Combining Search-based and Constraint-based Testing

Jan Malburg
Saarland University – Computer Science
Saarbrücken, Germany
malburg@st.cs.uni-saarland.de

Gordon Fraser
Saarland University – Computer Science
Saarbrücken, Germany
fraser@cs.uni-saarland.de

***Abstract*—Many modern automated test generators are based on either meta-heuristic search techniques or use constraint solvers. Both approaches have their advantages, but they also have specific drawbacks: Search-based methods get stuck in local optima and degrade when the search landscape offers no guidance; constraint-based approaches, on the other hand, can only handle certain domains efficiently. In this paper we describe a method that *integrates both techniques* and delivers the best of both worlds. On a high-level view, our method uses a genetic algorithm to generate tests, but the twist is that during evolution a constraint solver is used to ensure that mutated offspring efficiently explores different control flow. Experiments on 20 case study examples show that on average the combination improves branch coverage by 28% over search-based techniques and by 13% over constraint-based techniques.**

I. Introduction

Meta-heuristic search and constraint-based techniques have emerged as two successful approaches to automatically generate test data that achieve high coverage: Search-based testing casts the testing problem as a search problem and applies efficient algorithms to find inputs that can serve as suitable tests; constraint-based techniques use static and dynamic symbolic execution (DSE) to precisely calculate the needed inputs. Both approaches have their strengths and weaknesses: Search-based testing scales well and can handle any code and test criterion, but only works well if a heuristic provides sufficient guidance. In the example in Figure 1, search based testing can easily generate test data to satisfy the first and the third branch condition, but the second branch is an instance of the flag problem [1], which gives the search no guidance. Constraint-based testing exploits the efficiency of modern constraint solvers which are not dependent on search heuristics, but there are limits to both scalability and the types of constraints that can be handled. In the example in Figure 1, the third branch contains a problematic non-linear constraint, and if the `Math` library is not available in source code or bytecode, then deriving constraints can be difficult in the first place.

There is a growing awareness that these are not competing techniques but offer great potential for combination. Previous work in this area uses search-based techniques for particular cases where constraint solvers are not good [2], or hooks together search-based tools for method sequence generation with dynamic symbolic execution tools [3]. In contrast to such approaches where the two techniques co-exist, we propose a technique that intrinsically combines search-based testing with constraint-based testing, thus overcoming the problems of the original techniques, and making it easy to handle all branches in Figure 1.

```
double example(int x, int y, double z) {
  boolean flag = y > 1000;
  // ...
  if(x + y == 1024)
    if(flag)
      if(Math.cos(z)−0.95 < Math.exp(z))
        // target branch
  // ...
}
```

Fig. 1. Constraint-based testing can easily and quickly derive test inputs for the first two conditions, but not for the third. Search-based testing might take longer on the first condition, but it can solve the third condition. However, it has problems with the second condition. A hybrid approach has no problems with any of the branches in this example.

Viewed from a high level, our approach looks like a search-based testing technique: A genetic algorithm (GA) evolves a population of candidate solutions, and a fitness function guides this search towards achieving a given coverage criterion. However, to avoid that the search gets stuck and to increase the speed with which the state space is explored, we add a special mutation operator: Rather than just flipping bits or blindly manipulating an input value, this new operator considers the path conditions that represent the execution path of a candidate solution, and *negates one of these path conditions*, just like dynamic symbolic execution does. A constraint solver then produces the new, mutated input, and this new input is guaranteed to take a different execution path, boosting the exploration aspect of the search.

The effects of this combination can be dramatic: Experiments with our Java PathFinder [4] based prototype show that compared to traditional search-based techniques, the number of generations necessary to find a solution is reduced by an order of magnitude. At the same time, the branch coverage achieved is significantly higher than that of traditional search-based testing and constraint-based testing.

II. Background

As testing is an essential task in software development, a large number of different techniques has been proposed over the last decades. In this paper, we focus on *white-box testing*, i.e., the task of generating suitable inputs for programs using the program's source code. Assuming the common case where there is no automated oracle available, the objective is to create a representative test suite satisfying a given coverage criterion. In this context, two main techniques have emerged recently as most successful, both allowing one to generate inputs that achieve a high level of coverage on almost any program: Constraint-based testing and search-based testing.

