Satisfying Test Preconditions through Guided Object Selection

Yi Wei, Serge Gebhardt, Bertrand Meyer
Chair of Software Engineering
ETH Zurich, Switzerland
{*yi.wei,bertrand.meyer*}*@inf.ethz.ch,*
gserge@student.ethz.ch

Manuel Oriol
Dept. of Computer Science, University of York
York, UK
manuel@cs.york.ac.uk

***Abstract*—A random testing strategy can be effective at finding faults, but may leave some routines entirely untested if it never gets to call them on objects satisfying their preconditions. This limitation is particularly frustrating if the object pool *does* contain some precondition-satisfying objects but the strategy, which selects objects at random, does not use them.**

The extension of random testing described in this article addresses the problem. Experimentally, the resulting strategy succeeds in testing 56% of the routines that the pure random strategy missed; it tests hard routines 3.6 times more often; although it misses some of the faults detected by the original strategy, it finds 9.5% more faults overall; and it causes no noticeable overhead.

***Keywords*-random testing; precondition satisfaction; linear constraint solving**

I. Introduction

A random testing strategy randomly selects inputs for the program under test. Random strategies are popular because they are easy to implement, widely applicable and have small overhead in choosing test data. Despite the intuition that random strategies are too naive compared to systematic strategies, studies [1]–[4] show that they are effective in detecting faults.

When applied to Object-Oriented (O-O) programs with contracts, however, a pure random strategy may leave routines with strong preconditions entirely untested. Such routines are important because they often perform critical tasks and failing to test them reduces the quality of the generated test suite.

Many techniques have been proposed to address the issue of generating precondition-satisfying tests. Adaptive random testing [5], [6] produces test data that are evenly spread over the input domain, increasing the chance to select precondition-satisfying inputs. Model-based testing [7] builds up a model for the software embedding the pre- and postconditions of every state transition, and only generates tests conforming to that model. Mock objects [8] encapsulate the constraints required by preconditions and only return values satisfying those constraints. Search-based test case generation using evolutionary algorithms [9] has recently been applied to O-O programs as well.

For random testing, the problem of not being able to select precondition-satisfying objects effectively can be particularly frustrating if the object pool, from which the strategy selects objects for routine calls, *does* contain objects that satisfy certain preconditions, but they simply do not get selected at the right time. To correct this problem, we have developed an extension of random testing, the *guided object selection strategy* for satisfying preconditions (abbreviated as *ps-strategy*). As testing proceeds, the ps-strategy keeps track of precondition-satisfying objects; when a routine is to be tested, the strategy selects those objects with a higher probability.

Our results show that compared to the original random strategy (*or-strategy*), the ps-strategy:

- tests 56% of the routines otherwise missed;
- tests hard routines 3.6 times more often;
- finds 9.5% more faults overall, although it misses some of the faults detected by the original strategy;
- causes no noticeable overhead.

A package[1] is available online containing the source code of the ps-strategy, all the results presented here, and the instructions to reproduce the corresponding experiments.

This article is organized as follows: Section II explains the ps-strategy; Section III describes the experiments and Section IV presents results; Section V discusses the findings; Section VI includes the related work; Section VII draws conclusions.

II. Guided object selection strategy

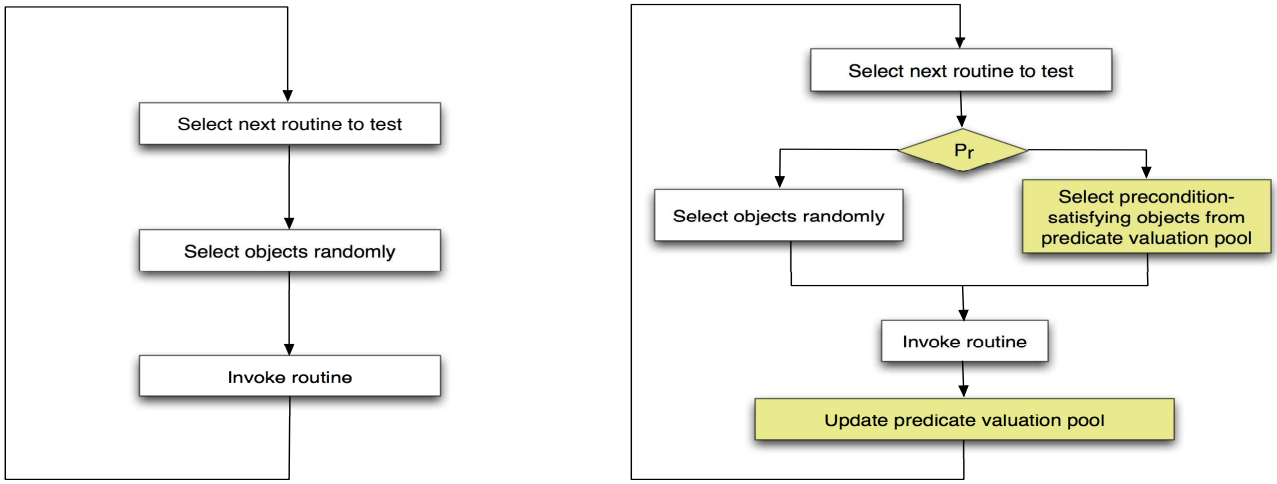
The ps-strategy is an extension of the or-strategy. This section first introduces the AutoTest [10], [11] tool implementing the or-strategy, then explains the guided object selection strategy.

A. The AutoTest Tool

AutoTest is an automatic testing tool implementing the or-strategy for Eiffel. It attempts to test every public routine in a given set of classes within a given time frame. The tool is integrated into the EiffelStudio [12] development environment.

