Summarizing the Development in Search Based Testing and Symbolic Execution for Test Case Generation

Shruti Gandhi

Department of Electrical and Computer Engineering
North Carolina State University
Raleigh, NC 27606, USA

sgandhi3@ncsu.edu

***Abstract* - Software testing is indispensable for any successful software development process. A software test consists of an input that executes a program and verifies the obtained output against the expected output. Many techniques for automatic generation of inputs have been proposed over the years. Most popular of these techniques have been based on either search based methods or symbolic execution or a combination of the two. These techniques have also been modified to develop frameworks to handle specific problems that satisfy a certain coverage criterion. This paper tracks the progress in this area in terms of search based testing and symbolic execution, and how both these techniques have been evolved in parallel to be combined as single technique, which has been further expanded to encompass whole test suite generation.**

Index Terms – search based testing, symbolic execution, dynamic symbolic execution, genetic algorithms, branch coverage

I. Introduction

The most successful approaches to automatically generate test data achieving high test coverage are usually based on meta-heuristic search or constraint solvers. While search based methods approach the testing problem as a search problem, constraint based methods use static or dynamic symbolic execution

A. *Motivation*

In recent times search based software engineering has become an epicenter for research in the field of software engineering. Consequently, numerous software testing techniques are being proposed for automating test case generation. But there is no comprehensive delineation of which testing technique is best suited in which context, and which technique have become redundant due to emergence of advanced testing techniques. Thus, this paper aims to provide a comprehensive guideline to enable the correct application for the most popular search based and constraint based testing techniques.

B. *Checklist*

This paper makes the following contributions -

1) Introduce the general concepts behind –
    a. Symbolic Execution
    b. Search Based Testing
    c. Integrated Symbolic Execution and Search Based Testing
    d. Test Suite Optimization
    e. Tools based on the above
2) Historical Context
3) Tracking Advancement
4) Description of the various evaluation strategies that have been used to assess the testing techniques
5) Conclusion
6) Recommendations for future work

II. Background

A. *Symbolic Execution*

The key idea behind symbolic execution is to generalize testing by using unknown symbolic variables in evaluation. A symbolic executor collects all path conditions (e.g. if clauses) and operations on symbolic variables along the path selected by a tester. Then these conditions are fed to a constraint solver to derive all inputs that make the program follow this path, as done in classic constraint based testing. On the other hand, a dynamic symbolic executor evaluates and collects path branching conditions, updates symbolic states whenever they are changed, this is done along a path selected by executing a program for a random value. One of the collected path constraints is negated to describe an unexecuted path, and exploration is continued along that path. [1] Dynamic Symbolic Execution which is a combination of symbolic and concrete execution uses concrete execution to drive the symbolic exploration of a program, the runtime values produced by symbolic execution are used to simplify path constraints to make them more feasible for constraint solving. [4]

B. *Search Based Testing*

Search based testing defines the testing problem as a search problem and applies efficient algorithms to find inputs that can serve as tests [1]. Here an optimization algorithm is guided by an objective function which is defined in terms of a test adequacy criterion to generate test data. An example of a meta-heuristic search technique as used in search-based testing

[11] 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 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.

C. *Integrated Symbolic Execution and Search Based Testing*

The most successful approaches to automatically generate test data achieving high test coverage are usually based on meta-heuristic search or constraint solvers. While the search based testing is scalable, and can work with any code and test criterions, its' success depends on the availability of suitable fitness functions that can guide the search to an optimal solution. The presence of local optima/ plateaux in the search landscape described by the fitness function can lead to getting stuck in local optima/degrade to random search, respectively. Similarly, constraint based testing that uses dynamic symbolic execution to generate constraints, while efficient, has limited scalability as there are certain domain of constraints it cannot handle such as non-linear or floating point arithmetics. These open issues called for the need of developing a testing method that surpassed the given methods. Work done in [1] [3] [5] leverages search based testing methods and constraint solving methods by integrating them to solve these problems.