A. Constraint-based testing

Constraint-based testing generates test data by solving constraints produced by symbolic execution. For example, if the aim was to generate test inputs that evaluate the first if-expression in Figure 1 to true, then the constraint solver would calculate values for the inputs x and y that make the expression $x + y = 1024$ true. In classic symbolic execution, the tester has to select a path, and along this target path symbolic execution collects all such path conditions (e.g., `if` clauses) and operations on the *symbolic* input variables and state. A constraint solver can then derive inputs that make the program follow this path. In *dynamic* symbolic execution (DSE), exploration is started with a random value for which the program is executed. Along this execution path branching conditions are collected whenever they are evaluated, and a symbolic state is updated whenever values are changed. One of the collected path constraints is negated to describe a hitherto unexecuted path, and exploration continues on this new path. Whenever there are values or constraints that cannot be handled symbolically, one can revert to concrete values (therefore DSE is also known as *concolic* testing)

DSE has been implemented in tools like DART [5] or Microsoft's parametrized unit testing tool PEX [6]. Although efficient, these tools still are limited with respect to their scalability, and some domains such as non-linear or floating point arithmetics currently cannot be (efficiently) handled by constraint solvers. For example, the third if-expression in Figure 1 is non-linear, and in addition the calls to the library methods of the `Math` package might not be available in source code or bytecode to allow symbolic execution.

B. Search-based testing

An example of a meta-heuristic search technique as used in search-based testing [7] is a genetic algorithm, where a population of candidate solutions (i.e., potential test cases) is evolved towards satisfying any chosen coverage criterion. The search is guided by a fitness function that estimates how close a candidate solution is to satisfying a coverage goal. A fitness function guides the search in choosing individuals for reproduction, gradually improving the fitness values with each generation until a solution is found. For example, to generate tests for branch coverage a common fitness function [7] integrates the *approach-level* (number of unsatisfied control dependencies) and the *branch distance* (estimation of how close the deviating condition is to evaluating as desired).

The success of search-based testing depends on the availability of appropriate fitness functions that guide towards an optimal solution. In practice, the search landscapes described by these fitness functions often contain local optima, i.e., candidate solutions that have better fitness than their neighbors but are not globally optimal, thus inhibiting exploration. Another problem are plateaux in the search landscape, where individuals have the same fitness as their neighborhood, which lets the search degrade to random search. For example, the second if-expression in Figure 1 offers no guidance for the search – the Boolean flag can only be true or false, which means the branch distance can only either be 1 or 0.

III. Hybrid Search- and Constraint-based Testing

To overcome the drawbacks of both, search-based and constraint-based testing, we combine the two techniques. This combined approach to a large extent works like a standard GA. First, an initial population of candidate solutions is generated randomly for the program under test. This population is evolved using both, standard search operators and a new DSE based operator: After selection, individuals are crossed over with a certain probability, and mutated using standard mutation operators such as bit flipping. With a certain probability, however, the new DSE based mutation operator is used. This new mutation operator is basically a single step of DSE: The path conditions for the considered individual are collected using DSE, and then one condition is selected randomly. A new constraint system is created, consisting of the path conditions that lead to the selected branch, conjoined with the negation of the selected path condition. This constraint set is passed to a constraint solver, and a solution to the constraint system represents a mutated individual which follows a different execution path than the original individual. If the constraint solver fails to find a satisfying assignment, then standard mutation is used as fallback. After determining the fitness values of the new population, the GA continues iterating until either a solution has been found or some other stopping condition is met.

This overcomes the individual problems of search-based testing and constraint-based testing: Values for constraints a solver cannot handle are derived evolutionary. For example, for non-linear constraints or constraints on function calls which cannot be instrumented DSE traditionally uses randomly chosen concrete values — in our approach, these concrete values are also used in the constraint system, but they are not just randomly chosen but optimized by the search. On the other hand, the search can easily escape local optima or plateaux, as mutation can change execution paths. Important issues in search-based testing such as the flag problem or the problem of nested predicates [1] can easily be overcome, and the exploration of the state space is boosted.