Figure 1(a) shows the workflow of the AutoTest tool with the or-strategy. Within a given time limit, AutoTest

[1] http://se.inf.ethz.ch/people/wei/download/ps_package.zip



(a) The or-strategy

(b) The ps-strategy

Figure 1: AutoTest workflow

repeatedly performs the following three steps to generate the next test case:

1) Select a routine AutoTest stores the number of times it tries to test a routine, and randomly chooses one of the least tried routines. For AutoTest, a test case is always a single routine call.

2) Select objects randomly AutoTest maintains an object pool. All objects created for or returned by routine calls are put into the object pool for future use. When it needs an object as target or argument for the routine under test, AutoTest will either randomly select an object from the pool or create a new one.

3) Invoke the routine AutoTest invokes the routine with the selected objects. If those objects satisfy the routine's precondition, the invocation defines a valid test case for that routine; otherwise the test case is invalid. After the invocation, the whole cycle starts over again.

B. A Motivating Example

The inability of random testing to select precondition-satisfying objects manifests itself in the object selection step. From time to time, there are objects in the object pool satisfying specific preconditions, but they only constitute a very small proportion of all the possible combinations; AutoTest is unlikely to pick them out in pure random selection.

As an example, Listing 1 shows the interface of routine *remove_left_cursor* and *new_cursor* in class DS_ARRAYED_LIST from the Gobo [13] library. Given a cursor object, *remove_left_cursor* removes the list item to the left of the cursor's position, and *new_cursor* returns a newly created cursor object. The precondition consists of 5 assertions, requiring validity of the cursor and existence of a position left to the cursor.

In our experiments, AutoTest failed to satisfy *remove_left_cursor*'s precondition in 30 hours of testing, leaving the routine untested. There were, however, objects in the object pool satisfying the precondition. In a randomly selected test run, at the end of the 50th minute, there are 356 list objects and 192 cursor objects. But only 5 out of the $356 \times 192 = 68,352$ list-cursor combinations satisfied the precondition. The probability that one such combination gets picked by random selection is only 0.007%.

Listing 1: Example of unsatisfied preconditions

```
remove_left_cursor (a_cursor: DS_ARRAYED_LIST_CURSOR)
        -- Remove item to left 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_before: not a_cursor.before
        not_first: not a_cursor.is_first

new_cursor: DS_ARRAYED_LIST_CURSOR
        -- New external cursor for traversal
    ensure
        cursor_not_void: Result /= Void
        valid_cursor: valid_cursor (Result)
```

C. The Guided Object Selection Strategy

One way to increase the likelihood of selecting precondition-satisfying objects is to keep track of the objects which satisfy each precondition. Figure 1(b) shows the workflow of the ps-strategy. Steps that differ from the or-strategy are highlighted:

- A heuristic function P_r decides whether to turn on precondition satisfaction for the selected routine.
- If precondition satisfaction is on, choose precondition-satisfying objects from the predicate valuation pool.
- After test case execution, evaluate which precondition predicates hold for objects that were used in the test case and update the predicate valuation pool.

Precondition-satisfying object selection The ps-strategy maintains, in addition to the object pool, a predicate valuation pool (*V-pool*). The V-pool keeps track of which objects satisfy precondition predicate clauses: for a predicate p with n arguments, the V-pool maintains a set S_p of n-tuples, each representing an object combination that satisfies p.

To map operands[2] for a routine call to object combinations in the V-pool, a function M_p is introduced for every predicate p:

$$M_p : TUPLE_m \rightarrow TUPLE_n$$

where $TUPLE_i$ denotes a set of i-tuples of objects. Given a m-tuple T_m representing the operands to a routine r, and a precondition predicate p with n arguments in r, $M_p(T_m)$ gives a n-tuple containing only the elements needed to evaluate p, in the order as they appear in T_m.

For example, for a list object l and a cursor object c, $M_{not\_first}(\langle l, c\rangle)$ for predicate not_first returns $\langle c\rangle$ because the predicate only mentions the cursor.

To pick objects for a routine r with m operands, the ps-strategy searches the object combination sets associated with r's precondition predicates for candidate objects to construct a m-tuple T_m, such that for each predicate p, $M_p(T_m) \in S_p$. As long as the V-pool is consistent, a tuple constructed this way satisfies the routine's precondition. If no such tuple exists, the ps-strategy resorts to random object selection; if there is more than one way to construct the tuple, the ps-strategy randomly chooses one construction.

Populating the V-pool After the execution of a passing test case, the ps-strategy populates the V-pool by evaluating precondition predicates whose signature conforms to the *relevant* objects and then adding the precondition-satisfying combinations to the V-pool. Relevant objects consist of the operands provided to the last routine call and the returned value, if any. The ps-strategy only uses relevant objects for predicate evaluation because on one hand, those objects are more likely to get changed during the last executed test case hence predicates might evaluate to a different truth value on them; on the other hand, evaluating predicates on all objects entails a huge overhead which decreases the overall effectiveness of the strategy. The ps-strategy tries to populate the V-pool only after passing test cases because a test case ending with an exception may leave relevant objects in inconsistent states.

[2]Operands of a routine call include its target and its arguments, if any.

Using the above example, suppose AutoTest generated a test case containing the following routine call:

```
o7 := o5.new_cursor
```

After executing this test case, the ps-strategy evaluates predicates including $o5.valid\_cursor(o7)$. Because the predicate evaluates to true (can be seen from the postcondition of new_cursor), the ps-strategy stores $\langle o5, o7\rangle$ in the predicate valuation pool for $valid\_cursor$.

Linear constraint solving Preconditions with predicates involing linear constraints occur often; Listing 2 shows a typical example. The or-strategy is ineffective for testing routines with such preconditions.