D. *Test Suite Optimization*

While generating tests from the source code satisfying a certain coverage criteria, the fundamental assumptions are that all coverage goals are equally important, equally difficult to reach and independent of each other. This flawed assumption leads to the general approach that works along devising a test case that exercises a particular coverage goal at a time, ignoring the possibility of a coverage goal being infeasible, or difficult to satisfy or causing collateral coverage. These problems cannot be efficiently predicted. To mitigate this problem instead of tackling individual coverage goals with distinct test cases, whole test suite optimization was proposed [2] [8] [10] that suggests to optimize the entire test suite towards satisfying a coverage criteria.

E. *Tools in Use*

Symbolic execution tools such as JPF [16] and CUTE/jCUTE [14] explore paths in the program under test symbolically and collect symbolic constraints at all branching points of an explored path. The collected constraints are solved if feasible, and a solution is used to generate a test that forces the execution of the program under test along the path. This process is repeated until all feasible paths have been explored or the number of explored feasible paths has reached the user-specified bound. Bounded exhaustive testing tools such as JPF[16], Rostra [21], and Symstra [22] generate exhaustive method sequences up to a small bound (with some pruning based on state equivalence or subsumption). However, sometimes covering some branches requires long method sequences whose length is beyond the low bound that can be handled by these tools.

Some evolutionary testing tools such as eToc [19] represent initial randomly generated method sequences as a population of individuals and evolve this population by mutating its individuals until a desirable set of method sequences is found.

However, because these evolutionary testing tools do not use program structure or semantic knowledge to directly guide test generation, they cannot provide effective support for generating desirable primitive method arguments even if the right method sequence skeleton is generated.

II. Historical Context (2008 – 11)

The historical context for this literature review is divided in three parts, 2.1 describes development in testing techniques based on symbolic execution, 2.2 describes development in testing techniques based on search based methods, 2.3 describes testing techniques based on the combination of the two approaches. In 2.4 we describe the emergence of whole test suite generation.

2.1 *Symbolic Execution*

Pex [6] which performs path-bounded model-checking was one of the first tools proposed in 2008 that used dynamic symbolic execution for .NET programs. Dynamic symbolic execution was first suggested in DART [11] was used on C programs and tracked linear integer arithmetic constraints. CUTE [15] was proposed in 2006 which also took into consideration pointer aliasing constraints. jCUTE [14] implemented CUTE for java. Another implementation of C source code

based dynamic symbolic execution was EXE [13] proposed in 2006 which implemented a number of further improvements, including constraint caching, independent constraint optimization,

bitvector arithmetic, and tracking indirect memory accesses symbolically. As against all the earlier tools Pex (reviewed in

Paper 6) was that instead of being specialized for a particular source language, and only including certain operations in the symbolic analysis Pex is language independent, and it can symbolically reason about pointer arithmetic as well as constraints from object oriented programs. It also improved upon the search order which was not prioritized in earlier tools to achieve high coverage quickly, which forced the user to precisely define bounds on the size of the program inputs and to perform an exhaustive search. Pex did this by developing search strategies that aimed at achieving high coverage fast without much user annotations. Its contemporary tools JPF[16] and XRT[17] which like Pex operated on managed programs (Java and .NET) but they used *static* symbolic execution, and they could not deal with stateful environment interactions.

It is well known that symbolic execution achieves high structural coverage for automatic test generation and the paths followed during symbolic execution form a symbolic execution tree, which represent all the possible executions through the program. However the difficulty to explore all possible program execution as the symbolic execution tree can become infinitely large in size. This limits the application of symbolic execution as it is vulnerable to scalability issues. Motivated by the high availability of multi-core computers, and the parallelizable nature of symbolic execution, in [4] Matt Staats and Corina Păsăreanu in 2011 proposed to mitigate this problem by parallelizing symbolic execution such that the essence of parallelization is maintained by eliminating synchronization overhead by designing approaches that required minimum inter process communication.

Pex developed in 2008 is a direct application of dynamic symbolic execution which brings in the limitation of scalability inherent in dynamic symbolic execution. The scalability issue is dealt head-on in [4] where the authors parallelize symbolic execution by statically partitioning a symbolic execution tree and distributing the partitions across parallel instances. This is an improvement over an underlying methodology used in Pex and its contemporary tools which are based on symbolic execution and dynamic symbolic execution.