As an example for the new mutation operator, consider a test case $x = 1024, y = 0, z = 0.0$ for our example function (Figure 1). Regular mutation of this test case might add or subtract small values to x, y, or z, replace them with random values, or flip bits in a bitwise representation. To reach the target branch, these mutation steps would need to gradually get the y value of the test case closer to 1000. In the case of 32-bit Integers, such a process can take a while — in the case of a Boolean flag as in the example, such a process might only succeed by chance. The new mutation operator would take the path condition $x + y > 1024 \wedge \neg(y > 1000)$, select one constraint to negate, and derive a new condition to solve such as $x+y > 1024 \wedge y > 1000$. A solution to the former constraint might be $x = 0, y = 1024, z = 0.0$, which is one branch closer to the target branch than the original test case. On the third branch condition, this DSE based mutation operator would fail just like DSE because of the limitations of constraint solvers, but the normal mutation operators would still optimize the value of z towards satisfying the condition, finally reaching the target branch.

IV. Evaluation

To evaluate the effectiveness of the hybrid approach, we have implemented a prototype and evaluated it on a set of standard benchmark examples, comparing random search (RA), GA, DSE, and our hybrid approach (GA-DSE) in terms of achieved coverage.

Our prototype is based on Java PathFinder (JPF [4]). Our GA implementation uses whole test suite generation [8], where a chromosome is a set of test cases, and each test case is a bitvector representing the input parameters. The fitness function aims to maximize the number of covered branches, while minimizing the branch distances (see Section II-B) of all branches in the program. In addition, the fitness function keeps track of whether a predicate in the source code has been executed often enough such that all branches can be covered; i.e., each branch predicate needs to be executed twice.

For DSE, we have implemented a technique to collect path conditions with JPF similar to the technique implemented in Symbolic JPF [9]. To solve constraints we used the open source constraint solver Choco [10]. The exploration strategy follows the state of the art according to the literature (e.g., [6]): It starts with a single random test case and iteratively adds new test cases by negating randomly selected path conditions.

The GA used in the experiments is identical to GA-DSE, except that the probability of choosing the new mutation operator is 0%. Random search is implemented by randomly generating new test cases, and if such a new test case improves the coverage it is added to the output test suite.

For the evaluation, we ran GA-DSE, the GA, DSE, and random search on each of a set of 20 case study subjects taken from the literature (see Table I). For GA-DSE, we set the probability for choosing traditional vs. the new mutation operator to 50%. The examples consist mainly of linear constraints; Gammq, Fisher, Bessj, Expint, and ASW have constraints involving floating point numbers, as well as some non-linear constraints involving a square root, logarithm, bit-operations, and a quadratic function. We chose a limit of 50,000 test executions for the random search, the genetic algorithm, and GA-DSE, as well as 50,000 constraint solvings for DSE. We empirically determined a timeout value of 500ms for the constraint solver as sufficient for most constraints. The GA and GA-DSE used a population size of 10 test suites of each three to five test cases initially. As randomized algorithms can produce different results in different runs, we repeated each run 25 times with different random seeds.

TABLE I
Branches and lines of code in the case study subjects, and achieved coverage using RA, GA, DSE, and GA-DSE. Highest coverage values are highlighted with bold text.

Case Study	Branches	LOC[1]	Branch Coverage			
			RA	GA	DSE	GA-DSE
IntAVLTreeMap [11]	128	330	0.63	0.64	**0.94**	0.93
BinHeap [12]	162	444	0.73	0.76	**0.96**	**0.96**
BinTree [12]	94	201	0.85	0.85	**1.00**	**1.00**
FibHeap [12]	210	474	0.89	0.90	**0.94**	**0.94**
TreeMap [12]	260	638	0.48	0.54	0.71	**0.89**
WBS [13]	90	170	0.40	0.53	**0.69**	**0.69**
FGS [13]	1096	881	0.39	0.47	0.64	**0.66**
HeapArray [11]	64	167	0.79	0.81	**0.95**	**0.95**
IntRedBlackTree [11]	168	356	0.46	0.49	0.88	**0.91**
Remainder [14]	24	32	0.93	0.92	0.95	**0.96**
AVLTree [15]	136	533	0.07	0.30	**1.00**	**1.00**
NodeCachingLL [15]	126	382	0.83	0.83	**0.97**	**0.97**
SinglyLinkedList [15]	108	181	0.81	0.83	**0.94**	**0.94**
TreeSet [15]	158	349	0.38	0.49	0.86	**0.89**
Fisher [16]	22	54	**0.56**	**0.56**	0.44	**0.56**
Gammq [17]	26	70	0.80	0.80	0.30	**0.84**
Bessj [17]	28	79	0.72	0.86	0.55	**0.96**
Expint [17]	30	50	0.58	0.75	0.45	**0.96**
ASW [13]	98	308	**0.85**	**0.85**	0.65	**0.85**
TCAS [18]	74	98	0.57	0.63	**0.69**	**0.69**
Σ / $\oslash$	3158	5917	0.64	0.69	0.79	**0.89**