Listing 2: Linearly-constrained precondition

```
item (i: INTEGER_32): G
      -- Item at index 'i'
      -- From class DS_ARRAYED_LIST
    require
      valid_index: 1 <= i and i <= count
```

To solve a linear constraint, the ps-strategy translates the precondition into a linear programming model and then consults the lp_solve [14] linear programming solver for solutions. For a model, lp_solve can generate a minimal and a maximal solution, consisting of the smallest and largest integer satisfying the constraint, respectively. The ps-strategy uses these two boundary values to define an interval from which a single value is randomly chosen. Although the chosen value is not necessarily a solution of the constraint, our experiments showed that it works in most of the cases. If the chosen value does not satisfy the constraint, the result will be a precondition violation for the routine under test without any further consequence.

The ps-strategy introduces two biases in choosing a value between the boundary values returned by lp_solve[3]:

- If potentially interesting values such as 0, ± 1, ± 2, ± 10, ± 100 are in the interval, then with probability 0.25 one of them will be selected randomly. Previous work [1] showed that AutoTest finds the most faults with this setting.
- With probability 0.125 a boundary value will be selected randomly, because experience in boundary testing [15] showed that boundary values are more likely to reveal faults.

Correcting the V-pool As described earlier, the ps-strategy adds new predicate-satisfying object combinations to the V-pool after every passing test case. As testing proceeds, the objects in the object pool may change because AutoTest reuses existing objects for new routine calls. Consequently

[3]These probabilities are parameters of AutoTest. The values used here were the ones we empirically found to work best for our experiments with no formal claim of optimality.

the information stored in the V-pool may become inconsistent, meaning that the V-pool indicates certain objects satisfy a predicate although this is no longer true.

Keeping track of all the objects affected by the last test case and re-evaluating relevant predicates in the V-pool would dramatically slow down the testing process. Instead, the ps-strategy always lazily assumes that the V-pool is consistent. Only when the test case fails with a precondition violation will the object combination in the V-pool corresponding for that failure be removed. As long as the ps-strategy can suggest precondition-satisfying objects at an acceptable success rate, the algorithm can still be effective.

Optimizations A straightforward implementation of the ps-strategy suffers from a huge overhead. On one hand, searching the V-pool for precondition-satisfying objects takes time; and this process gradually slows down because the pool size grows as testing proceeds. On the other hand, linear constraint solving is much slower than a lookup in the V-pool. Always enforcing precondition satisfaction can entail a $50\% \sim 70\%$ overhead (measured as the number of valid test cases generated in a fixed time period), leaving much less time for actual testing. Although the ps-strategy can test hard routines more often by always enforcing precondition satisfaction, the overall effectiveness of the test process decreases: far fewer faults are found in the same time.

As a tradeoff, the ps-strategy turns on precondition satisfaction only from time to time. It applies the following heuristic function to decide whether to turn it on:

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

where d is the duration in seconds of the test run so far (starting from 1) and t is the time relative to the starting point of the test run when r was tested for the last time. If r has not been tested, t is 0. C is a factor in the range $[0, 1]$. In our experiments, it was set to 0.8. The value of P_r, also within $[0, 1]$, is used as the probability to turn on precondition satisfaction for r. If r has not been tested for a long time, $\frac{t}{d}$ becomes very small because d keeps increasing as testing proceeds while t stays the same, as a result, the value of P_r increases; if r has been tested quite recently, $\frac{t}{d}$ is large, the value of P_r decreases.

The benefits of applying this heuristics are twofold: 1) Our experiments showed that the overhead dramatically decreased, but the ps-strategy could still test precondition-equipped routines quite often. 2) Precondition-equipped routines are tested throughout the whole test run, making it possible to test a routine in a more diversified (in the sense of object states) object pool as testing proceeds.

Another optimization addresses linear constraint solving, the most time consuming part of the ps-strategy. Every model as well as its solution are cached. The ps-strategy consults the cache before sending a model to lp_solve. If there is already a cached solution, it will skip the expensive solving work.

These two optimizations combined decreased the overhead dramatically to a mere 0.03% on average. For some classes, the ps-strategy even generates more tests than the or-strategy in the same time period.

III. Experimental Setup

An experiment of 3420 hours of testing was conducted to evaluate the performance of the ps-strategy. This section describes the tested classes and the experiment setup.

Class selection 92 classes with different semantics and code structures were chosen from the EiffelBase [16] and Gobo libraries. Both libraries are widely used in production software. The classes cover common data structures such as list, stack, queue, table, tree as well as a lexer based on regular expressions, and contain routines with preconditions of various strength.

Table I shows some metrics on these classes. In the table, *Pre-routines* means precondition-equipped routines, *Hard routines* means hard-to-test routines, for which over 90% of the test cases generated by the or-strategy violate the associated preconditions, and *Untested routines* are routines that could not tested by the or-strategy.

Test runs The classes were arranged into 57 groups. Strongly related classes (such as DS_ARRAYED_LIST and DS_ARRAYED_LIST_CURSOR) were put into the same group and tested together. When given a class group, AutoTest will try to test all the routines in them, which may result in a better object state diversification.

Each class group was tested in 30 AutoTest runs with different seeds to the pseudo-random number generator, with each run 1 hour long, under both the ps-strategy and the or-strategy, resulting in 3420 hours of testing in total.

Since seeds provided to the pseudo-random number generator influence the outcome [17], the results presented below are averaged out through the 30 runs of each class group using the median. The median often better expresses the common-run, unlike the mean, which is more affected by the extreme high or low values.

Computing infrastructure The ps-strategy is implemented on top of the AutoTest tool in EiffelStudio 6.4, which serves as the reference or-strategy in the comparison benchmark. The experiment was conducted on 9 PCs with Pentium 4 at 3.2GHz, 1GB of RAM, running Linux Red Hat Enterprise 4. AutoTest was the only CPU intensive program running during testing.

