EvoSE: Evolutionary Symbolic Execution

Mauro Baluda
Fraunhofer SIT
Darmstadt, Germany
mauro.baluda@sit.fraunhofer.de

ABSTRACT

Search Based Software Testing (SBST) and Symbolic Execution (SE) have emerged as the most effective among the fully automated test input generation techniques. However, none of the two techniques satisfactorily solves the problem of generating test cases that exercise specific code elements, as it is required for example in security vulnerability testing.

This paper proposes EvoSE, an approach that combines the strengths of SBST and SE. EvoSE implements an evolutionary algorithm that searches the program control flow graph for symbolic paths that traverse the minimum number of unsatisfiable branch conditions. Preliminary evaluation shows that EvoSE outperforms state-of-the-art SE search strategies when targeting specific code elements.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging

General Terms

Algorithms, Verification

Keywords

Test automation, search-based software testing, symbolic execution

1. INTRODUCTION

Test automation has the potential to drastically reduce the cost of quality assurance in software development. With this motivation, a great amount of research has been devoted in particular to the problem of generating test input values that thoroughly exercise a software system. Search Based Software Testing (SBST) and Symbolic Execution (SE) are two of the most popular approaches to generate test suites automatically [1].

The problem of finding an input that exercise a specific code element and the dual problem of proving that a code element is infeasible are well known undecidable problems. To avoid wasting resources towards infeasible goals, most recent SBST and SE approaches aim to maximize code coverage measures instead of focusing on specific code elements [4, 7]. EvoSE instead aims to generate test cases targeting specific code elements as this is required, for example, in security testing where one wishes to exploit a presumable vulnerability or test the validity of an assertion.

A number of goal oriented SE search strategies have been proposed to address the problems of testing software patches, improve code coverage and exploiting security vulnerabilities [12, 13, 14]. These approaches employ ad hoc heuristic to rank the program branches that should be expanded first in the next SE iterations with the goal of reaching quickly the target code elements. EvoSE instead implements a metaheuristic exploration of the program execution space thus overcoming the well known limitations of deterministic best-first search algorithms.

EvoSE follows the recent line of work that combines SBST and SE approaches to benefit from their complementary strengths and weaknesses [17, 3, 8, 5]. However, instead of searching in the numeric space of program inputs, EvoSE considers the combinatorial space of the program execution paths. Metaheuristic algorithms have been successfully applied in the context of combinatorial problems like the traveling salesman problem and software model checking [11, 9]. To the best of our knowledge, EvoSE is the first approach that investigates the use of evolutionary algorithms to guide the exploration of symbolic execution paths.

2. THE EVOSE APPROACH

EvoSE is a novel test generation approach that combines SBST and SE aiming to exercise specific program elements. The main departure from existing SBST approaches is the identification of a different search space. While classic SBST techniques perform a search in the input space of a program, EvoSE considers the space of the program execution paths that may lead to the target program element.

Classic SBST techniques try to minimize a fitness function that measures the distance between the concrete execution and the target code element, this is because most of the program inputs produce executions that do not reach the target code element. Instead EvoSE considers the program execution paths that reach the target code element in the program Control Flow Graph (CFG). The CFG is an approximation of the program behavior and also includes infeasible paths, that is, paths that cannot be executed under any program

A-TEST'15, August 30-31, 2015, Bergamo, Italy
© 2015 ACM. 978-1-4503-3813-4/15/08...$15.00
http://dx.doi.org/10.1145/2804322.2804325

input. In essence, *EvoSE implements a search in the CFG for feasible program paths* that reach the target code element.