2.2 *Search Based Testing*

Search based testing having got the kind of attention it had, there were some very fundamental questions still unanswered. Before 2010, there was no theoretical explanation for why and where evolutionary approaches work. Earlier studies compared Evolutionary Testing with other metaheuristic search methods and reported results on small numbers of programs, each with limited complexity. But in [6] on the other hand the authors not only predicted the suitability of Evolutionary Testing for structural test data generation problems using Royal Road and Schema Theory for Evolutionary Testing, but also did a number of empirical validations such as empirically validating that Evolutionary Testing performs well for Royal Road functions and this is due to the presence of the crossover operator, empirically assessing the performance of Evolutionary Testing compared to Hill Climbing and Random Testing; and empirically assessing Hybrid Memetic Algorithm which incorporates Hill Climbing into Evolutionary Testing.
Through [6], the authors lay the foundation for the work that has been done till now in Search Based Testing by clearly outlining the problems where evolutionary testing is best fit, and where hill climbing and random testing fit in the picture of Search Based Testing. This foundation helps in development of the context for combining search based methodologies esp. evolutionary testing with other testing techniques.

2.3 *Integrated Symbolic Execution and Search Based Testing*

In 2008 in [3], Kobi Inkumsah and Tao Xie proposed EVACON that leveraged evolutionary testing and symbolic execution to generate tests that included both suitable method arguments and suitable method sequences. In 2005, DART [11] used dynamic analysis of the program behavior during random testing to generate new test data to direct program execution along alternative program paths. While Randoop [18] proposed in 2007 used user-specified properties in conjunction with execution feedback to produce both fault-revealing and regression tests. By leveraging both symbolic execution and evolutionary testing, EVACON technique both generated test data to exercise different branches within the program under test and ensured optimal combination of method sequences. The figure below shows the overview of the EVACON framework, including four components: evolutionary testing, symbolic execution, argument transformation (for bridging from evolutionary testing to symbolic execution), and chromosome construction (for bridging from symbolic execution to evolutionary testing).

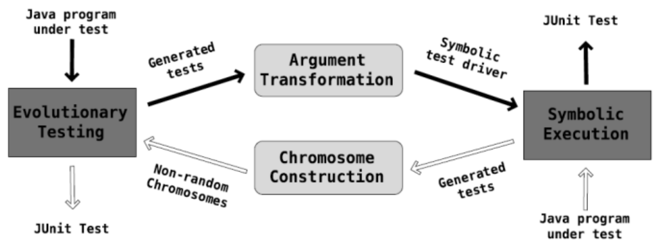

Figure 1: EVACON framework

In 2010, work done in [3] by Kobi Inkumsah and Tao Xie was extended by Lakhotia et.al. in [4]. While in Inkumsah and Xie [3] were the first authors to propose a framework (EVACON) combining evolutionary testing with DSE. Their framework targeted test data generation for object oriented code written in JAVA. They use two existing tools, eToc[19], an evolutionary test data generation tool, and jCUTE [14], a DSE tool. eToc constructs method sequences to put a class containing the method under test, as well as non–primitive arguments, into specific desirable states. jCUTE was then used to explore the state space around the points reached by eToc. But this approach did not address the problem of floating point,

computation in DSE which is an inherent limitation of the constraint solvers. Thus Lakhotia et.al. in [4] provided
a general framework for handling constraints over floating point variablesand were the first to propose a combination of DSE with SBST in order to improve code coverage in the presence of floating point computations. This work was further extended to be included as a plugin for Pex [6].
In 2011, Malburg and Fraser [1] proposed a technique that combined genetic algorithm and dynamic symbolic execution which in contrast to the technique proposed by Inkumsah and Xie [3] was an alternating combination of GA and DSE. Authors in [3] proposed a combination of an existing evolutionary testing tool with a DSE tool, neither of the techniques itself were refined, and the combination was serial which meant nested predicates containing problematic constraints for both search and constraint-solving could not have been overcome. Lakhotia et al. [4] in FloPSy, extended the DSE tool PEX to use a search-based approach to solve floating point constraints which was a combination of two techniques for a specific setting (floating point constraints) rather than a general approach. Furthermore, FloPSy performed the search in order to solve constraints; when constraints were not available because of native code, then such an approach was useless. Also, FloPSy was an improvement to DSE, hence the authors in [1] proposed GA-DSE which was an improvement over both search based and constraint based techniques.

