Evotec: Evolving the Best Testing Strategy for Contract-Equipped Programs

Lucas Serpa Silva, Yi Wei, Bertrand Meyer
Chair of Software Engineering
ETH Zurich, Switzerland
Email:lssilva@gmail.com,
{*yi.wei,bertrand.meyer*}*@inf.ethz.ch*

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

***Abstract*—Automated random testing is effective at detecting faults but it is certainly not an optimal testing strategy for every given program. For example, an automated random testing tool ignores that some routines have stronger preconditions, they use certain literal values, or they are more error-prone. Taking into account such characteristics may increase testing effectiveness.**

In this article, we present *Evotec*, an enhancement of random testing which relies on genetic algorithms to evolve a best testing strategy for contract-equipped programs. The resulting strategy is optimized for detecting more faults, satisfying more routine preconditions and establishing more object states on a given set of classes to test.

Our experiment tested 92 classes over 1710 hours. It shows that *Evotec* detected 29% more faults than random+ and 18% more faults than the precondition-satisfaction strategy.

***Keywords*-Automated Software Testing, Genetic Algorithm, Static-Analysis**

I. Introduction

Random testing [1] is widely used because it is effective in detecting faults [2], [3] as well as easy to implement. One issue with a random strategy, however, is that this simplicity comes at the cost of only using straightforward strategies. In particular, it does not leverage on information from past executions.

Most commonly, a random strategy tests all routines with the same frequency and use primitive values (integers, reals) with a fixed probability. But in reality, routines may have specific characteristics which need special treatment. For example, some routines use certain literal values, contain more errors, or have preconditions which are hard to satisfy. An optimized testing strategy should take into account such characteristics and test these routines more thoroughly to maximize the effectiveness of testing. The difficulty is that static analysis alone is not enough to provide an optimized version of the testing session as the number of literals, the complexity of the contracts, or even the number of errors in the code that are loosely associated to the actual failure rate obtained with a specific strategy. We thus need smarter strategies for random testing.

This paper presents a fully automated random-based testing strategy *ev-strategy* which adapts and optimizes itself for the classes under test in order to maximize the number of detected faults, the number of satisfied preconditions and the number of established object states. The *ev-strategy* uses a genetic algorithm [4] to evolve a set of randomly generated test suites into a best testing strategy for a given set of classes, and then uses that best strategy to test the classes. Genetic algorithms are commonly used [5]–[7] in testing tools, but mostly with the focus on maximizing code coverage. None of them maximizes at the same time the number of faults, contract-satisfying ability and object states.

We implemented the *ev-strategy* in a tool called *Evotec*[1] which builds on top of AutoTest [8], an automated testing framework for Eiffel. In a large scale experiment involving 92 classes and 1710 hours of testing time, we compared the *ev-strategy* with two other existing testing strategies available in AutoTest:

- The original random+ [9] testing strategy (the *rp-strategy*), with no optimization at all.
- The precondition-satisfaction strategy [10] (the *ps-strategy*) which aims at satisfying the most preconditions of the routines under test.

The results show that the *ev-strategy* detected 29% more faults than the *rp-strategy* and 18% more faults than the *ps-strategy*.

Section II provides background information on contract-based testing and genetic algorithms; Section III describes the *ev-strategy* in detail; Section IV presents the experiments evaluating the effectiveness of the *ev-strategy*; Section VI lists relevant research; and we conclude in Section VII.

II. Background

This section presents background information for contract-based random testing and genetic algorithm.

A. Contract-Based Random Testing

A test consists of two parts: test inputs and test oracles. Test inputs drive the software under test into certain execution path, and test oracles decide if the execution or the results from the execution are valid. Various test input generation methods exists. To a large extent, building

[1]Source code of the tool and experiment details are available at http://evotec.origo.ethz.ch/

automatically test oracles remains an open issue in the testing community. Most of the testing tools either rely on manual inspection of the testing results or use code coverage as their test oracle.

Design by Contract [11] is a software methodology in which every routine is associated with contracts in forms of pre and postcondition assertions. A precondition expresses the constraints under which a routine will function properly. A postcondition expresses properties of the state resulting from a routine's execution. Design by Contract is natively supported in the Eiffel programming language [12]. In practice, programmers write (partial) contracts systematically [13]. As an example, Listing 1 demonstrates the contracts for routine put from the class ARRAY.

Listing 1. Pre and postcondition of routine *put* from class *ARRAY*

```
put (v: like item; i: INTEGER)
    -- Replace i-th entry, by v.
  require i ≥ lower and i ≤ upper
  ensure item (i) = v
```

The precondition (require clause) specifies that i must be within the index range [lower, upper] of the array. The postcondition (ensure clause) specifies that after the routine's execution, v must be put at the i-th entry.