Table I lists the achieved average branch coverage values. The superiority of GA-DSE over random search and the GA is striking, with GA-DSE achieving higher coverage on all examples but Fisher and ASW, where it achieves the same coverage (higher coverage is significant at level 0.05 for 18 cases, measured using a Mann-Whitney U test). Compared to DSE, for nine out of the 20 examples GA-DSE achieves significantly higher coverage (significant at level 0.05 using a Mann-Whitney U test). Little surprising, these nine examples include the five with floating point numbers. For the remaining 11 examples, both DSE and GA-DSE achieve the same coverage, i.e., there is no statistically significant difference. For AVLTree and BinTree both achieve 100% coverage, and for the remaining nine examples it seems that both algorithms cover all feasible branches, i.e., higher coverage is not possible. On the whole, this confirms our expectation that by combining GA with DSE in the GA-DSE algorithm one achieves significantly higher coverage than its constituents GA and DSE as well as random testing.

GA-DSE achieves significantly higher branch coverage than random search, a GA, or DSE.

[1]LOC stands for non-commenting lines of source code, calculated with JavaNCSS (http://javancss.codehaus.org/)

V. Related Work

Inkumsah and Xie [3] proposed a combination of an existing evolutionary testing tool with a DSE tool. In contrast to our approach, neither of the techniques itself is refined, and the combination is serial and not alternating like GA-DSE, which means nested predicates containing problematic constraints for both search and constraint-solving cannot be overcome. Lakhotia et al. [2] introduced FloPSy, which extended the DSE tool PEX to use a search-based approach to solve floating point constraints. In contrast to our approach it is a combination of two techniques for a specific setting (floating point constraints) rather than a general approach. Furthermore, FloPSy performs the search in order to solve constraints; when constraints are not available because of native code, then such an approach cannot help. In addition, FloPSy is an improvement to DSE, whereas GA-DSE can be seen as a contribution to improve both search-based as well as constraint-based techniques.

Majumdar and Sen [19] interleaved DSE with random search. DSE is used to provide an exhaustive local search, while random search is used to address the state problem and explore more diverse parts of the state space. In contrast to GA-DSE, in this combination the search is local, and only one of the two underlying techniques is used at a time.

VI. Conclusions

Constraint solving and meta-heuristic search techniques are both successfully applied to generate test cases, but both approaches have their advantages and disadvantages. In practice, this means that while one technique might work well on a particular problem, it might be inferior on another problem, and it is difficult to predict how a technique will perform for a given problem. To overcome this issue, in this paper we presented a novel combination of evolutionary search and dynamic symbolic execution: Interpreted from the viewpoint of a search-algorithm, our solution uses DSE to improve exploration and to overcome problematic areas in the search landscape. Interpreted from the viewpoint of DSE, a search-wrapper controls the DSE exploration and handles those cases where a constraint solver fails.

Our experiments have shown that the combination achieves higher coverage than pure search requiring less iterations, and also same or higher coverage than DSE. If one of the underlying techniques performs particularly bad, then this can easily be compensated given enough iterations. At the same time, the use of search techniques allows us to easily change the search objective: For example, to generate test cases not for branch coverage but for other coverage criteria or any other objective, all that is required is to change the fitness function. Furthermore, it is easy to integrate secondary objectives such as the test suite size.

Acknowledgments. Gordon Fraser is funded by the Cluster of Excellence on Multimodal Computing and Interaction at Saarland University, Germany. We thank Eva May, Jeremias Rößler, and Andrea Arcuri for feedback on earlier versions of this paper.

References

[1] M. Harman, L. Hu, R. Hierons, J. Wegener, H. Sthamer, A. Baresel, and M. Roper, "Testability transformation," *IEEE Trans. Softw. Eng.*, vol. 30, pp. 3–16, January 2004.