IV. Evaluation

This section presents the results of our experiments. The results compare the performance between the ps-strategy and the or-strategy in the following ways: tested routines and their test frequency; detected faults; the test case generation

Table I: Metrics for tested classes

Type	Classes	LOC	Variations	Pre-routines	Hard routines	Untested routines
Lexer	30	32,108	regular expression, NFA, DFA, lexer	1,290	499	296
List	24	15,482	array, single, double, bidirectional, sorted	913	252	81
Hashed	2	5,156	hash table	66	18	6
Queue	4	7,135	bounded, unbounded, priority	48	2	0
Set	11	15,471	binary tree based, array based, hashed, sorted	299	50	8
Stack	1	1,281	linked list based	15	2	0
String	1	4,815	array based	80	19	8
Tree	19	16,102	binary, n-nary, AVL, red black, search tree	441	144	10
Total	**92**	**97,550**		**3,152**	**986**	**409**

speed, followed by an analysis of the ps-strategy success rate.

A. Increase in the Number of Tested Routines

The primary goal of the ps-strategy is to test more routines. Figure 2 shows the number of test cases generated for every hard routine by both strategies. In the figure, the x-dimension enumerates all the hard routines, and the y-dimension shows the number of test cases for a hard routine under both strategies: every vertical line represents a hard routine. In each line, the height of the dark section represents the number of test cases generated by the or-strategy for that routine; and the height of the light section represents the number of test cases generated by the ps-strategy. To clearly reveal the test case generation trend, the routines are first sorted by the number of test cases generated by the or-strategy, then by the number of test case generated by the ps-strategy.

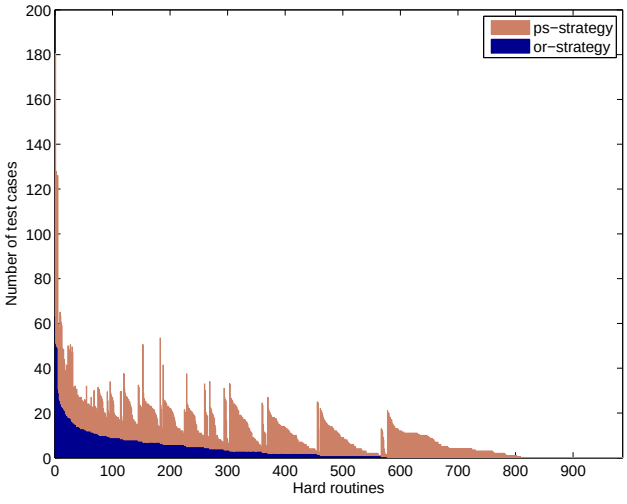


Figure 2: Number of test cases for hard routines

Figure 2 shows that the ps-strategy could test more routines than the or-strategy: extending roughly from 600 to 800 in the x-axis, only light sections appear, indicating only the ps-strategy could generate test cases for those routines. In total, there are 986 hard routines; the or-strategy could test 577 (58.5%) of them, and the ps-strategy could test 802 (81.3%) of them. The ps-strategy could test 231 (56%) out of 409 routines that the or-strategy missed. However, the ps-strategy missed 6 (1%) routines that could be tested by the or-strategy (Figure 2 is too small to show those 6 routines clearly).

The figure also shows that the ps-strategy can generate more test cases (3.6 times more on average) for hard routines, reflected by that the light section is higher than the dark section. Since random testing cannot guarantee complete state coverage, the more test cases for a routine, the higher the chance that the routine is tested in a different state.

In Figure 2, the peaks in the light part reveals that preconditions are much easier to satisfy for some routines than for others, so the ps-strategy can generate more test cases for the former category.

B. Increase in the Number of Detected Faults

The number of detected faults[4] is the most important criterion to evaluate the performance of a testing strategy. Figure 3 shows the histogram of the percentage increase in the number of faults found by the ps-strategy for all the class groups. Compared with the or-strategy, out of the 57 groups, the ps-strategy found more faults in 28 groups, found the same number of faults in 19 groups, and found fewer faults in 10 groups. In 3 groups, the ps-strategy detected over 30% more faults.

C. Kinds of Faults Detected by the ps-strategy and the or-strategy

Previous work [17] of ours showed that random testing can find different faults with different seeds to the pseudo-random number generator. In order to access the overall fault detection ability of these two strategies, we looked at the actual faults that are found by the ps-strategy and the or-strategy in all 30 runs in a class group.

Figure 4 shows the number of group-wise distinct faults that are detected only by the ps-strategy, only by the or-strategy, and by both strategies in each class group. Group-wise distinct faults are the set of faults that are detected in

[4]The faults are real faults in production software.

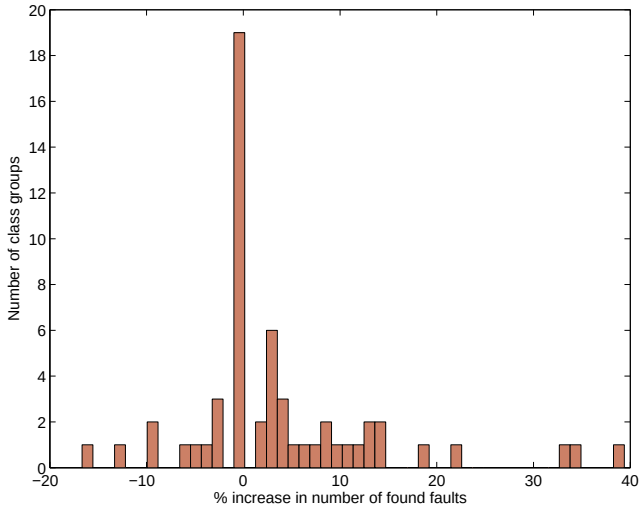


Figure 3: Increase in number of faults

at least one run out of the 30 runs for that group. In the figure, every vertical bar represents a class group. In each bar, the height of each colored section represents the number of group-wise distinct faults that are detected by a particular strategy or by both strategies.