Contracts greatly facilitate testing. Preconditions serve as input filters and postconditions serve as test oracles. This enables random testing as a fully automated strategy, as implemented in AutoTest: objects of both primitive and reference types are generated randomly. Only inputs satisfying the precondition of the routine under test are passed to execution, and if there is a postcondition violation during the routine's execution, a fault[1] is detected.

AutoTest uses the following original *rp-strategy* to construct objects needed for routines under test:

- To select an object of primitive type, AutoTest either generates one randomly or picks one from a predefined set of *potentially interesting* values, such as 0, ±1, ±2, ±10, ±100.
- To select an object of reference type, AutoTest either creates a new one by calling a constructor from the underlying class or picks an existing object of conforming type. All created objects or objects returned from a routine call are kept for further selection because objects with diversified states are valuable for detecting faults.

Experiments showed that this *rp-strategy*, even though seems to be naive, is effective at detecting faults [2].

One problem with the above *rp-strategy* is that for routines with strong preconditions, random selection may fail to select even a single valid input, leaving those routines untested at all. The idea behind the precondition-satisfaction strategy *ps-strategy* is to increase the likelihood of selecting a precondition-satisfying object from the object pool. It is an extension to the *rp-strategy* strategy. Besides the object pool, the *ps-strategy* keeps track which objects satisfy which precondition assertions. During object selection, the *ps-strategy* choose those precondition-satisfaction objects with higher probability. Experiments [10] showed that the *ps-strategy* was able to test more routines and detected slightly more faults than the *rp-strategy*.

Both the *rp-strategy* and the *ps-strategy* treats classes routines and values under test equally and do not take into account any static or runtime profiling of the program under test.

B. Genetic Algorithms

Genetic algorithms are robust [4] search algorithms based on natural selection. The algorithm starts by initializing or randomly generating a set of chromosomes and at the end of each generation, each chromosome is evaluated based on a fitness function. The next generation is created using the best chromosomes of the previous and generating new ones using two mechanisms: *breeding* and *mutation*. Breeding essentially consists in mixing two or more chromosomes to producing a third one, while mutation consists in replacing random parts of a chromosome with random values. This process repeats for a predefined number of generations or until the objective value of the population has converged.

Genetic algorithms are widely used in testing, where chromosomes encode testing strategies, such as routines to test and arguments for those routines. Most of the genetic algorithm based testing tools optimize code coverage [5]–[7]. In this work, we use a generic algorithm to optimize the fault detecting ability, contract-satisfaction ability and object state diversification of a testing strategy. Our chromosomes are the calls and arguments that we use during testing.

III. Evolutionary Testing

This section first gives an overview (Section III-A) of how the *ev-strategy* works and then it provides details of the technique: how to encode (Section III-B) and rank (Section III-C) a testing strategy, how to evolve the best strategy from random ones (Section III-D) and finally how to apply the evolved best strategy to classes under test (Section III-E).

A. Overview

The *ev-strategy* is fully automated. Figure 1 provides an overview of how it proceeds when given a set of classes under test:

1) Extracting literal values such as integers from those classes. These literal values are potentially interesting as test inputs.
2) Running AutoTest, the random testing framework for Eiffel to collect an initial set of test suites.

[1]A failed test execution reveals an exception. We map exceptions happening at the same location and with the same type to the same fault

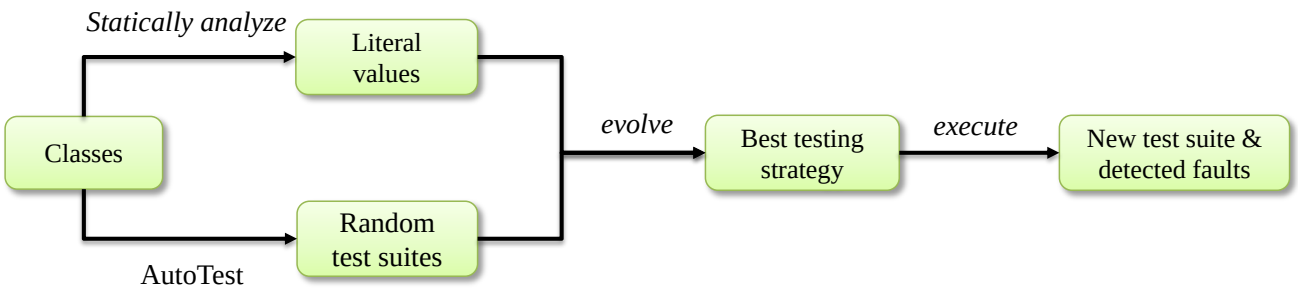

Figure 1. Overview of how the *ev-strategy* works.

