SUSHI: A Test Generator for Programs with Complex Structured Inputs

Pietro Braione
University of Milano-Bicocca
Italy
braione@disco.unimib.it

Giovanni Denaro
University of Milano-Bicocca
Italy
denaro@disco.unimib.it

Andrea Mattavelli
Imperial College London
United Kingdom
amattave@imperial.ac.uk

Mauro Pezzè*
Università della Svizzera italiana (USI)
Switzerland
mauro.pezze@usi.ch

ABSTRACT

Random and search-based test generators yield realistic test cases based on program APIs, but often miss structural test objectives that depend on non-trivial data structure instances; Whereas symbolic execution can precisely characterise those dependencies but does not compute method sequences to instantiate them. We present SUSHI, a high-coverage test case generator for programs with complex structured inputs. SUSHI leverages symbolic execution to generate path conditions that precisely describe the relationship between program paths and input data structures, and converts the path conditions into the fitness functions of search-based test generation problems. A solution for the search problem is a legal method sequence that instantiates the structured inputs to exercise the program paths identified by the path condition. Our experiments indicate that SUSHI can distinctively complement current automatic test generation tools.

CCS CONCEPTS

• **Software and its engineering → Software testing and debugging**;

KEYWORDS

Automatic test case generation, Symbolic execution, Search-based software engineering

ACM Reference Format:
Pietro Braione, Giovanni Denaro, Andrea Mattavelli, and Mauro Pezzè. 2018. SUSHI: A Test Generator for Programs with Complex Structured Inputs. In *ICSE '18 Companion: 40th International Conference on Software Engineering , May 27-June 3, 2018, Gothenburg, Sweden.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3183440.3183472

1 INTRODUCTION

Automatic test case generation alleviates the burden of writing tests [2, 5, 8, 10]. Current test generators exploit random testing, search-based testing, and symbolic execution. Random test generators synthesize test cases by randomly sampling method sequences to exercise the program under test through its public interface [10]. Search-based test generators steer the random selection with fitness functions that favor the execution of specific code elements, for instance program branches [7]. Both random and search-based test generators ignore the relationship between the program control flow and the input data structures. As a result, they can successfully synthesize method sequences that instantiate simple data structures, but do not satisfactorily deal with test goals that can only be triggered with non-trivial input data structures.

The branch at line 14 in Figure 1 exemplifies a non-trivial dependence from data structures. The program takes two lists of processes (line 3), sets the processes in list primary to have high priority (lines 4–5), builds a new list that includes all processes of list primary (line 6) and possibly the ones of list secondary (lines 7–10), and finally processes the resulting list by executing the high priority processes first (lines 11–13) while postponing the others (line 14). A test case that executes the postponing code at line 14 must invoke method execPool with a primary list that contains more processes than physical units (eight), and a secondary list with at least one low-priority process. State-of-the-art random and search-based test generators like Randoop [10] and EvoSuite [7] do not cover the branch at line 14 in Figure 1 in 12 hours: A pure random sampling has negligible probability to find the required inputs; Whereas a search-based algorithm has no guidance to identify that, to satisfy the branch condition p.getPriority<HIGH_PRIORITY, the process p must belong to list secondary, and that this can happen only when list primary is longer than eight.

Some symbolic executors generate path conditions that precisely characterize the constraints over the input data structures to execute given program paths [2, 9, 11, 13, 14], for instance a path condition for the branch at line 14 in Figure 1. However, they do not directly generate executable test cases that instantiate concrete data structures that satisfy the path conditions. Some test generators work around this problem by synthesizing tests that bypass the program interfaces and directly manipulate the memory, but recent studies indicate that this practice leads to many unrealistic and sometimes illegal test cases [1, 6]. Other symbolic executors exploit static analysis to identify method sequences that satisfy the path conditions [12, 13]. They succeed in generating test cases that instantiate simple data structures, but fail to deal with complex data structures due to both theoretical limitations and the difficulty of tuning the efficiency-precision tradeoff of their static analysis.

This paper presents SUSHI, a test generator that efficiently synthesizes test cases composed of sequences of legal method calls that exercise program paths that depend on complex structured inputs. SUSHI instantiates the approach that we introduced in a recent paper [1]. SUSHI (i) symbolically executes the program to

*Also with University of Milano-Bicocca, Italy.