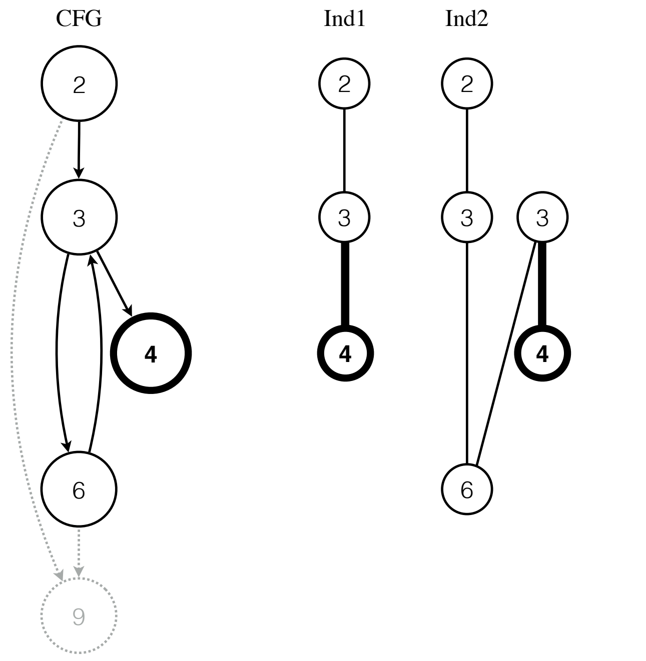
Evolutionary algorithms come in different forms, we designed EvoSE as a *memetic algorithm*, a metaheuristic algorithm that combines a classic Genetic Algorithm (GA) with systematic local optimization. In the following we describe the EvoSE design and in particular we focus on the problem representation, the fitness function, the crossover and mutation operators, and finally on the systematic local search algorithm. The description as well as the current implementation are limited to the intraprocedural case.

2.1 Problem Representation

EvoSE searches for feasible paths in the program CFG. The individuals that compose the evolving population are represented by variable-length lists of branches that are contiguous in the CFG graph. Each list begins with the program entry point and ends with the target program element. The initial population of candidate solutions is seeded performing random walks in the CFG.

```
1:  int s=read();
2:  for(int i=0; i < s; i++) {
3:     if(i > 100) {
4:        exit(ERROR);
5:     } else {
6:        ...
7:     }
8:  }
9:  exit();
```

(a) Example program



(b) CFG of the program in Figure 1a and two individuals of the GA population Ind1 and Ind2

PC1: `i=0 && i<s && i>100`
PC2: `i=0 && i<s && i<=100 && i+1<s && i+1>100`

(c) Path conditions of the GA individuals in Figure 1b

Figure 1: Example EvoSE problem

As an example, consider the code in Figure 1a and the corresponding CFG in Figure 1b. The CFG nodes are labeled by the corresponding line numbers in the code, dashed lines represents CFG portions that can be ignored as they cannot be part of a path reaching the target node 4. In this program, the only feasible path that reaches the target is one that enters the loop 100 times. The two infeasible paths encoded by the individuals Ind1 and Ind2, constitute the initial population of the GA, they enter the loop one time and two times respectively.

2.2 Fitness Function

It is well known that metaheuristic search techniques perform better when the search landscape is smooth, that is, when elements that are nearby in the search space have similar fitness values [6]. For this reason, state of the art SBST techniques employ fitness functions that combine two different metrics: *approach level* and *branch distance*. The *approach level* is a discrete distance value that is computed by counting the nodes that separate the exercised path and the target element in the program CFG. The *branch distance* is a smooth measure that, considering the branch with the smallest *approach level*, tells how far the execution went from taking the opposite side of the conditional (the interested reader can refer to [15] for more details). Similarly, the fitness function adopted in EvoSE is not a binary value that tells if the considered execution path is feasible or not, but is instead a continuous value obtained by performing a Dynamic Symbolic Execution (DSE) of the program along that path.

Classic SE allows to map a certain program execution path to a program input, if it exists, that produces an execution following the same path. This can be achieved automatically by solving the conditions that appear on the path with the help of an SMT solver. EvoSE uses a MaxSMT solver and can therefore obtain a smoother measure of how far the path is from being feasible. Given a list of constraints, MaxSMT solvers identify the largest subset of elementary constraints that is satisfiable. The fitness function is computed dynamically by executing the program along the desired path using as input the solution provided by the MaxSMT.