3) The *ev-strategy* first encodes the test suites and literal values as testing strategy chromosomes, then it uses a genetic algorithm to mutate and evolve those strategies according to a fitness function. The fitness function favours strategies which detect more faults, satisfy more routine preconditions and induce more diversified object states.
4) After strategy evolution, the *ev-strategy* selects the best strategy and executes it on classes under test to generate the final test suite.

B. Encoding Testing Strategies

Components of Testing Strategies A testing strategy consists of three sets of parameters:

- Primitive values: specifies a set of values for each of the five primitive types (boolean, natural, integer, real and character).
- Routine call sequence: a sequence of invocations to the routines under test. Each invocation specifies the target and argument objects for that routine.
- Object pool: A collection of objects of both primitive and reference types, used to provide candidate objects for routine invocations. The object pool has restrictions on the maximal number of objects for each type.

For primitive types, the *rp-strategy* and the *ps-strategy* use both randomly generated values as well as values from a predefined set, such as 0, ± 1, ± 2, ± 10, ± 100. Although previous experiment [2] showed that those predefined primitive values contribute to detecting more faults, it is unlikely that they are optimal for every group of classes.

For reference types, the *rp-strategy* and the *ps-strategy* both try to create new objects via constructor methods or reuse existing ones from the object pool. The *rp-strategy* treats each object equally, and the *ps-strategy* selects precondition-satisfying objects with a higher probability. Compared to these two strategies, the *ev-strategy* takes the fault-detecting ability, the precondition-satisfying ability and the object state diversification into account, favouring objects that reveal more faults, satisfy more preconditions and diversify in more object states.

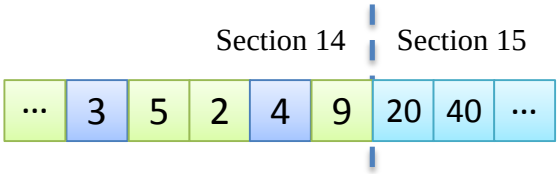

Figure 2. Testing strategy encoding

Encoding a Testing Strategy A testing strategy is encoded as an array of floating numbers with 15 sections. Each section holds values for a parameter in the testing strategy. Table I lists these 15 parameters (Column PARAMETER), their section length (Column #VALUES) and the range of the values in each section (Column RANGE).

The first 13 sections store 20 values for each primitive type. These values are used when instantiating objects. Section 15 encodes the maximum number of objects for each type in the object pool. Section 14 as illustrated in Figure 2 encodes the routine calls.

The chromosome does not specify the specific routine or object to be used but instead specifies indexes. Figure 2 provides a simplistic illustration of a chromosome. In this scenario, to test a routine *Evotec* reads the value 3 from the chromosome, and use it as an index to select a routine from the table of possible routines to test.

Having selected the routine, *Evotec* then verifies how many parameter this routine takes, in this case two parameters so it reads value 5 and 2 from the chromosome. It checks the type of the first parameter, retrieves a table of compatible objects and uses the object with index 5. This process is repeated for each parameter.

Since *Evotec* knows which types are needed to execute each routine, the chromosome just needs to specify which object from the list of possible objects has to be used.

Because the number of routines and the number of available types is not known in advance, the values from the chromosome may specify invalid indexes, thus a real index is computed:

$$realIndex \equiv chromosomeIndex \bmod listSize \quad (1)$$

Where the *listSize* is the size of the list of routines when computing the indexes of routines to be called or the size of the list of available objects of a given type when computing the indexes of objects for a routine call.

With this approach, adding or removing a routine call is very simple. Whenever a mutation makes the *realIndex* equals zero, the routine call is removed and when the *realIndex* is modified from zero to a different number, a routine call is added. With this approach, different mutations and crossover methods can be used without having to worry about corrupting the chromosome.