2.4 *Test Suite Optimization*

In 2011 Fraser and Arcuri proposed a framework EVOSUITE [2] which formed the basis for [1]. While generating tests from the source code satisfying a certain coverage criteria, the fundamental assumptions are that all coverage goals are equally important, equally difficult to reach and independent of each other. This flawed assumption motivated the authors to challenge the general approach that worked along devising a test case that exercises a particular coverage goal at a time, ignoring the possibility of a coverage goal being infeasible, or difficult to satisfy or causing collateral coverage. These problems could not be efficiently predicted, which led the authors to propose an approach to mitigate them. Instead of tackling individual coverage goals with distinct test cases, the authors suggested an approach to optimize the entire test suite towards satisfying a coverage criteria.
The goal of the authors was to target difficult faults for which automated oracles were not available (which is a common situation in practice). Because in these cases the outputs of the test cases have to be manually verified, then the generated test suites should be of manageable size. There were two contrasting objectives: the “quality” of the test suite (e.g., measured in its ability to trigger failures once manual oracles are provided) and its size. The approach the authors followed in this paper was to satisfy the chosen coverage criterion (e.g., branch coverage) with the smallest possible test suite.
Malburg and Fraser [1] used whole suite test generation [2] for GA implementation in a hybrid approach integrating search based testing with constraint based testing. The Figure below illustrates the main steps in EVOSUITE: It starts by randomly generating a set of initial test suites, which are evolved using evolutionary search towards satisfying a chosen coverage criterion. At the end, the best resulting test suite is minimized, resulting in a test suite.

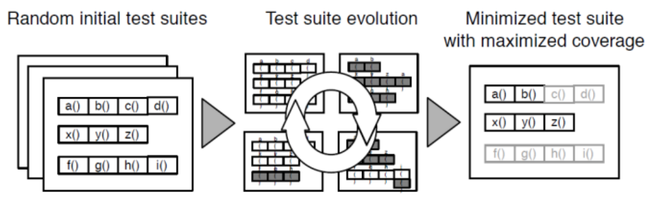
Random initial test suites
Test suite evolution
Minimized test suite with maximized coverage
a()
b()
c()
d()
x()
y()
z()
f()
g()
h()
i()

Figure 2: The EVOSUITE process: A set of randomly generated test suites is evolved to maximize coverage, and the best result is minimized.

III. TRACKING ADVANCEMENT (2011 – 15)

All the work reviewed till now, was establishing the groundwork for the work done after 2011, most of which has culminated into better test suite optimization approaches, and improvement in evaluation strategies undertaken till 2011.
Tracking the advancement for this literature review is also divided in three parts, 3.1 describes the emergence of whole test suite generation, and in 3.2 we describe the evolution in evaluation techniques.

3.1 *Test Suite Optimization*

In 2012, Gross et al. proposed an approach EXSYST [8] which extended the EVOSUITE tool proposed by Fraser and Arcuri [2], which performs search at the API level; EXSYST shares the fitness function with EVOSUITE, but uses its own representation and search operators. The authors extended EVOSUITE because of two basic shortcomings which limit test case generation tools to have widespread usage. One such shortcoming identified by the authors was the *oracle problem* which occurs when test case generators do not use oracles to assess the test cases thereby only generating executions and not the test cases. The missing oracle results in failures in programs going unnoticed. Another problem is that of *infeasibility* of the generated test cases which gives false failures and thus prohibit the widespread use of test case generation tools. These problems motivated the authors to use GUIs as filters against infeasible executions, and generate GUI events by using a search based approach to avoid the limitations of using system interface for test case generation.
EXSYST is also an improvement over Randoop [18] which though can generate a large numbers of test cases in short time, is based on the unrealistic assumption that the tester will manually generate oracles for large sets of randomly generated test cases.