EvoSE guarantees that the dynamic symbolic execution follows the desired path by forcing the concrete evaluation of branch conditions in the style of execution hijacking [16]. The symbolic evaluation of branch conditions instead is computed as usual. EvoSE defines a *graded feasibility* measure by counting the number of branches that are not in the subset identified by MaxSMT, this measure is used to direct the search towards maximally satisfiable paths and eventually identify feasible execution paths. *Graded feasibility* can be regarded as a replacement for the *approach level* metric. EvoSE computes the classic *branch distance* measure for each unsatisfiable branch along the execution and normalizes it according to the formula proposed by Arcuri in [2].

In summary, given an execution path p, the fitness value $f(p)$ that needs to be minimized is the sum of the path's graded feasibility $grad\_feas(p)$ and the normalized branch distance $b\_dist(p, b_i)$ for all the branch conditions b_i that are not satisfied by the MaxSMT solution $maxsmt(p)$ to the symbolic constraints collected for path p:

$$f(p) = grad\_feas(p) + \sum\nolimits_{b_i \notin maxsmt(p)} norm(b\_dist(p, b_i)) \tag{1}$$

Consider the GA individuals Ind1 and Ind2 in Figure 1b. Both their respective path conditions PC1 and PC2 reported in Figure 1b are infeasible and their MaxSMT solutions contains all but the last branch condition (represented with a bold line). Both the individuals have therefore *graded feasibility* 1. The *branch distance* however is smalled for P2 because the value of i is 0 and thus the expression $i + 1$ is closer to 100 compared to i. This matches our intuition that an execution that gets closer to executing the loop 100 times should be favored in the population evolution.

The fitness function used in EvoSE, while being closely related to the classic branch distance, is computed for each unsatisfied condition along the path and not only for the branch that is closest to the target code element. For this reason, the genetic algorithm favors individuals that contain a large number of jointly-satisfiable branch conditions, independently of their position in the path. In light of Goldberg's *building block hypothesis*, we suggest that sequences of jointly-satisfiable branch conditions constitute the fundamental building blocks of the optimal, fully satisfiable, execution path.

2.3 Crossover Operator

Cut and splice is a typical choice as a one-point crossover operator for individuals of variable length like the paths in a graph. Given two individuals of the form `A|B` (list A followed by list B) and `C|D`, the new offsprings will have the form `A|D` and `C|B` respectively.

To guarantee that the newly generated offsprings encode existing paths in the CFG, EvoSE combines *cut and splice* with a repair strategy that uses the CFG to connect part `A` of the first individual with the longest possible postfix of part `D` of the second individual. In the example from Figure 1, the crossover operator could try to join the prefix [2,3] of Ind1 with the postfix [3,6,3,4] of Ind2. Such crossover produces the path [2,3,3,6,3,4] which is not a valid path and needs to be repaired.

The repair strategy would use the CFG to reconnect the two sub-paths and would produce the individual Ind3 that encodes the path [2,3,6,3,6,3,4]. If connecting the last branch in the prefix of Ind1 to the first branch in the postfix of Ind2 is not possible, the repair strategy would consider the next branch in the postfix of Ind2. Eventually the process will produce a valid individual as Ind1 and Ind2 share at least the last branch, that is, the target code element.

2.4 Mutation Operator

The EvoSE mutation operator selects randomly a branch in the individual and replaces it with the paired branch in the CFG. This produces an individual that encodes an execution path in which one branch condition evaluates differently respect to the original individual. Consider Ind3 from the previews paragraph, the mutation operator might decide for example to replace branch 4 with the paired branch 6 (the execution of branch 4 or 6 being the two possible outcomes of the evaluation of the condition at line 2).

As for the case of the crossover operator in Section 2.3, the individual obtained after the mutation might not encode a valid CFG path. This is the case in our example where the mutated path would end on branch 6 and not on the target branch 4. The repair strategy described earlier is finally used to reconnect branch 6 to branch 4 producing the valid individual Ind4 that encodes the path [2,3,6,3,6,3,6,3,4].