Table I
CHROMOSOME ALLELE SPECIFICATION

SECTION	PARAMETER	#VALUES	RANGE[FROM,TO]
1	BOOLEAN	20	-1,1
2	CHARACTER_32	20	0 , 600
3	CHARACTER_8	20	0 , 255
4	INTEGER_16	20	-32768, 32767
5	INTEGER_32	20	-2147483648, 2147483647
6	INTEGER_64	20	-9223372036854775808, 9223372036854775807
7	INTEGER_8	20	-128, 127
8	NATURAL_16	20	0, 65535
9	NATURAL_32	20	0, 4294967295
10	NATURAL_64	20	0, 1.84E+019
11	NATURAL_8	20	0, 255
12	REAL_32	20	-1.0e30, 1.0e30
13	REAL_64	20	-1.0e30, 1.0e30
14	ROUTINE_CALL	5000	0, 500
15	OBJECT_POOL	500	0, 100

C. Ranking Testing Strategies

To guide the evolution, the genetic algorithm requires a fitness function which decides how *good* a testing strategy is. The fitness function is defined as:

$$\theta = (10000 * \alpha) + (1000 - (10 * \beta) + \lambda) + \omega \qquad (2)$$

This function consists of four components:

1) Unique number of faults: α is the number of unique faults detected by a strategy.
2) Number of unique states: β is the number of unique object states established by a strategy. For an object, we abstract its state by a vector of its field values.
3) Number of untested routines: λ is the number of routines without a valid test case during the execution of a strategy.
4) Precondition score: ω is the measure of how close the strategy was to successfully generate tests for untested routines. These routines were not tested because their preconditions are not satisfied. ω is the sum of the number of precondition assertions that a strategy is able to satisfy for the untested routines.

The number of unique faults detected is the most important measurement of how good a testing strategy is, therefore α has the highest factor (10000). When two strategies find the same number of faults, the strategy that has tested more routines and tested them in more diversified object states is considered better. When two strategies have the same score for α, β and λ, the strategy that was able to satisfy more preconditions for the untested routines is considered better. The precondition score is used because a testing strategy may have difficulties to generate test data for routines with strong preconditions. Thus the precondition score favours strategies which satisfy more expressions in those preconditions.

To calculate α, β, λ and ω for a testing strategy, the *ev-strategy* executes the strategy for 1 minute and collects the required data from the execution. A testing strategy is considered as better if its θ value is larger.

D. Evolving the Best Strategy

The *ev-strategy* starts by initializing 16 individual testing strategies as the first population. The *ev-strategy* uses a random strategy combined with static-analysis to generate this initial set of testing strategies. In our implementation, a modified version of the *rp-strategy* is used. This modified version first applies a simple static analysis over the classes under test to extract literal primitive values, and then feeds the extracted values to the random testing strategy. During initialization, a probability of 0.8 is used to decide whether to use a value from the extracted literal values, as opposed to use a randomly generated value. The nature of the random strategy ensures that the generated population consists of individuals which are not too close to each other.

The *ev-strategy* uses the fitness function to evaluate and rank these 16 testing strategies, and applies the *stochastic reminder* [14] algorithm to select the best half (8 of them) of the strategies based on their rankings to keep for the next generation.

To introduce diversities in those kept strategies, the *ev-strategy* applies the *flip mutator* algorithm on them: Except for the best scoring strategy, the flip mutator algorithm replaces the values of some genes in the chromosome by random values.

To form a new generation of testing strategies, the *ev-strategy* still needs to construct another 8 strategies. There is 60% of chance that a new strategy will be generated randomly and 40% of chance that a new strategy will be produced from two strategies kept as parents (this 60/40 ratio was decided using the result of preliminary experiments). Generating a strategy randomly is straightforward: the *ev-strategy* initializes a chromosome with all random values. To produce a new strategy from two existing ones, the *ev-strategy* uses the *partial match* crossover algorithm. The partial match crossover algorithm combines two testing strategies P1 and P2 to produce two new strategies C1 and C2. It randomly selects a number of positions and swap the genes between C1 and C2 at those positions.

This testing strategy evaluation, selection and mutation process is repeated for each generation.

We use the GAlib [14] framework to implement the genetic algorithm. GAlib accepts a set of chromosomes (encoding testing strategies in our case), a fitness function and some configuration parameters as inputs, and outputs a chromosome considered as the *best* according to the fitness function and time given for evolution. The configuration parameters define the strength of mutation and the length of the evolution. Table II summarizes the parameters.

The population size and number of generation were selected based on the time allowed for evolution. Although only one minute was used for testing, a portion of the time was consumed processing the results of each execution run.