Constraint-based testing interprets program paths as constraint systems using dynamic symbolic execution, and applies constraint solvers to derive new program inputs, and has been successfully implemented in tools like DART [11] and Microsoft's parameterized unit testing tool
PEX [6]. Although EXSYST uses a search-based approach, the authors claim that in principle also approaches based on dynamic symbolic execution could be used for GUI testing.

In 2013 Fraser and Arcuri [10] further extended their EVOSUITE tool in several directions by, using a much larger and more varied case study, verifying that the presence of infeasible branches has no negative impact on performance, and by providing theoretical analyses to shed more light on the properties of the proposed approach. The authors demonstrate the effectiveness of EVOSUITE by applying it to 1,741 classes coming from open source libraries and an industrial case to evaluate this search-based testing of object-oriented software EVOSUITE. The authors show strong statistical evidence that the EVOSUITE approach yields significantly better results (i.e., either higher coverage or, if the same coverage, then smaller test suites) compared to the traditional approach of targeting each testing goal independently.

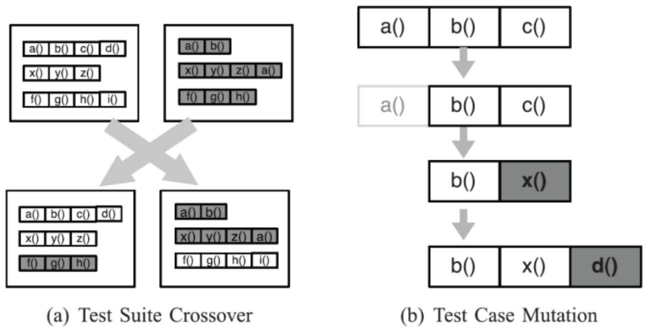

Figure 3: Crossover and mutation are the basic operators for the search using a GA. Crossover is applied at test suite level; mutation is applied to test cases and test suites.

3.2 *Evaluation Techniques*

Over the years, a lot of complex techniques for automated testing of software have been proposed, one problem faced when proposing these techniques is the difficulty to mathematically prove their effectiveness, this is where empirical analysis comes into picture. One of the biggest challenge in empirical testing is the choice of the case study. Currently this choice is not made in a systematic way, i.e., researchers choose software artifacts without providing any specific and unbiased motivation. Even a manual choice of software artifacts when choosing case studies for test data generation for open source software can introduce bias in the results. The absence of of any empirical study in the literature addressing threats to external validity focussing on possible biases due to case study selection motivated the authors to conduct an empirical study with statistical sound selection case studies for open source software.

In 2012, Fraser and Arcuri [9], the authors conducted a survey of the literature on test generation for object-oriented software to get a better picture of the current practice in evaluations in software engineering research. The authors explicitly listed the number the number of container classes from among all the classes considered as container classes represent a particular type of classes that avoids many problems such as environment interaction and can result in high coverage by even simple random testing. Out of the 44 evaluations considered in the literature survey, 17 papers were found to exclusively focus on container classes, 29 selected their case study programs from open source programs, while only six evaluations included industrial code, 17 evaluations used artificially created examples, either by generating them or by reusing them from the literature. Also, it was found that not a single paper out of those considered justifies why those particular set of classes was selected, and how this selection was done.
Thus, to select an unbiased sample of Java software, the authors decided to go with SourceForge open source development platform, for it was a dominant site of the type having more than 300,000 registered projects at the time of their experiment. Since there were 48,109 Java programming projects at the site at the time, it wasn't possible to apply EVOSUITE (a tool which automatically generates test suites for Java classes, targeting branch coverage - Read Review 2 for details) to all the projects in reasonable time, hence random sampling of data set was performed. For each chosen project the most recent sources from the corresponding source repository were downloaded to build the program. the issues faced here were - empty projects, misclassified projects, old projects relying on unavailable Java APIs. For projects that the authors weren't able to compile, they downloaded the binaries as EVOSUITE did'nt require source code for test generation.The authors had to consider a total of 321 projects until they had a set of 100 projects in binary format, which the authors called [SF100 corpus of classes](http://www.evosuite.org/subjects/sf100/)

IV. Statistical Analysis

The papers reviewed in this essay, all have one thing in common, they all have specific strategies for evaluation based on which they show how their proposed technique fares against other techniques. The evaluations strategies for symbolic execution, dynamic symbolic execution, their combination, and for whole test suite optimization techniques are described below.