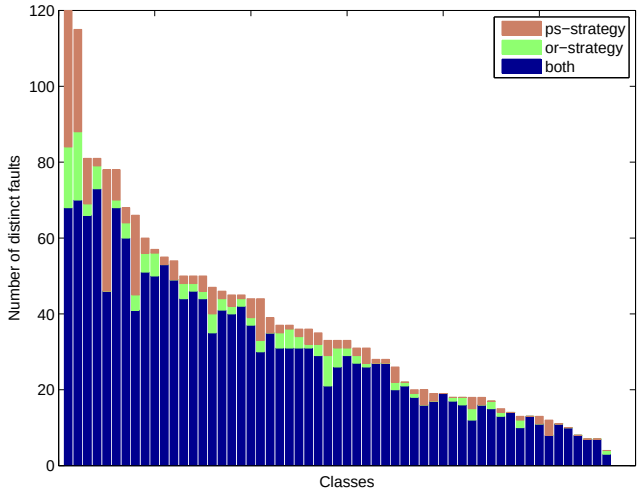


Figure 4: Group-wise distinct faults detected by the ps-strategy and the or-strategy

Figure 4 shows that most of the faults were found by both strategies, but some were found only by the ps-strategy and some only by the or-strategy.

The same faults can be found in multiple class groups, for example, a fault in class ARRAY is likely to be caught in many groups because they all use ARRAY. After removing all duplications, the strategy-wise distinct faults are defined as the set of distinct faults that are detected in at least one run in any group under a certain strategy, or a set of strategies.

There were 1124 strategy-wise distinct faults detected by the two strategies, out of which the or-strategy found 962 and the ps-strategy found 1053, yielding a 9.5% increase. 891 (79.3%) faults were found by both strategies, 162 (14.4%) were found only by the ps-strategy and 71 (6.3%) only by the or-strategy.

D. Fault Detection Probability

Due to its nature, a random strategy may detect different faults in different test runs. Fault detection probability measures how likely a random strategy detects a fault in a test run. It has an important practical implication: the higher the probability, the less runs are needed to detect a fault. Ideally, a strategy can detect all the faults in any single run, then only one run is sufficient, which stands in contrast to the common application of random testing today — test the same program repeatedly with different seeds.

Our experiments contain $R = 30$ runs per class group, the detection probability for a group-wise distinct fault f under strategy s can be measured by:

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

where $N(f, s)$ is the number of runs in which f is detected in that group under s.

Figure 5 shows the histogram for group-wise distinct fault detection probability distribution for both strategies, revealing that the two strategies are quite similar: around 35% of the faults were detected with probability 1, and 22% of the faults were detected with a probability below 0.1. If these two distributions are treated as two stochastic variables, the Pearson correlation coefficient between them is 0.99. This indicates that in the sense of fault detection probability, the two strategies perform almost identically.

Since some faults were found only by a particular strategy, it is interesting to know whether those faults can be detected with a high probability by that strategy or just by luck (with a low probability) due to the random nature. If the probability is high, the strategy must have some characteristics for finding such faults.

Figure 6 plots the group-wise distinct fault detection probability difference between the two strategies. In the figure, every point in the x-dimension represents a fault f, its corresponding y coordinate is the detection probability difference between the ps-strategy and the or-strategy, which is calculated as $D(f, ps) - D(f, or)$. 39% of the faults are detected in both strategies equally often (with 0 difference); 37% of the faults are more likely to be detected in the ps-strategy (with positive detection probability difference); and 24% were more likely to be detected in the or-strategy (with negative difference). This means both strategies have the tendency to detect some kinds of faults more often.

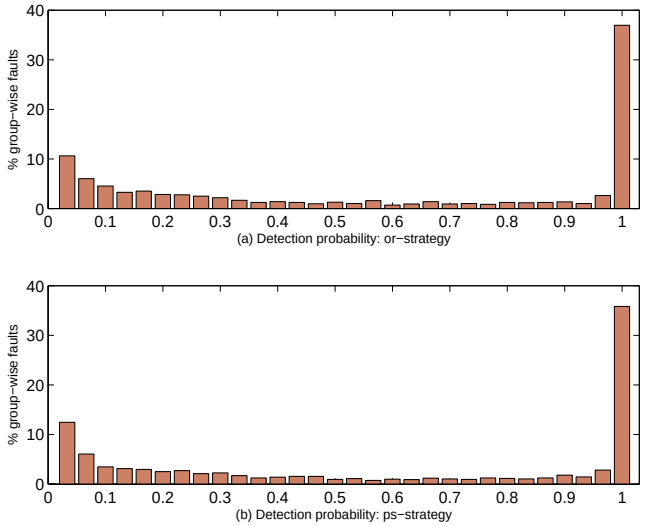


Figure 5: Fault detection probability

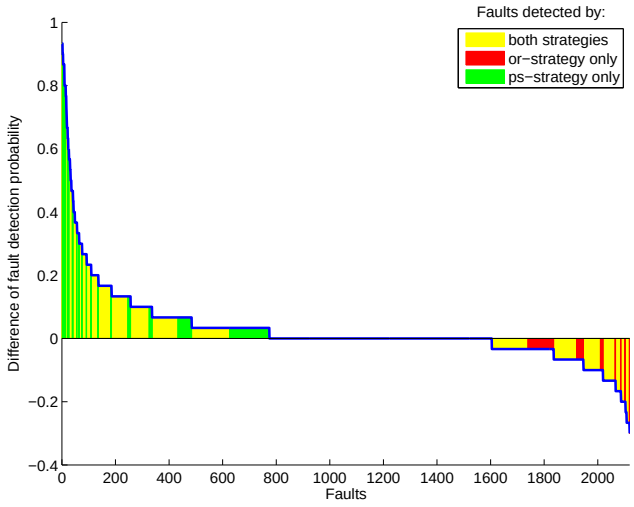


Figure 6: Comparison of fault detection probability

Faults found only by the ps-strategy or only by the or-strategy are highlighted in different colors in Figure 6. They appear above or below the x-axis respectively. Some of them are detected with relatively high probability (The ones to the very left and to the very right of the figure.) The fact that some strategy-specific faults were detected with high probability suggests that they are detected thanks to the characteristic of that strategy instead of pure luck.