Table II
GENETIC ALGORITHM PARAMETERS

PARAMETER	VALUE
Population size	16
Population replacement	8
Number of generations	4
Crossover probability	0.4
Mutation probability	0.4
Crossover algorithm	Partial Matching
Selection algorithm	Stochastic Remainder
Mutation algorithm	Flip Mutator

E. Applying the Best Strategy

After the genetic algorithm reports a best testing strategy, the *ev-strategy* applies it to perform the real testing on the given classes. The application is straightforward: given a time frame, the *ev-strategy* invokes routines with arguments as specified in the best strategy in order. If all the indexes in the section have all been used and there is still some time left, it will loop and start reading the indexes from beginning. Note that this repetition does not result in the same testing effort because the states of the objects change during each run and due the shifting of the indexes. After a loop the first few indexes may be used as a routine call arguments instead of a routine call, leading to a new sequence of routine calls.

IV. EXPERIMENTAL EVALUATION

This section presents our experimental evaluation for the *ev-strategy*. The experiments applied the *ev-strategy* to 92 classes in a total of 1710 hours and the results are compared with the outcome from both the *ps-strategy* and the *rp-strategy* retrieved in the same experimental setting [10].

Classes under test. The classes under test are from two Eiffel libraries EiffelBase [15] and Gobo [16]. Both libraries have long development history and are widely used in Eiffel community. These classes implement common data structures such as lists, stacks, queues, tables, trees and a lexer based on regular expressions. Left part of Table III summaries categories of the chosen classes (CATEGORY), the number of classes in each category (#C), the number of routines (#R) and then number of pre and postcondition assertions in those classes (#PRE and #POST).

Test runs. These 92 classes were arranged into 57 groups, with strongly related classes put into the same group. For example, DS_ARRAYED_LIST and DS_ARRAYED_LIST_CURSOR were put into the same group because the former represents an arrayed list and the latter represents an external iterator of that list. According to our previous experience, classes can be tested more thoroughly when their strongly related classes are tested together.

Previous work [17] showed that random testing can find different faults with different seeds to the pseudo random number generator. The *ev-strategy* relies on random testing to generate initial test suites, in order to achieve statistical significant result, all 57 class groups were tested with the *ev-strategy* in 30 runs, with each run initialized with a different seed to the pseudo random number generator and lasts one hour long, resulting in a total of 1710 testing hours.

In each test run, the *ev-strategy* uses 27 minutes to evolve the best strategy, and use the left 33 minutes to perform actual testing using the best strategy. Note that this time arrangement is different from experiments for the *rp-strategy* and the *ps-strategy*, in which all 60 minutes are used for actual testing, because these two strategies do not need an explicit preparation time.

Computation infrastructure. The experiments were conducted in a grid of dedicated machines with an Intel Pentium 4 CPU at 3 GHz and 1 GB of RAM running Red Hat Enterprise Linux 5.3.

A. Experimental Results

The experiments show that the *ev-strategy* detected more faults than both the *rp-strategy* and the *ps-strategy*. The right part of Table III lists the number of unique faults[2] detected by the *ev-strategy* (#F_{ev}), by the *ps-strategy* (#F_{ps}), and by the *rp-strategy* (#F_{rp}). In total the *ev-strategy* detected 18% more faults than the *ps-strategy* and 29% more faults than the *rp-strategy*. The following sections analyze the results in detail.

B. Number of faults detected over time

Since the *ev-strategy* as well as the *ps-strategy* and the *rp-strategy* are fully automated, the most important criterion to compare them is to compare the number of detected faults. Figure 3 plots the number of unique faults that each strategy detected over time. In Figure 3, the x-axis is the testing time in minutes, and the y-axis is the number of unique faults detected.

Figure 3 shows that the *ev-strategy* outperforms the other two strategies by a large portion, while the *ps-strategy* only performs slightly better than the *rp-strategy*: in total, the *ev-strategy* detected 641 faults, the *ps-strategy* detected 539

[2]During the testing period, a fault can be detected multiple times, we do not count repeatedly detected faults.

Table III
METRICS FOR TESTED CLASSES AND EXPERIMENTAL RESULTS

CATEGORY	#C	LOC	#R	#PRE	#POST	VARIATIONS	$\#F_{rp}$	$\#F_{ps}$	$\#F_{ev}$
Lexer	30	32,108	2,914	2,332	2,717	regular expression, NFA, DFA, lexer	100	145	100
List	24	15,482	2,237	1,707	2,025	array, single, double, bidirectional, sorted	169	169	266
Hashed	6	5,156	706	528	672	hash table	7	6	7
Queue	4	7,135	318	173	259	bounded, unbounded, priority	17	19	28
Set	7	15,471	791	655	713	binary tree based, array based, hashed, sorted	36	29	45
Stack	1	1,281	59	27	53	linked list based	4	4	4
String	1	4,815	222	161	236	array based	9	11	18
Tree	19	16,102	1,968	1,477	1,637	binary, n-nary, AVL, red black, search tree	153	156	173
Total	**92**	**97,550**	**9,215**	**7,060**	**8,312**		**495**	**539**	**641**