1. PEX 2008
   The authors ran Pex on about 10 machines (different configurations, similar to P4, 2GHz, 2GB RAM) for three days; each machine was processing one class at a time. In total, the analysis involved more than 10,000 public methods with more than 100,000 blocks and more than 100,000 arcs. Block coverage and arc coverage were used as metrics for evaluation

but the number of blocks and arcs actually reachable were not determined.

2. Parallel Symbolic Execution 2011
   For each NPW (number of parallel workers) ‘x’, the authors randomly sampled ‘x’ test suites and determined for each coverage obligation ‘o’ the earliest time ‘o’ was satisfied by any test suite. This information was used to determine the earliest time all obligations ‘o’ are satisfied by any of the sampled test suites. The authors termed this the *time to finish (TTF)* for the run and performed resampling for each NPW 1,000 times (or the maximum number of possible times) and average the resulting TTFs.

3. A Theoretical and Empirical Study of Search-Based Testing 2010
   The authors used nine different programs for their empirical study, of which 38 functions containg 760 branches were studied. The factors that influenced their decision of choosing the projects they studied were-
   a. Input Domain Size: The search space sizes for the programs ranged from 10<sup>5</sup> to 10<sup>524</sup>.
   b. Difficulty of the Problem: Each separate branch denoted a different search problem, and its size determined the level of difficulty.
   c. Complexity: The code analyzed contained many examples of complex, unstructured control flow, unbounded loops, and computed storage locations in the form of pointers and array access.

4. FloPSy 2010
   The empirical study was split into two parts. The first part contained a set of benchmark functions which are commonly used to evaluate optimization algorithms. The second part consisted of two real world programs comprising 152, 372 lines of C# code which were chosen because they contained arithmetic operations over floating point variables. Block coverage was used as a metric of statistical measure. Pex was used for test data generation and 100% block coverage of the methods indicated that the Pex goal has been reached.

5. Combining Search Based and Constraint Based Testing 2011
   The authors developed a prototype based on Java PathFinder and evaluated it on a set of 20 case study objects containing a combination of linear/non-linear constraints, and used 50,000 test executions to compare the performance of random search, genetic algorithm (GA), DSE, GA-DSE (proposed hybrid) based on achieved coverage.

6. EVOSUITE 2011
   The authors ran EVOSUITE against the single branch strategy for each of the 727 public classes, to compare their achieved coverage. Each experiment comparison was repeated 100 times with different seeds for the random number generator.
   - Mann Whitney U Test*- It is a non-parametric statistical hypothesis test, i.e. it allows the comparison of two samples with unknown distributions. It was used to assess whether the effectiveness of EVOSUITE and single branch based approach were statistically different.
   - Vargha-Delaney Aˆ12 statistic*- To measure the magnitude of the difference in a standardized way i.e. using the _effect size_; the Aˆ12 statistic was used. In the context of this experiment, the Aˆ12 is an estimation of the probability that, better coverage is obtained upon running EVOSUITE, than running the single branch strategy. When two randomized algorithms were equivalent, then Aˆ12 = 0.5. A high value Aˆ12 = 1 meant that, in all of the 100 runs of EVOSUITE, higher coverage values were obtained than the ones obtained in all of the 100 runs of the single branch strategy.
   - Boxplots were used to visualize the comparison between the two approaches for all the six case studies by using as a measure –
     a. Average branch coverage
     b. Aˆ12 for coverage
     c. Average length values when Aˆ12 = 0.5
     d. Aˆ12 for length

7. EXSYST 2012
   To compare the effectiveness of search-based system testing with state-of-the-practice techniques, the authors compared EXSYST against three tools: Randoop [18], EVOSUITE [2] and GUITAR [20]. The authors run the respective tool on five test subjects for 15 minutes and determined
   1. The number of tests executed,
   2. The number of failures encountered,
   3. The instruction coverage achieved.