E. Test Case Generation Speed

The test case generation speed is measured as the number of valid test cases generated per minute. The overhead of the ps-strategy compared to the or-strategy is defined as the speed difference between these two strategies. Figure 7 shows the speed of the ps-strategy relative to the or-strategy over time. Every curve represents a class group. A line above the x-axis means the ps-strategy is faster, below the 0 line means the ps-strategy is slower. For most of the groups, the curve is close to the 0 line with little variance throughout the testing period. The thick curve around the 0 line is the median of all groups. It stays close to the 0 line, meaning that the ps-strategy brings almost no overhead (only 0.03%). This does not mean that the extra steps involved in the ps-strategy, such as V-pool building and searching, do not take up time; it means that even though they need time, the overall speed is compensated by the fact that more test cases are generated for hard routines while they would otherwise result in precondition violations.

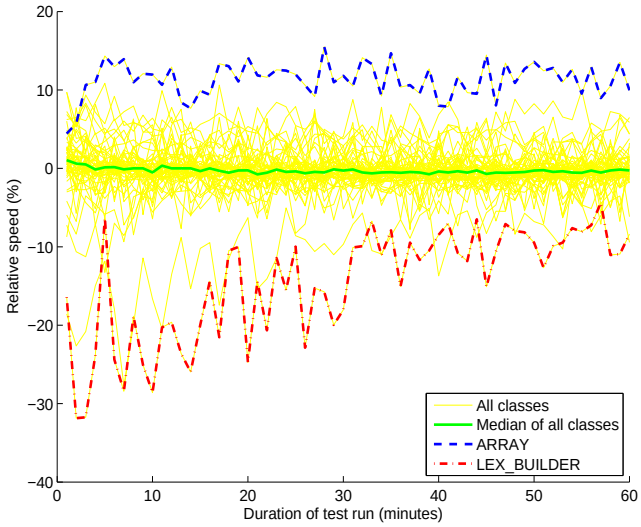


Figure 7: Test case generation speed

The two class groups with the highest and lowest speed are also highlighted in the figure with thick curves, indicating that the ps-strategy performs best on class group ARRAY and worst on class group LEX_BUILDER. Both classes have quite a few linearly-constrained preconditions. On ARRAY, the ps-strategy was 10% faster because often there are solutions to the linear constrains; on LEX_BUILDER, the solution hardly exists, so lp_solve spent a lot of time without success, resulting in a 20% overhead.

F. Success Rate of the ps-strategy

As described above, the V-pool may contain inconsistent information, which can mislead the ps-strategy to make wrong suggestions. The suggestion success rate reflects the level of consistency of the V-pool. For a class group, the success rate is measured as the percentage of the number of correct suggestions out of the total number of suggestions. Figure 8 shows the success rate for every class group over

time. Every curve represents the success rate of a class group. Depending on the class, the success rate varies in a wide spectrum, from as low as 20% to as high as 99%. For most of the class groups, the success rate is above 40%.

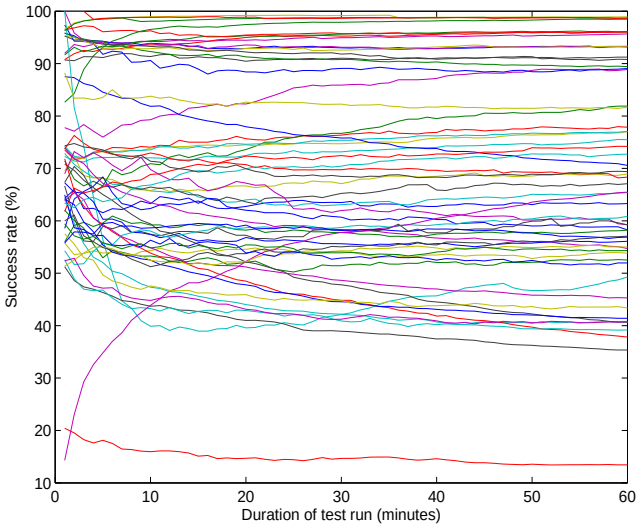


Figure 8: ps-strategy suggestion success rate

Figure 8 also shows that for many groups, the success rate goes down as testing proceeds. This suggests that the effectiveness of the ps-strategy in terms of precondition satisfaction decreases over time. The pattern is that if the class group contains routines that are not testable even by the ps-strategy, the success rate goes down. This is because as the testing proceeds, the suggestions are increasingly targeting only hard routines. Those suggestions must have very low success rate, otherwise the untestable routines would be tested.

The success rate may decrease, but it will not go down to 0, because the P_r heuristic function makes sure that all precondition-equipped routines are tested evenly often throughout the whole testing process. As a result, the success rate converges to a certain level which depends on the number of hard routines in the class. In fact, most of the curves in the figure reach a plateau. There are some curves that do not show a plateau, the reason could be that the testing time was not long enough.

V. Discussion

This section first discusses the routines still untested by the ps-strategy, then provides a remark on the importance of speed in random testing, and finally covers the threats to validity of the results.