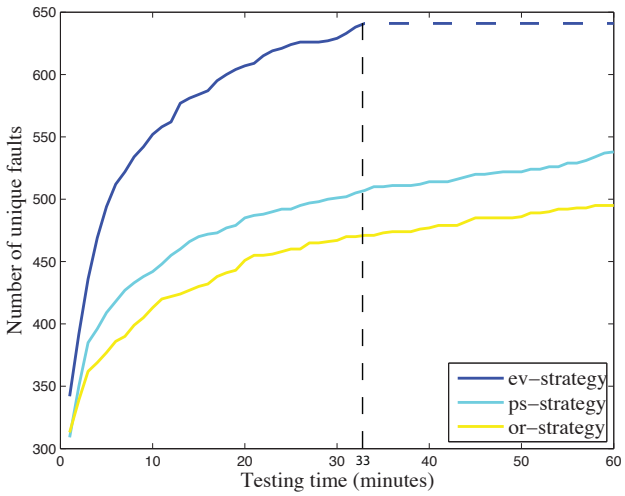

Figure 3. Unique faults detected over time

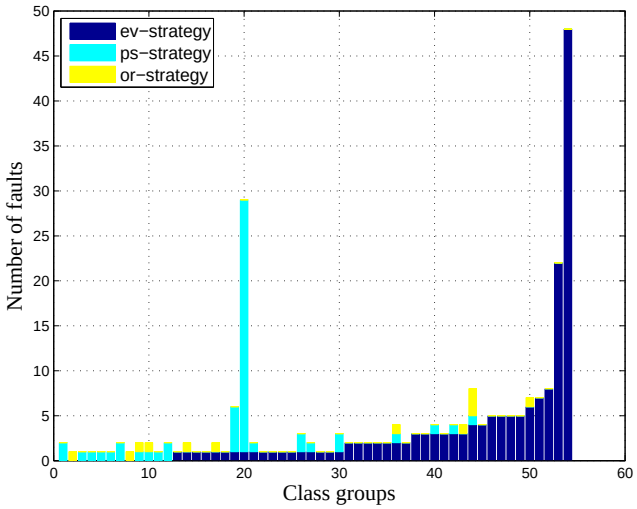

Figure 4. Faults detected by a single strategy

faults and the *rp-strategy* detected 495 faults. In other words, the *ev-strategy* detected 18% more faults than the *ps-strategy* and 29% more faults than the *rp-strategy*.

Note that there is a plateau (starting from the 33th until the 60th minutes) for the curve representing the *ev-strategy* strategy. This plateau does not suggest that the *ev-strategy* could not detect more faults after the first half an hour, it is due to the fact that the actual testing time for the *ev-strategy* is 33 minutes because the first 27 minutes are used for evolving the best testing strategy for the classes under test. We cut off the evolutionary testing after 33 minutes to keep the comparison to other testing strategies fair (all the strategies are strictly restricted to one hour).

Since throughout all the testing time, the number of faults detected by the *ev-strategy* increases consistently, we think that given more time, the *ev-strategy* can detect more faults, hence outperforms the other strategies even more.

C. Kinds of faults detected by each strategy

Figure 3 shows that different strategies detected different number of faults, but are they detecting different faults? In the experiments, there are 736 unique faults detected in total, 440 of them are detected at least once by all three strategies. However, there are some faults which were only detected by some strategy. Figure 4 shows the number of faults that are only detected by a single strategy. In Figure 4, the x-axis iterates over different class groups; and the y-axis represents the relative number of faults detected by a strategy. There are only 54 groups in Figure 4, instead of 57, because in those missing 3 groups, all three strategies detected the same faults.

Figure 4 shows that the *ev-strategy* detected more faults that could not be detected by the other two strategies, reflected by the fact that the area taken by the *ev-strategy* bars is larger than the ones taken by the *ps-strategy* or the *rp-strategy* bars. In total, the *ev-strategy* detected 169 faults that are not detected by the other two strategies (for the *ps-strategy* and the *rp-strategy*, the number is 95 and 12, respectively).

Ideally, we would like the *ev-strategy* to be able to detect all the faults that can be detected by other strategies, but Figure 4 shows this is not the case: practically, this means in order to detect as many faults as possible, all strategies should be used.