   The more instructions a test covered, the more it exercised aspects of internal program behavior.

8. EVOSUITE 2013
   Since the independence of the order in which test cases are selected and the collateral coverage are inherent to the EVOSUITE approach; the evaluation focuses on the improvement over the single branch strategy. For the evaluation, the authors chose a total of 19 open source libraries and programs.
   a. to analyze in more detail some specific types of software, the authors used a translation of the String case study subjects

b. To avoid a bias caused by considering only open source code, the authors selected a subset of an industrial case study project

c. To avoid bias in analyzing the results, the authors presented and discussed the results of their empirical study grouped by project. Different testing techniques can have comparatively different performance on different types of SUT is assumed.

VI. Conclusions

This paper has presented a summary of the papers reviewed chronologically from 2008 to 2013 which have search based software testing and symbolic execution based testing as the central motif. In this paper we have attempted to represent a flow in the research being undertaken, and the direction of the research over a span of five years. A pattern has been seen where the initially automated test case generation was limited to constraint solving, and over the years has been combined with various methods of search based testing for improvement in coverage.

VII. Future Recommendations

The following future work is recommended –

Symbolic Execution

Parallelization of symbolic execution [4] using simple static and dynamic portioning could be extended and employed in tools such as PEX and EVOSUITE to improve the scalability of symbolic execution, and might lead to a wider adoption of automated test case generation tools.

Search Based Testing

The work done in [6] compares various search based testing methodologies and establishes the specific application areas where these techniques are best suited. This knowledge if best utilized could prove insightful in applying the best search based methodology in the best scenario. This work could be extended to include more evolutionary methodologies such as Ant Colony Optimization and other swarm based evolutionary algorithms.

Integrated Symbolic Execution and Search Based Testing

The integration of symbolic execution and search based testing methods have improved from [3], [5] to [1]. A further improvement in [1] could be a more holistic evaluation of the proposed technique, probably on the class corpus suggested by Fraser et al. in [9]

Whole Test Suite Optimization

The EVOSUITE tool have been improvised by Fraser et al. in [2], [8] and [10], the future work here could be –

- Combination of search-based test generation with dynamic symbolic execution, and search optimizations such as testability transformation or local search to further improve the achieved coverage.
- Integration of enhancements in the literature of search algorithms and their evaluation in EVOSUITE, as, for example, island models and adaptive parameter control.
- Investigation of ways to support the developer by automatically producing effective assertions for mitigating the _oracle problem_ and, making the produced test cases more readable for ease of understanding.

VII. Acknowledgment

This paper is a part of CSC-791 Automated Software Engineering. The author would like to thank the instructor Dr. Tim Menzies and the teaching assistant Rahul Krishna for their support and motivation in pursuing this project.

VIII. References

[1] Jan Malburg, Gordon Fraser. 2011. “Combining search-based and constraint-based testing”, *26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011)*, pp. 436-439.

[2] Gordon Fraser and Andrea Arcuri. 2011. “Evolutionary Generation of Whole Test Suites”. In *Proceedings of the 2011 11th International Conference on Quality Software* (QSIC '11). IEEE Computer Society, Washington, DC, USA, 31-40.

[3] K. Inkumsah and Tao Xie. 2008. “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). IEEE Computer Society, Washington, DC, USA, 297-306.

[4] Matt Staats and Corina Păsăreanu. 2010. “Parallel symbolic execution for structural test generation”. In *Proceedings of the 19th international symposium on Software testing and analysis* (ISSTA '10). ACM, New York, NY, USA, 183-194.

[5] Kiran Lakhotia, Nikolai Tillmann, Mark Harman, and Jonathan De Halleux. 2010. “FloPSy: search-based floating point constraint solving for symbolic execution”. In *Proceedings of the 22nd IFIP WG 6.1 international conference on Testing software and systems* (ICTSS'10), Alexandre Petrenko, Adenilso Simão, and José Carlos Maldonado (Eds.). Springer-Verlag, Berlin, Heidelberg, 142-157. Mark Harman and Phil McMinn. 2010. A Theoretical and Empirical Study of Search-Based Testing: Local, Global, and Hybrid Search. *IEEE Trans. Softw. Eng.* 36, 2 (March 2010), 226-247.