ICSE '18 Companion, May 27-June 3, 2018, Gothenburg, Sweden

ACM ISBN 978-1-4503-5663-3/18/05.
https://doi.org/10.1145/3183440.3183472

```
1  int PHYSICAL_UNITS = 8;
2  int HIGH_PRIORITY = 10;
3  void execPool(List<Process> primary, List<Process> secondary) {
4    for (Process p : primary)
5      p.setPriority(HIGH_PRIORITY);
6    List<Process> toExec = new LinkedList<>(primary);
7    if (primary.size() > PHYSICAL_UNITS) {
8      switchToVirtualUnits();
9      toExec.addAll(secondary);
10   }
11   for (Process p : toExec) {
12     if (p.getPriority() >= HIGH_PRIORITY)
13       exec(p);
14     else postpone(p);
15   }
16 }
```

Figure 1: Sample program execPool()

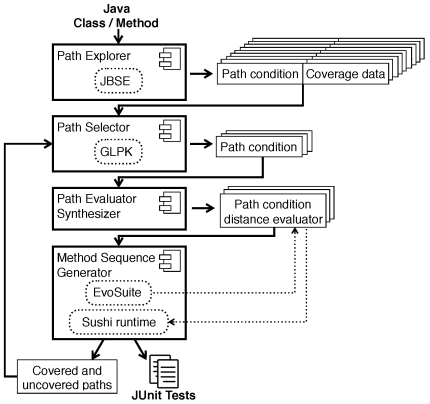

Figure 2: SUSHI architecture

obtain path conditions that characterize the dependencies between the program paths and the structure of the input data, (ii) selects a suitable subset of path conditions to address some coverage goals, (iii) converts the selected path conditions into fitness functions that quantify the distance of an input state from satisfying a path condition, and (iv) exploits the fitness functions for a meta-heuristic search to generate legal sequences of method calls that satisfy the path conditions.

This paper extends the research paper [1] by exploring in detail the design and implementation of the SUSHI tool. This paper (i) presents the modular architecture of the tool, (ii) specifies the path selection algorithm that SUSHI uses to maximize coverage with respect to a chosen structural coverage criterion, (iii) illustrates the SUSHI meta-heuristic search phase that exploits the EvoSuite test generator ([7]) guided by the fitness functions derived from the path conditions, (iv) describes the feedback loop that SUSHI uses to mitigate the impact of infeasible paths on test generation process, and (v) discusses how an end-user can use SUSHI to automatically generate high-coverage test cases.

2 SUSHI DESIGN

SUSHI is an open-source test generator for Java[1] that synthesizes test suites with high branch coverage. Figure 2 illustrates the modular architecture of SUSHI: The *Path Explorer* leverages symbolic execution to explore the program execution space and to compute the execution conditions of the program paths. The *Path Selector* translates the task of selecting a subset of paths to maximize branch coverage into a linear programming problem that it solves with a specialized library. The *Path Evaluator* implements our recent approach [1] that turns path conditions into executable evaluators that quantify the distance of a concrete state from satisfying the corresponding path condition. The *Method Sequence Generator* synthesizes concrete test cases for the selected path conditions with a search-based engine that uses the evaluators as fitness functions.

2.1 Path Explorer

SUSHI exploits the symbolic executor JBSE [3] to both traverse the program paths in depth-first order (up to some user-defined bounds) and compute the corresponding path conditions. JBSE deals with heap-symbolic reasoning and includes a set of techniques to account for structural invariants of structured inputs [2, 4].

SUSHI leverages JBSE to collect both path conditions and coverage information, that is, the set of program branches traversed during symbolic execution. For example, for the program in Figure 1, JBSE explores a path that traverses the branch at line 14 with the path condition[2]

$$primary.size > 8 \wedge secondary.size > 0 \wedge secondary.header.next.elem.priority < 10$$

that characterizes the set of inputs that execute the branch at line 14.

2.2 Path Selector

The *Path Selector* identifies an adequate subset of paths to cover the program branches by formulating and solving the path selection task as a linear integer programming problem. The *Path Selector*:

(1) turns the coverage information into a matrix of coefficients c_{ij}, such that each row i corresponds to an explored path and each column j corresponds to an explored branch (a coefficient c_{ij} is 1 if the i-th path covers the j-th branch, and 0 otherwise),
(2) associates each explored path with a variable s_i that the solver sets to either 0 or 1 to indicate whether the given path belongs to the path selection, and
(3) uses the GLPK solver[3] to find an optimal assignment of the variables s_i that minimizes the number of selected paths, $paths = \sum_i s_i$, while guaranteeing the coverage of all explored branches, expressed as a set of linear constraints $\sum_i c_{ij} * s_i > 0$, one for each branch j. The resulting assignment indicates the subset of paths that maximize branch coverage.

