based contracts

For *extend* (*v*):



seq = (old *seq*) ++ [*v*]

translates into:

post4: forall *i* . *i* >= 1 and *i* <= old *count* implies *i_th* (*i*) = old *i_th* (*i*)

post5: *i_th* (old *count* + 1) = *v*

Three inference techniques

extend (*v: ANY*)

ensure

1. Templates based on feature signatures

post2: **forall** *o* . *o* /= *v* **implies** *occurrences* (*o*) = **old** *occurrences* (*o*)
post3: **forall** *o* . *o* /= *v* **implies** *has* (*o*) = **old** *has* (*o*)

2. Templates based on sequences

post4: **forall** *i* . *i* >= 1 **and** *i* <= **old** *count* **implies** *i_th* (*i*)=**old** *i_th* (*i*)
post5: *i_th* (**old** *count* + 1) = *v*

3. Decision tree learning

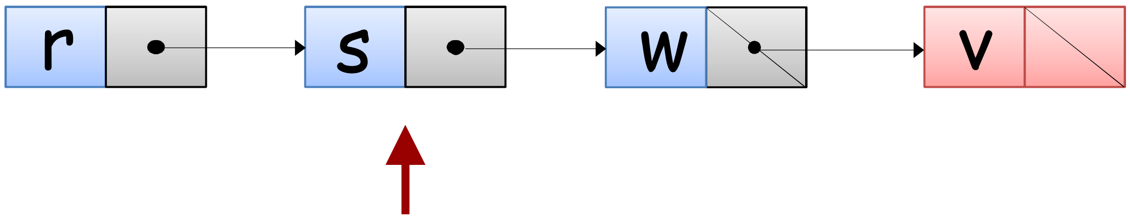
post6: **old** *after* **implies** *index* = **old** *index* + 1
post7: **not old** *after* **implies** *index* = **old** *index*

post8: *count* = **old** *count* + 1
post9: *last* = *v*

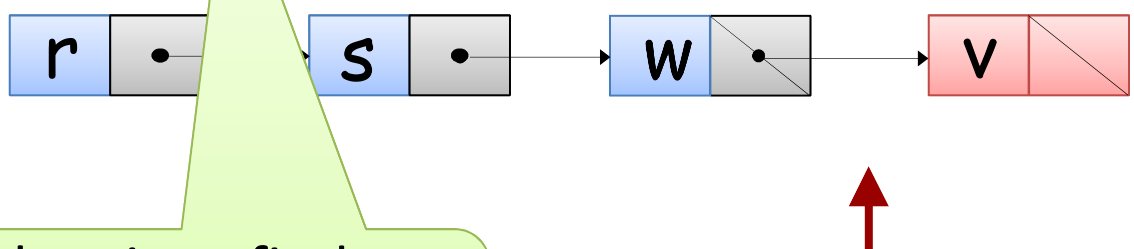
Technique 3: using decision tree to infer implications

extend(*v*):

If cursor is *before* or inside the list, *index* stays.



If cursor is *after* the list, *index* is increased by 1 to make sure the cursor is still *after* the list.



The problem is to find suitable antecedents out of many candidates

Decision trees to infer implications

Decision tree learning

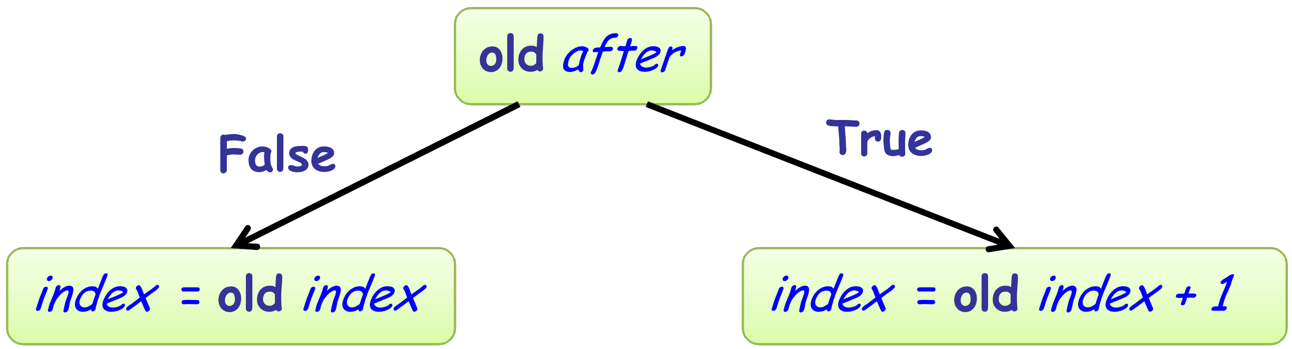
- Works *backward*, from effects to possible causes
- No need to specify antecedents *a priori*

In the *extend* example:

- *index* - **old** *index* evaluates to either 0 or 1
- A decision tree tells in which cases the value is 0 and in which cases the value is 1

Building decision trees

Use expressions in the change profile to build the tree:

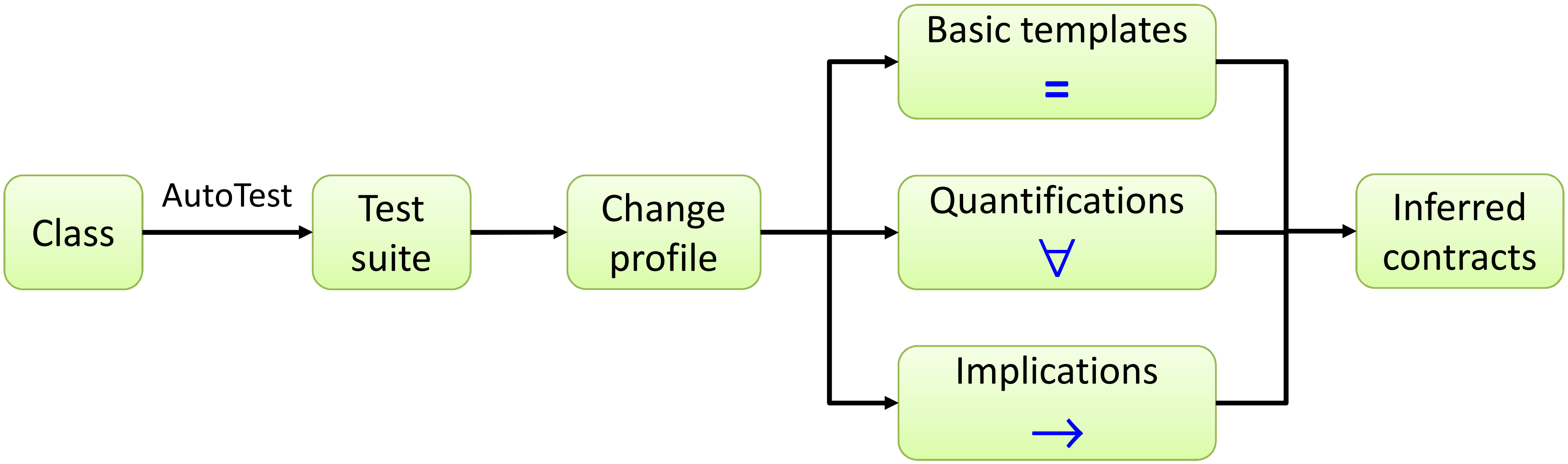


This tree translates into:

post6: (old *after*) implies *index* = old *index* + 1

post7: (not old *after*) implies *index* = old *index*

AutoInfer: results



Results:

94% of inferred postconditions are sound

75% of modifier methods with complete postconditions

But only 50% of inferred preconditions are sound

Problem with dynamic contract inference

Inferred contracts are generalized from program execution traces. Those traces are usually partial. So the inferred contracts may be unsound, especially for preconditions.

```
SET.merge (other: SET)
    require
        inferred: Current.disjoint (other)
```

- Reflects that all the generated tests invoke *merge* with disjointed sets
- Test generation should be improved

Generating tests to violated inferred invariants

```
SET.merge (other: SET)
   require
      p: Current.disjoint (other)
```

- Inferred precondition p summarizes already covered state space
- They also suggests where the test generation should explore – the uncovered part, defined by not p

Potential benefits:

- *A way to detect whether inferred contracts are unsound*
- *A way to force test generation to go into new state region*

Stateful testing: generating invariant-violating tests

For each of the inferred precondition p for a routine, try to generate tests such that p does not hold on entry point of that routine:

Keep track of what objects are generated during testing

```
SET.merge (other: SET)
   require
        p: Current.disjoint (other)
```

- If there exists *s1* and *s2* such that *not s1.disjoint(s2)*, use them directly.
- Otherwise, try to construct objects satisfying *not p*:
  s1.put(v); s2.put(v); s1.merge(s2)

Analyze the behavior of routines that are executed so far

Stateful testing: results

Drives testing to unexplored state space, hence more likely to detect new faults

Results: (for 13 data structure classes)

- Improved the soundness of inferred contracts (pre and postconditions) from 60% to over 99%
- Detected 70% new faults in 7% of the time

Conclusions

- AutoTest: Contract-based random testing
  generates inputs randomly
  uses preconditions as input filter
  uses postconditions as output validator
- AutoInfer: Dynamic contract inference
  generalizes observations from program executions
  is able to infer quantifications and implications
  inferred contracts can be unsound
- Stateful Testing: generate invariant-violating tests
  uses inferred contracts as guidance
  forces testing to go into unexplored state space
  identifies (most of) unsoundly inferred contracts
  detects new faults quickly