Figure 4 reveals that certain strategies are more effective at detecting faults for certain classes. Take the two highest

bars for instance, one is from the *ps-strategy* and the other is from the *ev-strategy*. They reveal that 28 out of 56 faults exclusively found by the *ps-strategy* were found in the LEX_BUILDER class alone, and 48 of 169 exclusively found faults by EV were found in the ARRAY class. This shows that 50% of the improvement achieved by the *ps-strategy* and 28% by the *ev-strategy* comes from a single class.

We analyzed further the faults found exclusively by *ps-strategy* and *rp-strategy*. With the exception of two faults found by *ps-strategy* in respectively 46% and 86% of the runs on given classes, none of the faults was found in more than 2 of the 30 runs (6.6%) on classes in which they could be found. This makes it very unlikely to discover them in a single run of the tool. Comparatively, the faults found solely by *ev-strategy* were found on average in 18% of the runs on classes in which they could be detected. We believe that if we were able to test longer during the discovery phase, we would improve this number.

V. Threats to Validity

The following threats may affect the generalization of the experimental results to other programs: (1) even though we tried to choose classes with different semantics and complexity, they may not be representative of programs in general, (2) due to time limit, the test runs in the experiments may not be long enough, (3) the experiments may deliver different results with different genetic algorithms or different parameters.

VI. Related Work

The full automation of unit testing has recently gained momentum with the development of numerous completely automated testing tools [8], [18]–[21]. Three main kinds of tools currently exist based respectively on static analysis, random techniques, and evolutionary strategies.

Java PathFinder [18] and Symstra [19] successfully apply symbolic execution techniques to optimize the branch-coverage of test suites. Increasing the branch-coverage, however, is not sufficient to increase the number of bugs found in a program [22]. None of these approaches use genetic algorithms.

Numerous tools such as DART [20], AutoTest [8], implement random testing. Other tools such as DSD-Crasher [21] use invariants derived with Daikon [23], whose quality depends on the diversity of observed states during the executions. None of these tools has integrated any aspects of evolutionary strategies. In this paper we compare our evolution strategy with two other random-based strategies implemented in AutoTest and show that it outperforms them both. It is likely that other testing tools would also benefit from integrating some similar techniques to ours.

Genetic algorithm has also been used to automate unit testing. Since the early 1990s, a number of studies have been conducted on evolutionary testing, but the impact and applicability of these studies to the software industry vary. The type of the input data being generated is an important attribute of an automated tester. There has been a number of studies in evolutionary testing focused on how to generate test cases for procedural programs [24]–[30], but none of these approaches applied to object-oriented languages.

Some studies explored how to generate test cases for object-oriented programs [5]–[7], [31]. These generally use branch coverage as the optimization parameter. There is however little evidence of a correlation between branch coverage and the number of uncovered faults [22]. Past research has shown that evolutionary testing is a good approach to automate the generation of test cases for structured programs [27], [32], [33] but to make this approach attractive today, the system must be able to automatically generate test cases for object-oriented programs and to use a good set of metrics to measure the quality of the generated test cases. Our approach uses the random generation of test cases of object-oriented programs as a starting point and optimizes it further by using genetic algorithms to increase the number of faults found and established object states. It is unclear how other techniques would fare in such a context.

VII. Conclusions

This paper presents a fully automated testing technique which uses a genetic algorithm to optimize random testing sessions for given programs. The resulting testing strategy is optimized for detecting more faults, satisfying more routine preconditions and establishing more object states.

Experiments applying this technique to 92 classes showed an improvement of at least 18% on the number of detected faults compared to original random testing. These results suggest that adapting the testing strategy to the class under test can considerably improve testing effectiveness.

We implemented the technique in the open-source *Evotec* tool. *Evotec* is written in Eiffel and works on Eiffel classes. The technique described in the paper is however applicable to other object-oriented languages that support contracts, such as JML for Java and Spec# for C#.

References

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

[2] 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.

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

[4] D. E. Goldberg, *Genetic Algorithms in Search, Optimization, and Machine Learning*. Addison-Wesley Professional, January 1989.

[5] 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.

[6] S. Wappler and F. Lammermann, "Using evolutionary algorithms for the unit testing of object-oriented software," in *GECCO '05: Proceedings of the 2005 conference on Genetic and evolutionary computation*. New York, NY, USA: ACM, 2005, pp. 1053–1060.

[7] S. Mairhofer, "Search-based software testing and complex test data generation in a dynamic programming language," *Master's thesis, Blekinge Institute of Technology*, 2008.

[8] 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.