SUSHI, which currently maximizes branch coverage, can be easily adapted to address other structural testing criteria by modifying the *Path Explorer* component to yield a suitable mapping between the explored paths and the target code elements.

2.3 Path Evaluator Synthesizer

The *Path Evaluator Synthesizer* converts a selected path condition into a distance evaluator program, which computes the *distance* of a test case t from satisfying all the clauses in the path condition. In a nutshell, the evaluator program inspects the execution state

[1] https://github.com/pietrobraione/sushi

[2] We show only the key execution conditions of the branch at line 14.

[3] https://www.gnu.org/software/glpk/

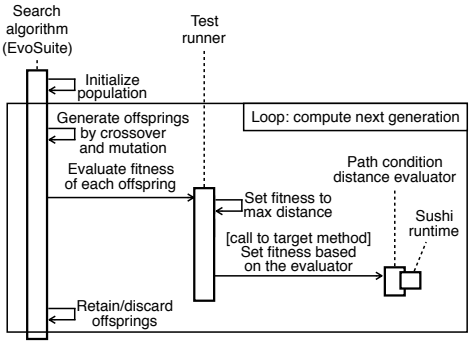

Figure 3: Search algorithm with path condition fitness

that corresponds to the test case t to verify that it complies with the execution conditions in the path condition.

The evaluator program is an executable method that SUSHI invokes with the inputs that the test case t is currently passing to the program under test. The evaluator returns either the minimal distance 0—if the inputs satisfy all the clauses in the corresponding path condition—or a distance greater than 0 otherwise. Each unsatisfied path condition clause contributes to increase the returned distance by $(0, 1]$, with values that are increasingly closer to 1 for inputs that miss the clause by far.

For example, for the path condition in Section 2.1, a test case with 9 processes in list primary and 1 low-priority process in list secondary evaluates to the optimal distance 0, since it satisfies all the clauses in the path condition. Conversely, a test case with 8 processes in list primary evaluates to a distance $\alpha > 0$, because it does not satisfy the clause $primary.size > 8$; a test case with seven processes in list primary evaluates to a distance $\beta > \alpha$, because it misses the same clause for a larger amount, and so on. We refer the readers to [1] for a detailed mathematical definition of the evaluator.

The *Path Evaluator Synthesizer* is the core technical novelty of SUSHI [1]. Differently from traditional symbolic executors [12–14], SUSHI generates only test cases that correspond to legal sequences of method calls: SUSHI converts the path conditions into the fitness functions of a meta-heuristics search process that steers the search towards method sequences that produce only reachable program states that satisfy the path conditions.

2.4 Method Sequence Generator

The *Method Sequence Generator* integrates the path condition distance evaluators into a meta-heuristic search process to synthesize method call sequences that instantiate program states that satisfy the corresponding path conditions.

The *Method Sequence Generator* leverages the EvoSuite tool as a meta-heuristic search engine based on genetic algorithms. Figure 3 illustrates the search algorithm and its integration with the path condition distance evaluators. The search algorithm starts with an initial *population* of randomly generated method sequences, and incrementally creates new generations. The *Method Sequence Generator* builds a new generation by extending the current generation with new method sequences (called *offsprings*) generated by applying evolutionary operators. Standard evolutionary operators include *mutation*, which randomly changes, adds or deletes a statement, and *crossover*, which combines two method sequences [7].

```
1 ExecutionHandler executionHandler = new ExecutionHandler();
2 LinkedList<Process> list0 = new LinkedList<>();
3 LinkedList<Process> list1 = new LinkedList<>();
4 list0.add(new Process());
5 // ... then, the test case adds 8 additional Process to list0
6 Process process10 = new Process();
7 process10.setPriority(4);
8 list1.add(process10);
9 executionHandler.executePool(list0, list1);
```

Figure 4: A SUSHI test case for the program in Figure 1

For each newly generated offspring, the algorithm computes a *fitness value* that represents the distance from generating a method sequence that invokes the target method in a state that satisfies all the clauses in the path condition, and thus exercise the corresponding program path. The search algorithm computes the fitness value by executing the method sequence. If the method sequence does not invoke the target method, then the test runner returns a maximal distance value, since the method sequence cannot cover the target path by construction. Otherwise, upon intercepting a call to the target method, the test runner executes the path condition distance evaluator using the same parameters as the target method invocation and sets the fitness value according to the distance value returned by the evaluator. In the case of multiple calls to the target method, the resulting fitness is the best value among all calls.