2.5 Local Search

GA can be very effective in finding solutions that approximate the global optimum but might miss some local optimization opportunity due to their stochastic nature. To overcome this limitation, *memetic algorithms* combine GA with deterministic local optimization [10].

For a given notion of neighborhood, a local search is performed by systematically evaluating all the neighbors of a candidate solution, retaining the individual with the best fitness. The process is repeated until no further improvement is possible. In our problem representation we could identify as neighbor of a given individual, any individual produced by flipping one of the conditions along the path, that is, any individual that can be obtained by a single application of the mutation operator defined in Section 2.4. This definition however produces a very large neighborhood that is impractical for a deterministic local search.

In EvoSE we included a local search strategy that visits all the neighbors that can be obtained by replacing one of the infeasible branches along the individual's execution path. We called this simple strategy *regret minimization* and we applied it as a fourth genetic operator operator after selection, crossover and mutation. Our preliminary evaluation showed that regret minimization is effective in improving the optimization process and does not affect negatively the EvoSE performance.

Consider again the individual Ind1 from Figure 1b. The only unsatisfied condition in the encoded path is the last one: `i>100`. Flipping this condition produces the individual Ind2 that has a better fitness then Ind1 as the only unsatisfied condition (i.e. `i+1>100`) leads to a smaller *branch distance.*

3. PRELIMINARY EVALUATION

We implemented EvoSE with the help of the open source evolutionary algorithms framework DEAP [1] and the symbolic execution engine CREST [2]. We implemented the evolutionary operators described in Section 2 using the DEAP infrastructure . We modified CREST to symbolically execute (possibly) infeasible paths using *execution hijacking*, integrate the MaxSMT solver Yices [3], and compute the fitness function from Equation 1.

```
extern char *curr_ptr;
GetKeyword(char *kw) {
  char word[KWDSLEN+1];
  char ch=getchar(curr_ptr);
  int i=0;
  while((isalnum(ch)||ch=='_') && i<KWDSLEN) {
    word[i++]=ch;
    ch = getchar(curr_ptr);
  }
  word[i]=0;
  if(strcmp(kw, word) == 0) {
    printf("TARGET!");
  }
}
```

Figure 2: A two-pass text parser.

[1] https://deap.readthedocs.org

[2] https://code.google.com/p/crest

[3] http://yices.csl.sri.com

We evaluated EvoSE on the function `GetKeyword` that implements a two-pass text parser. A simplified version of the parser code is in Figure 2. The function execution reaches the target branch at line 12 when the keyword pointed by the variable `kw` is found on the input buffer pointed by `curr_ptr`. The loop at lines 6—9 filters out the input characters that are not alphanumeric.

From our experience, it is very hard for classic SE to generate an input buffer that reaches line 12. This is because the decision at line 11 depends from which branch was traversed earlier at line 6. Only characters that are alphanumeric in fact, can match the alphanumeric keyword `kw`, this relation however cannot be observed by an analysis that explores paths independently. In EvoSE, paths that correspond to inputs with many alphanumeric characters will produce smaller fitness values due to the small branch distance at line 11 and will therefore be favored in the evolutionary process.

We compared EvoSE against the different SE search strategies implemented by CREST. We used keywords `kw` of variable lengths and limited the execution time to 60 minutes for each of the experiments. We found that EvoSE could discover keywords of length up to 50 characters while CREST could only identify keywords up to 4 characters long. The analysis of the results revealed that EvoSE needed to generate paths over 1000 branches long to reach the target line, such depth analysis of the symbolic execution space is normally considered out of reach. These first empirical results indicate that evolutionary algorithms can be effective in directing the exploration of SE paths towards target program elements.

4. CONCLUSION

EvoSE is a novel, goal-oriented test generation technique that combines SBST and SE. The core of the technique is a metaheuristic search strategy to efficiently explore program executions paths. Preliminary experimental results indicate that EvoSE can be more effective then classic SE strategies in producing test inputs that exercise specific code elements.

Future research directions include the extension of the genetic operators to the interprocedural case, the investigation of alternative population seeding strategies and a thorough evaluation of the EvoSE effectiveness, in particular for problems arising in the context of security testing.