[2] K. Lakhotia, N. Tillmann, M. Harman, and J. de Halleux, "FloPSy - search-based floating point constraint solving for symbolic execution," in *22nd IFIP International Conference on Testing Software and Systems*, ser. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2010, pp. 142–157.

[3] K. Inkumsah and T. Xie, "Improving structural testing of object-oriented programs via integrating evolutionary testing and symbolic execution," in *Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering (ASE'08)*. Washington, DC, USA: IEEE Computer Society, 2008, pp. 297–306.

[4] W. Visser, K. Havelund, G. Brat, and S. Park, "Model checking programs," in *Proceedings of the 15th IEEE International Conference on Automated Software Engineering*, ser. ASE '00. Washington, DC, USA: IEEE Computer Society, 2000.

[5] P. Godefroid, N. Klarlund, and K. Sen, "DART: directed automated random testing," in *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'05)*. New York, NY, USA: ACM, 2005, pp. 213–223.

[6] N. Tillmann and J. N. de Halleux, "Pex — white box test generation for .NET," in *International Conference on Tests And Proofs (TAP'08)*, ser. LNCS, vol. 4966. Springer, 2008, pp. 134 – 253.

[7] P. McMinn, "Search-based software test data generation: a survey: Research articles," *Software Testing Verification Reliability*, vol. 14, no. 2, pp. 105–156, 2004.

[8] G. Fraser and A. Arcuri, "Evolutionary generation of whole test suites," in *Proceedings of the 11th International Conference On Quality Software (QSIC'11)*. IEEE Computer Society, 2011, pp. 31–40.

[9] S. Anand, C. S. Păsăreanu, and W. Visser, "JPF-SE: A symbolic execution extension to Java PathFinder," in *Proceedings of the 13th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'07)*. Berlin, Heidelberg: Springer-Verlag, 2007, pp. 134–138.

[10] Choco Team, "Choco: an open source java constraint programming library," Ecole des Mines de Nantes, Research report 10-02-INFO, 2010. [Online]. Available: http://www.emn.fr/z-info/choco-solver/pdf/choco-presentation.pdf

[11] R. Sharma, M. Gligoric, V. Jagannath, and D. Marinov, "A comparison of constraint-based and sequence-based generation of complex input data structures," in *Proceedings of the 3rd International Conference on Software Testing, Verification, and Validation Workshops*. Washington, DC, USA: IEEE Computer Society, 2010, pp. 337–342.

[12] W. Visser, C. S. Păsăreanu, and R. Pelánek, "Test input generation for Java containers using state matching," in *Proceedings of the 2006 International Symposium on Software Testing and Analysis (ISSTA'06)*. New York, NY, USA: ACM, 2006, pp. 37–48.

[13] M. Staats and C. Păsăreanu, "Parallel symbolic execution for structural test generation," in *Proceedings of the 19th International Symposium on Software Testing and Analysis (ISSTA'10)*. New York, NY, USA: ACM, 2010, pp. 183–194.

[14] H. Sthamer, "The automatic generation of software test data using genetic algorithms," Ph.D. dissertation, University of Glamorgan, Pontyprid, Wales, UK, April 1996.

[15] J. P. Galeotti, N. Rosner, C. G. López Pombo, and M. F. Frias, "Analysis of invariants for efficient bounded verification," in *Proceedings of the 19th International Symposium on Software Testing and Analysis (ISSTA'10)*. New York, NY, USA: ACM, 2010, pp. 25–36.

[16] E. Dorrer, "F-distribution," *Commun. ACM*, vol. 11, no. 2, pp. 116–117, 1968.

[17] C. Schneckenburger and J. Mayer, "Towards the determination of typical failure patterns," in *4th International Workshop on Software Quality Assurance, co-located with ESEC/FSE'07 (SOQUA'07)*. New York, NY, USA: ACM, 2007, pp. 90–93.

[18] H. Do, S. Elbaum, and G. Rothermel, "Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact," *Empirical Software Engineering*, vol. 10, pp. 405–435, October 2005.

[19] R. Majumdar and K. Sen, "Hybrid concolic testing," in *Proceedings of the 29th International Conference on Software Engineering (ICSE'07)*. Washington, DC, USA: IEEE Computer Society, 2007, pp. 416–426.