[9] I. Ciupa, B. Meyer, M. Oriol, and A. Pretschner, "Finding Faults: Manual Testing vs. Random+ Testing vs. User Reports," in *2008 19th International Symposium on Software Reliability Engineering (ISSRE)*. Ieee, Nov. 2008, pp. 157–166. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=4700320

[10] Y. Wei, S. Gebhardt, B. Meyer, and M. Oriol, "Satisfying test preconditions through guided object selection," *Software Testing, Verification, and Validation, 2008 International Conference on*, vol. 0, pp. 303–312, 2010.

[11] B. Meyer, *Object-Oriented Software Construction*, 2nd ed. Prentice Hall PTR, March 2000.

[12] ECMA, *Eiffel: Analysis, Design and Programming Language*, 2nd ed., ECMA, http://www.ecma-international.org/publications/standards/Ecma-367.htm, 2005.

[13] P. Chalin, "Are practitioners writing contracts?" in *The RODIN Book*, ser. LNCS, vol. 4157, 2006, p. 100.

[14] M. Wall, *GAlib: A C++ Library of Genetic Algorithm Components.*, MIT, http://lancet.mit.edu/ga/, 1996.

[15] Eiffel Software, "Eiffelbase," http://freeelks.svn.sourceforge.net.

[16] Eric Bezault et al, "Gobo library and tools," http://www.gobosoft.com.

[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*. Washington, DC, USA: IEEE Computer Society, 2008, pp. 72–81.

[18] W. Visser, C. S. Păsăreanu, and S. Khurshid, "Test input generation with java pathfinder," in *ISSTA '04: Proceedings of the 2004 ACM SIGSOFT international symposium on Software testing and analysis*. New York, NY, USA: ACM, 2004, pp. 97–107.

[19] T. Xie, D. Marinov, W. Schulte, and D. Notkin, "Symstra: A framework for generating object-oriented unit tests using symbolic execution," 2005.

[20] P. Godefroid, N. Klarlund, and K. Sen, "Dart: directed automated random testing," in *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*. New York, NY, USA: ACM, 2005, pp. 213–223.

[21] C. Csallner, Y. Smaragdakis, and T. Xie, "Dsd-crasher: A hybrid analysis tool for bug finding," *ACM Trans. Softw. Eng. Methodol.*, vol. 17, no. 2, pp. 1–37, 2008.

[22] Y. Wei, M. Oriol, and B. Meyer, "Is coverage a good measure of testing effectiveness," ETH Zurich, Tech. Rep. 674, 2010.

[23] M. D. Ernst, J. CockrelI, W. G. Griswold, and D. Notkin, "Dynamically discovering likely program invariants to support program evolution," *Software Engineering, International Conference on*, vol. 0, p. 213, 1999.

[24] J. Hunt, "Testing control software using a genetic algorithm," *Engineering Applications of Artificial Intelligence*, vol. 8, no. 6, pp. 671–680, 1995.

[25] J. T. Alander, T. Mantere, and P. Turunen, "Genetic algorithm based software testing," 2007.

[26] B. F. Jones, H. H. Sthamer, and D. E. Eyres, "Automatic structural testing using genetic algorithms," *Software Engineering Journal*, vol. 11, no. 5, pp. 299–306, 1996.

[27] C. C. Michael, G. McGraw, and M. A. Schatz, "Generating software test data by evolution," *IEEE Trans. Softw. Eng.*, vol. 27, no. 12, pp. 1085–1110, 2001.

[28] E. Alba and J. Chicano, "Software testing with evolutionary strategies," *Rapid Integration of Software Engineering Techniques*, pp. 50–65, 2006.

[29] P. McMinn and M. Holcombe, "Evolutionary testing of state-based programs," in *GECCO '05: Proceedings of the 2005 conference on Genetic and evolutionary computation*. New York, NY, USA: ACM, 2005, pp. 1013–1020.

[30] N. Tracey, J. Clark, and K. Mander, "Automated program flaw finding using simulated annealing," *SIGSOFT Softw. Eng. Notes*, vol. 23, no. 2, pp. 73–81, 1998.

[31] J. Wegener, "Evolutionary unit testing of object-oriented software using a hybrid evolutionary algorithm," 2010.

[32] M. Harman and P. McMinn, "A theoretical & empirical analysis of evolutionary testing and hill climbing for structural test data generation," in *ISSTA '07: Proceedings of the 2007 international symposium on Software testing and analysis*. New York, NY, USA: ACM, 2007, pp. 73–83.

[33] H. Sthamer, "The automatic generation of software test data using genetic algorithms," PhD Thesis, University of Glamorgan, Pontyprid, Wales, Great Britain, 1996.