The algorithm returns a test case upon identifying a method sequence with optimal fitness. For instance, when using the evaluator that represents the path condition discussed in Section 2.1, the method sequence generator yields the test case in Figure 4 that successfully covers the branch at line 14 in Figure 1. The execution of EvoSuite is constrained by a time budget, and the *Method Sequence Generator* may not return a test case for a path condition when the search exceeds the time budget.

2.5 Handling Infeasible Paths

The *Method Sequence Generator* produces concrete test cases only if the selected path condition corresponds to a feasible path, that is a valid program behavior. Nonetheless, the *Path Explorer* may conservatively compute some unsatisfiable path conditions, because of the inability to decide the satisfiability of some formulas. Yet, the *Path Explorer* might overlook some implicit program invariants, and produce path conditions that violate these constraints.

SUSHI mitigates the impact of infeasible paths with two complementary mechanisms:

(1) it relies on the support of JBSE to exploit specifications of preconditions and invariants provided by the developers [2, 4];
(2) it implements a feedback loop to monitor the concretization of the path conditions to method call sequences. If the *Method Sequence Generator* fails to concretize a path condition, SUSHI reiterates from the *Path Selector* (Figure 2) to consider alternative path conditions that contribute to achieve optimal coverage.

The feedback loop benefits from a parallel deployment of the *Method Sequence Generator* to improve the efficiency of the generation process: SUSHI instantiates a pool of EvoSuite processes to work simultaneously on the selected path conditions via their corresponding distance evaluators. Whenever an EvoSuite process succeeds in generating a test case for a path condition, it notifies the *Path Selector* of the covered branches. At each iteration, the

Path Selector optimizes a simplified problem that does not consider covered branches, and previously selected path conditions.

3 USING SUSHI

SUSHI is accessible both stand alone from command-line and as a plugin for some development environments, namely the Eclipse IDE. Developers can execute SUSHI by specifying the desired options as a Java class that extends the interface sushi.configure.ParametersModifier, and launch SUSHI with such options. SUSHI options specify the signature of the method (or the class) under test, the program binaries and dependencies, and the output directory where SUSHI stores the generated test cases. Developers can also specify options to customize the size of the pool of EvoSuite processes to spawn in parallel, a specific set of target branches to be considered in the path selection phase, and any custom option accepted from the underlying tools JBSE and EvoSuite, including the invariants to use during symbolic execution and the time budgets for both tools. SUSHI produces executable JUnit test cases.

SUSHI provides high visibility on intermediate outputs to allow for monitoring all the phases of the test generation process and inspect the intermediate results. The intermediate outputs include (i) the symbolic execution traces, each identified with a *trace-id* that allows users to replay the symbolic execution of that specific path with JBSE, (ii) the coverage information of the explored paths, (iii) the list of path conditions selected by the *Path Selector*, (iv) the Java code of the path condition distance evaluators that the *Path Evaluator Synthesizer* builds, and (v) the log file produced when running each EvoSuite instance. These intermediate outputs are valuable information both for debugging SUSHI and for inspecting the results. The intermediate outputs support debugging by providing information about the observed behaviors, and augment the usefulness of the final results by for example supporting the inspection of the symbolic execution traces that are selected but not concretized, to spot missing data structure invariants, and improve the effectiveness of the test generation process by refining the set of invariants and precondition used for symbolic execution.

4 EVALUATION

We evaluated SUSHI by generating test cases for a set of benchmark programs that work with non-trivial data structures as inputs. We compared the branch coverage of the test cases generated with SUSHI against the coverage of the test cases generated with JBSE, Seeker, and EvoSuite. JBSE generates test cases by directly manipulating the heap memory [3]. Seeker exploits static analysis to build method sequences that steer the symbolic executor Pex towards uncovered branches [12]. EvoSuite implements search-based testing using the distance from uncovered branches as a fitness function [7]. We experimented with symbolic executors configured with both a small (*partial*) and a comprehensive (*accurate*) set of invariants.

Table 1 reports the branch coverage of the generated test suites. We refer the readers to [1] for all details of the experimental setting, the corresponding replication package, and fine-grain analysis of the results. The results indicate that SUSHI consistently improves over all other embodiments of symbolic execution (JBSE and Seeker) in all experiments, and exercises relevant complementary branches with respect to EvoSuite, both with partial and accurate invariants.