A. Routines Untested by the ps-strategy

The ps-strategy could not test 184 routines, which are classified into the following three categories:

Unsatisfiable preconditions (19%) Some routines have unsatisfiable preconditions. This is an artifact of the class hierarchy design. For example, Listing 3 shows the interface of routine *fill* and the implementation of routine *extendible* in class ARRAY. By design, *fill*'s precondition is not satisfiable because *fill* does not make sense for ARRAY, so its precondition is "disabled". But in ARRAYED_LIST, a descendant class of ARRAY, *fill* makes sense, so its precondition is redefined to return True. AutoTest will repeatedly try to satisfy those unsatisfiable preconditions without success.

Listing 3: Unsatisfiable precondition

```
fill (other: CONTAINER)
        -- Fill with as many items of 'other' as possible.
    require
        other_not_void: other /= Void
        extendible: extendible

extendible: BOOLEAN
        -- May items be added?
    do
        Result := False
    end
```

Not supported by AutoTest or by the testing environment (32%) Routines of this type have preconditions only satisfiable on certain environment, for example, some routines requires to be tested on the .NET platform while our experiment was conducted only on Linux; or their preconditions are not satisfiable by current implementation of AutoTest.

Other (49%) This type is more interesting because it shows the limitation of the random testing strategy or the guided object selection strategy. There are four possible cases:

- Objects satisfying the desired preconditions never populate the object pool. This is because a random strategy cannot fully explore object state space.
- The object pool does contain objects satisfying the desired preconditions, but the ps-strategy does not consider them during the predicate evaluation phase because they do not occur to be relevant objects of the same passing test case.
- Objects satisfying the desired preconditions got damaged before they are selected to test the corresponding routine. This is because after precondition predicates evaluation and V-pool updating, the ps-strategy does not use the objects immediately, instead, it will continue to randomly choose the next routine to test. One way to solve this problem is to test a routine immediately after its precondition is observed to hold. We call this scheme *eager routine selection*.
- The test runs were not long enough.

Overcoming these limitations is part of our future work.

B. Importance of Speed

As in the or-strategy, speed plays an important role in the ps-strategy because the algorithm relies on randomness

to diversify the object pool, and a diversified pool greatly contributes to fault detection. In the prototype of the ps-strategy, we tried two other variations for better precondition satisfaction by sacrificing speed, but they suffered from a slow object pool diversification process, and detected fewer faults in the end:

- Iterating through all objects in the object pool searching for precondition-satisfying combinations. While this made sure that every suggestion is correct, it brought a huge overhead. Even with turning on precondition-satisfaction only from time to time, the overhead was still above 50%.
- Always enforcing precondition satisfaction for a routine. As mentioned earlier, this also came with a huge overhead.

Previous work [6] of ours also showed that speed loss can influence the effectiveness of random testing in finding faults. So the fact that the ps-strategy involves only 0.03% overhead is plausible.

C. Threats to Validity

The following three threats may influence the generalization of our results:

- Although the chosen classes have different semantics and complexities, they may not be representative for all O-O programs.
- AutoTest is one implementation of random testing using a pseudo-random number generator. We tried to keep the algorithm of AutoTest as general as possible, but other implementations of random testing may produce different results.
- At the end of a 1 hour test run, for many of the classes, the number of faults did not reach a plateau. Given more time, AutoTest may continue to find new faults. The results, especially the faults found only by the ps-strategy or the or-strategy may be different from those reported here if the classes are tested for a longer time. It is also possible that more routines can be tested with longer testing time.

VI. Related Work

Tools such as JCrasher [18], Eclat [19], Jartege [20] and Jtest [21] for random testing O-O software drew a lot of interest in recent years. Although they target languages without contracts (mostly Java), the fact that object behavior depends on its state requires implicit preconditions anyway: generating a test case to insert a value into a list at an non-existing position does not make sense. These tools either cannot detect precondition violations or ignore invalid test cases, like what AutoTest does with the or-strategy.

Adaptive random testing [5] is an enhancement of random input selection. Previous work [6] of ours showed that adaptive random testing based on *object distance* can detect new faults in O-O programs. The idea of enhancing the data selection part of the random testing strategy is similar to the work presented here but we focused on precondition satisfaction, and the method that we used was quite different.

Model-based testing is closely related to this work because it generates precondition-satisfying test cases. Spec Explorer [22] requires a model of the software under test and only generates valid test cases conforming to that model. However, software models are manually provided, while ps-strategy is fully automatic. Although tools such as ADABU [23] used contract inference technique to construct the model automatically, the test suite used for the inference is manually written.

Mock objects are also commonly used in precondition satisfaction and are usually provided manually. Pex [24] used manually provided mock objects to return values satisfying the path conditions derived from symbolic execution (can be treated as strengthened preconditions). One criticism on mocks is that one may end up testing a different program if the mocks provide inconsistent behavior. Compared to mocks, the guided object selection uses objects that were correctly constructed before.

Korat [25] generates all non-isomorphic inputs satisfying a given predicate (precondition) from a given bound size and a set of primitive objects, which are used to construct the final input object. But it is difficult to apply Korat to classes with complex internal structures, such as the regular expression based lexer we used in the experiments.

Several methods [9], [26], [27] based on evolutionary algorithms have been applied to testing O-O software. But so far, we have not seen large-scale experiments showing its applicability in general.

VII. Conclusions and Future Work

The guided object selection strategy is a fully automatic method for satisfying routine preconditions in random testing O-O programs. Compared to the original random strategy, it is able to test 56% of the routines that were not testable before, it generates 3.6 times more test cases for hard routines, finds almost 10% more faults over all tested classes and has negligible overhead. The results suggest that even though the guided object selection strategy missed some routines and faults that are tested or detected by the original strategy, it is more effective than the latter.

Future work includes: 1) testing classes in longer runs; 2) experimenting with eager routine selection for better precondition satisfaction; and 3) understanding why the guided object selection strategy misses some faults detected by the original random testing.

References

[1] I. Ciupa, A. Leitner, M. Oriol, and B. Meyer, "Experimental assessment of random testing for object-oriented software," in *ISSTA '07: Proceedings of the 2007 international symposium on Software testing and analysis*. New York, NY, USA: ACM, 2007, pp. 84–94.