[6] Mark Harman and Phil McMinn. 2010. “A Theoretical and Empirical Study of Search-Based Testing: Local, Global, and Hybrid Search”. *IEEE Trans. Softw. Eng.* 36, 2 (March 2010), 226-247.

[7] Nikolai Tillmann and Jonathan De Halleux. 2008. “Pex: white box test generation for .NET”. In*Proceedings of the 2nd international conference on Tests and proofs* (TAP'08), Bernhard Beckert and Reiner Hähnle (Eds.). Springer-Verlag, Berlin, Heidelberg, 134-153.

[8] Florian Gross, Gordon Fraser, and Andreas Zeller. 2012. “Search-based system testing: high coverage, no false alarms”. In *Proceedings of the*

2012 International Symposium on Software Testing and Analysis (ISSTA 2012). ACM, New York, NY, USA, 67-77.

[9] Gordon Fraser and Andrea Arcuri. 2012. “Sound empirical evidence in software testing”. In*Proceedings of the 34th International Conference on Software Engineering* (ICSE '12). IEEE Press, Piscataway, NJ, USA, 178-188.

[10] Gordon Fraser and Andrea Arcuri. 2013. “Whole Test Suite Generation”. *IEEE Trans. Softw. Eng.* 39, 2 (February 2013), 276-291.

[11] Godefroid, P., Klarlund, N., Sen, K.: “DART: directed automated random testing”. SIGPLAN Notices 40(6), 213–223 (2005)

[12] Godefroid, P.: “Compositional dynamic test generation”. In: POPL 2007: Proceedings of the 34th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages, pp. 47–54. ACM Press, New York (2007)

[13] Cadar, C., Ganesh, V., Pawlowski, P.M., Dill, D.L., Engler, D.R.: “Exe: automatically generating inputs of death”. In: CCS 2006: Proceedings of the 13th ACM conference on Computer and communications security, pp. 322–335. ACM Press, New York (2006)

[14] Sen, K., Agha, G.: “CUTE and jCUTE: Concolic unit testing and explicit path model-checking tools”. In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 419–423. Springer, Heidelberg (2006)

[15] Sen, K., Marinov, D., Agha, G.: “Cute: a concolic unit testing engine for c”. In:ESEC/FSE-13: Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering, pp. 263–272. ACM Press, New York (2005)

[16] Anand, S., Pasareanu, C.S., Visser, W.: “Jpf-se: A symbolic execution extension to java pathfinder”. In: Grumberg, O., Huth, M. (eds.) TACAS 2007. LNCS, vol. 4424, pp. 134–138. Springer, Heidelberg (2007)

[17] Grieskamp, W., Tillmann, N., Schulte, W.: “XRT - Exploring Runtime for .NET - Architecture and Applications”. In: SoftMC 2005: Workshop on Software Model Checking, July 2005. Electronic Notes in Theoretical Computer Science (2005)

[18] C. Pacheco and M. D. Ernst, “Randoop: Feedback-directed random testing for Java,” in *Companion to ACM SIGPLAN Conference on Object Oriented Programming Systems and Applications Companion*, 2007, pp.815–816.

[19] Tonella, P.: “Evolutionary testing of classes”. In: ISSTA 2004, pp. 119–128 (2004)

[20] "GUITAR — a GUI Testing frAmewoRk", 2009. Website.

[21] T. Xie, D. Marinov, and D. Notkin, “Rostra: A framework for detecting redundant object-oriented unit tests,” in *Proc. IEEE International Conference on Automated Software Engineering*, 2004, pp. 196–205.

[22] T. Xie, D. Marinov, W. Schulte, and D. Notkin, “Symstra: A framework for generating object-oriented unit tests using symbolic execution,” in *Proc. International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, 2005, pp. 365–381.