5. REFERENCES

[1] S. Anand, E. K. Burke, T. Y. Chen, J. Clark, M. B. Cohen, W. Grieskamp, M. Harman, M. J. Harrold, and P. Mcminn. An orchestrated survey of methodologies for automated software test case generation. *J. of Systems and Software*, 86(8):1978–2001, 2013.

[2] A. Arcuri. It really does matter how you normalize the branch distance in search-based software testing. *Software Testing, Verification and Reliability*, 23(2):119–147, 2013.

[3] A. Baars, M. Harman, Y. Hassoun, K. Lakhotia, P. McMinn, P. Tonella, and T. Vos. Symbolic search-based testing. In *Automated Software Engineering (ASE), 2011 26th IEEE/ACM International Conference on*, pages 53–62, Nov 2011.

[4] M. Baluda, P. Braione, G. Denaro, and M. Pezzè. Enhancing structural software coverage by incrementally computing branch executability. *Software Quality Journal*, 19(4):725–751, 2011.

[5] P. Dinges and G. Agha. Solving complex path conditions through heuristic search on induced polytopes. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, pages 425–436, New York, NY, USA, 2014. ACM.

[6] A. E. Eiben and J. E. Smith. *Introduction to Evolutionary Computing*. SpringerVerlag, 2003.

[7] G. Fraser and A. Arcuri. Whole test suite generation. *IEEE Transactions on Software Engineering*, 39(2):276 –291, feb. 2013.

[8] J. P. Galeotti, G. Fraser, and A. Arcuri. Improving search-based test suite generation with dynamic symbolic execution. In *IEEE International Symposium on Software Reliability Engineering*, 2013.

[9] P. Godefroid and S. Khurshid. Exploring very large state spaces using genetic algorithms. In J.-P. Katoen and P. Stevens, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 2280 of *Lecture Notes in Computer Science*, pages 266–280. Springer Berlin Heidelberg, 2002.

[10] H. Ishibuchi, T. Yoshida, and T. Murata. Balance between genetic search and local search in memetic algorithms for multiobjective permutation flowshop scheduling. *Evolutionary Computation, IEEE Transactions on*, 7(2):204–223, April 2003.

[11] P. Larrañaga, C. Kuijpers, R. Murga, I. Inza, and S. Dizdarevic. Genetic algorithms for the travelling salesman problem: A review of representations and operators. *Artificial Intelligence Review*, 13(2):129–170, 1999.

[12] K.-K. Ma, K. Y. Phang, J. S. Foster, and M. Hicks. Directed symbolic execution. In *Proceedings of the 18th International Conference on Static Analysis*, SAS'11, pages 95–111, Berlin, Heidelberg, 2011. Springer-Verlag.

[13] P. D. Marinescu and C. Cadar. Katch: High-coverage testing of software patches. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, pages 235–245, New York, NY, USA, 2013. ACM.

[14] S. Sidiroglou-Douskos, E. Lahtinen, N. Rittenhouse, P. Piselli, F. Long, D. Kim, and M. Rinard. Targeted automatic integer overflow discovery using goal-directed conditional branch enforcement. *SIGPLAN Not.*, 50(4):473–486, Mar. 2015.

[15] N. Tracey, J. Clark, K. Mander, and J. McDermid. An automated framework for structural test-data generation. In *Proceedings of the 13th IEEE International Conference on Automated Software Engineering*, ASE '98, Washington, DC, USA, 1998. IEEE Computer Society.

[16] P. Tsankov, W. Jin, A. Orso, and S. Sinha. Execution hijacking: Improving dynamic analysis by flying off course. In *Proceedings of the 2011 Fourth IEEE International Conference on Software Testing, Verification and Validation*, ICST '11, pages 200–209, Washington, DC, USA, 2011. IEEE Computer Society.

[17] T. Xie, N. Tillmann, P. de Halleux, and W. Schulte. Fitness-guided path exploration in dynamic symbolic execution. In *Proceedings of the 39th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2009)*, pages 359–368, June-July 2009.