[2] E. J. Weyuker and B. Jeng, "Analyzing partition testing strategies," *IEEE Trans. Softw. Eng.*, vol. 17, no. 7, pp. 703–711, 1991.

[3] R. Hamlet, "Random testing," in *Encyclopedia of Software Engineering*. Wiley, 1994, pp. 970–978.

[4] S. Ntafos, "On random and partition testing," in *ISSTA '98: Proceedings of the 1998 ACM SIGSOFT international symposium on Software testing and analysis*. New York, NY, USA: ACM, 1998, pp. 42–48.

[5] T. Y. Chen, H. Leung, and I. K. Mak, "Adaptive random testing," in *ASIAN*, ser. Lecture Notes in Computer Science, vol. 3321. Springer, 2004, pp. 320–329.

[6] I. Ciupa, A. Leitner, M. Oriol, and B. Meyer, "ARTOO: adaptive random testing for object-oriented software," in *ICSE '08: Proceedings of the 30th international conference on Software engineering*. New York, NY, USA: ACM, 2008, pp. 71–80.

[7] M. Veanes, C. Campbell, W. Grieskamp, W. Schulte, N. Tillmann, and L. Nachmanson, "Model-based testing of object-oriented reactive systems with spec explorer," in *Formal Methods and Testing*, ser. Lecture Notes in Computer Science, R. M. Hierons, J. P. Bowen, and M. Harman, Eds., vol. 4949. Springer, 2008, pp. 39–76.

[8] D. Thomas and A. Hunt, "Mock objects," *IEEE Software*, vol. 19, no. 3, pp. 22–24, 2002.

[9] A. Arcuri and X. Yao, "Search based testing of containers for object-oriented software," University of Birmingham, School of Computer Science, Tech. Rep. CSR-07-3, April 2007. [Online]. Available: ftp://ftp.cs.bham.ac.uk/pub/tech-reports/2007/CSR-07-3.pdf

[10] B. Meyer, I. Ciupa, A. Leitner, and L. L. Liu, "Automatic testing of object-oriented software," in *SOFSEM '07: Proceedings of the 33rd conference on Current Trends in Theory and Practice of Computer Science*. Berlin, Heidelberg: Springer-Verlag, 2007, pp. 114–129.

[11] B. Meyer, A. Fiva, I. Ciupa, A. Leitner, Y. Wei, and E. Stapf, "Programs that test themselves," *IEEE Software*, pp. 22–24, 2009.

[12] Eiffel Software, "EiffelStudio 6.4 GPL Edition," http://www.eiffel.com/.

[13] Gobo Eiffel Project. Gobo library (revision 6661). http://freeelks.svn.sourceforge.net.

[14] lp_solve linear programming solver 5.5. http://lpsolve.sourceforge.net/5.5.

[15] D. Hoffman, P. A. Strooper, and L. J. White, "Boundary values and automated component testing," *Softw. Test., Verif. Reliab.*, vol. 9, no. 1, pp. 3–26, 1999.

[16] Eiffel Software. Eiffelbase library (revision 356). http://freeelks.svn.sourceforge.net.

[17] I. Ciupa, A. Pretschner, A. Leitner, M. Oriol, and B. Meyer, "On the predictability of random tests for object-oriented software," in *ICST '08: Proceedings of the 2008 international conference on Software testing, verification, and validation*. Washington, DC, USA: IEEE Computer Society, 2008, pp. 72–81.

[18] C. Csallner and Y. Smaragdakis, "JCrasher: an automatic robustness tester for java," *Softw. Pract. Exper.*, vol. 34, no. 11, pp. 1025–1050, 2004.

[19] C. Pacheco and M. D. Ernst, "Eclat: Automatic generation and classification of test inputs," in *ECOOP*, ser. Lecture Notes in Computer Science, A. P. Black, Ed., vol. 3586. Springer, 2005, pp. 504–527.

[20] C. Oriat, "Jartege: A tool for random generation of unit tests for java classes," in *QoSA/SOQUA*, ser. Lecture Notes in Computer Science, R. Reussner, J. Mayer, J. A. Stafford, S. Overhage, S. Becker, and P. J. Schroeder, Eds., vol. 3712. Springer, 2005, pp. 242–256.

[21] Parasoft Corporation, "Jtest," http://www.parasoft.com/.

[22] "Spec Explorer tool. Microsoft Research," http://research.microsoft.com/specexplorer.

[23] V. Dallmeier, C. Lindig, A. Wasylkowski, and A. Zeller, "Mining object behavior with ADABU," in *WODA '06: Proceedings of the 2006 international workshop on Dynamic systems analysis*. New York, NY, USA: ACM, 2006, pp. 17–24.

[24] N. Tillmann and W. Schulte, "Unit tests reloaded: Parameterized unit testing with symbolic execution," *IEEE Softw.*, vol. 23, no. 4, pp. 38–47, 2006.

[25] C. Boyapati, S. Khurshid, and D. Marinov, "Korat: automated testing based on java predicates," in *ISSTA '02: Proceedings of the 2002 ACM SIGSOFT international symposium on Software testing and analysis*. New York, NY, USA: ACM, 2002, pp. 123–133.

[26] P. Tonella, "Evolutionary testing of classes," in *ISSTA '04: Proceedings of the 2004 ACM SIGSOFT international symposium on Software testing and analysis*. New York, NY, USA: ACM, 2004, pp. 119–128.

[27] Y. Cheon and M. Kim, "A specification-based fitness function for evolutionary testing of object-oriented programs," in *GECCO '06: Proceedings of the 8th annual conference on Genetic and evolutionary computation*. New York, NY, USA: ACM, 2006, pp. 1953–1954.