Table 1: Branch coverage with SUSHI and competing tools

		SUSHI		JBSE		Seeker	EvoSuite
	Invariants	Partial	Accurate	Partial	Accurate	Accurate	Not used
Subjects (branches)	avl (40 br.)	100%	100%	45%	100%	70%	100%
	treemap (164)	84%	88%	61%	88%	25%	88%
	caching (36)	86%	86%	69%	86%	75%	81%
	tsafe (36)	97%	97%	§14%	§14%	36%	67%
	gantt (26)	62%	92%	19%	92%	31%	92%
	clos01 (279)	4%	37%	3%	§16%	*n.a.	15%
	clos72 (23)	70%	78%	#0%	#0%	*n.a.	43%

Gray highlights fault-revealing tests
* .NET version of the program not available
# Tests throw runtime exceptions
§ Tests for linear path conditions only

Moreover, SUSHI is the only tool that generates test suites that reveal the faults in the programs. The results also indicate a significant impact on the accuracy of the invariants on the effectiveness of SUSHI: SUSHI consistently reaches equal or higher coverage when used with accurate invariants than with partial invariants. We plan to work on the automatic synthesis of invariants, and to exploit even more the complementarity with EvoSuite to enhance the effectiveness of the test generator.

5 CONCLUSION

We presented SUSHI, a novel open-source test generator for Java tailored for programs with complex structured inputs. Preliminary results on a benchmark of Java programs suggest that SUSHI has the potential to improve on state of the art of automatic test generators.

REFERENCES

[1] P. Braione, G. Denaro, A. Mattavelli, and M. Pezzè. Combining symbolic execution and search-based testing for programs with complex heap inputs. In *Proc. of the Intl. Symposium on Software Testing and Analysis*, pages 90–101, 2017.

[2] P. Braione, G. Denaro, and M. Pezzè. Symbolic execution of programs with heap inputs. In *Proc. of the European Software Engineering Conf. held jointly with the Intl. Symposium on Foundations of Software Engineering*, pages 602–613, 2015.

[3] P. Braione, G. Denaro, and M. Pezzè. JBSE: A symbolic executor for java programs with complex heap inputs. In *Proc. of the European Software Engineering Conf. held jointly with the ACM SIGSOFT Intl. Symposium on Foundations of Software Engineering*, pages 1018–1022, 2016.

[4] P. Braione, G. Denaro, and M. Pezzè. Enhancing symbolic execution with built-in term rewriting and constrained lazy initialization. In *Proc. of the European Software Engineering Conf. held jointly with the ACM SIGSOFT Intl. Symposium on Foundations of Software Engineering*, pages 411–421, 2013.

[5] C. Cadar, D. Dunbar, and D. Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proc. of the Symposium on Operating Systems Design and Implementation*, pages 209–224, 2008.

[6] L. Cseppento and Z. Micskei. Evaluating symbolic execution-based test tools. In *Proc. of the Intl. Conf. on Software Testing, Verification and Validation*, 2015.

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

[8] P. Godefroid, M. Y. Levin, and D. Molnar. Sage: Whitebox fuzzing for security testing. *ACM Queue*, 10(1):20–27, 2012.

[9] S. Khurshid, C. S. Păsăreanu, and W. Visser. Generalized symbolic execution for model checking and testing. In *Proc. of the Intl. Conf. on Tools and Algorithms for Construction and Analysis of Systems*, 2003.

[10] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball. Feedback-directed random test generation. In *Proc. of the Intl. Conf. on Software Engineering*, pages 75–84, 2007.

[11] N. Rosner, J. Geldenhuys, N. Aguirre, W. Visser, and M. F. Frias. BLISS: improved symbolic execution by bounded lazy initialization with SAT support. *IEEE Transactions on Software Engineering*, 41(7):639–660, 2015.

[12] S. Thummalapenta, T. Xie, N. Tillmann, J. de Halleux, and Z. Su. Synthesizing method sequences for high-coverage testing. In *Proc. of the Conf. on Object-Oriented Programming Systems and Applications*, pages 189–206, 2011.

[13] N. Tillmann and J. de Halleux. Pex: White box test generation for .NET. In *Proc. of the Intl. Conf. on Tests and Proofs*, pages 134–153, 2008.

[14] W. Visser, C. S. Păsăreanu, and S. Khurshid. Test input generation with java pathfinder. In *Proc. of the Intl. Symposium on Software Testing and Analysis*, 2004.