Combining Symbolic Execution and Search-Based Testing for Programs with Complex Heap 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

Despite the recent improvements in automatic test case generation, handling complex data structures as test inputs is still an open problem. Search-based approaches can generate sequences of method calls that instantiate structured inputs to exercise a relevant portion of the code, but fall short in building inputs to execute program elements whose reachability is determined by the structural features of the input structures themselves. Symbolic execution techniques can effectively handle structured inputs, but do not identify the sequences of method calls that instantiate the input structures through legal interfaces. In this paper, we propose a new approach to automatically generate test cases for programs with complex data structures as inputs. We use symbolic execution to generate path conditions that characterise the dependencies between the program paths and the input structures, and convert the path conditions to optimisation problems that we solve with search-based techniques to produce sequences of method calls that instantiate those inputs. Our preliminary results show that the approach is indeed effective in generating test cases for programs with complex data structures as inputs, thus opening a promising research direction.

CCS CONCEPTS

• **Software and its engineering → Software testing and debugging**; *Formal software verification*;

KEYWORDS

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

ACM Reference format:
Pietro Braione, Giovanni Denaro, Andrea Mattavelli, and Mauro Pezzè. 2017. Combining Symbolic Execution and Search-Based Testing for Programs with Complex Heap Inputs. In *Proceedings of 26th International Symposium on Software Testing and Analysis , Santa Barbara, CA, USA, July 2017 (ISSTA'17),* 12 pages.
https://doi.org/

*Also with University of Milano-Bicocca, Italy.

ISSTA'17, July 2017, Santa Barbara, CA, USA

ACM ISBN 978-1-4503-5076-1/17/07…$15.00
https://doi.org/

1 INTRODUCTION

Automatically generating test cases amounts to sample the program input space, characterise relevant values for the inputs, and synthesise method sequences that instantiate the input variables of the program under test. The problem of sampling the input space and characterising relevant input values has been widely investigated [10, 25, 28, 40, 44, 51]. Functional and model-based testing sample the input space according to specifications and models, structural testing techniques are driven by the code structure [47].

The challenge of producing sequences of method calls that initialise the input and the state variables that involve complex data structures has been only partially addressed so far.

Specification-based test generation techniques, like Korat [7], TestEra [38], and Udita [33], exploit invariant specifications to identify test-relevant states of input data structures, but rely on the possibility of assigning member fields directly (direct heap manipulation) to instantiate the objects that correspond to the identified input states. Symbolic techniques, like JPF [53], jPET [1], JBSE [10, 11], and BLISS [48], identify input states to execute code elements, but rely on direct heap manipulation to instantiate the data structures that comprise these input states. Test cases that instantiate the input data structures by directly manipulating the heap memory may either not compile, since they directly access private fields, or create unreachable program states, since they bypass the program functions that preserve the invariants [17], and are often hardly readable and understandable by developers [18].

The challenge of automatically generating sequences of method calls that suitably instantiate input structures has been addressed with random, search-based, and symbolic execution approaches.

Random and search-based approaches generate valid and executable test cases by executing the program under test and sampling the execution space through the program interfaces [16, 28, 45]. These approaches work with limited knowledge of the relationship between program branches and input data structures. They effectively generate method sequences that instantiate simple data structures, but fail in generating method sequences that instantiate non-trivial data structures, such as the ones needed to elicit complex and possibly inter-procedural dependencies that may either trigger subtle failures, or cover interesting code elements.

Symbolic analysis approaches synthesise path conditions, which indicate the conditions on the input values to execute given program paths, and generate test cases by identifying concrete values that satisfy the path conditions. Many symbolic executors can effectively handle programs that take primitive inputs [4, 5, 8, 12, 13, 34, 49, 54], but only some can efficiently reason on input data structures as well [1, 10, 20, 48, 51, 53].

```
1 static final int N = ...; //some constant value
2 void sample(LinkedList list, Object obj) {
3   list.addLast(obj);
4   Object r = list.remove(N);
5   if (r == obj) {
6     // error!
7   }
8 }
```

Figure 1: Program *sample* with an input double linked list

```
1 LinkedList list = new LinkedList();
2 Object obj = new Object();
3 list.add(new Object());
4 list.add(new Object());
5 list.add(new Object());
6 list.add(new Object());
7 sample(list, obj);
```

Figure 2: A test case that executes line 6 of program *sample* in Figure 1 (with $N = 4$)

Some approaches combine symbolic execution with search-based techniques. They use symbolic execution to improve method sequences generated with search-based techniques, by symbolically exploring the program paths that can be reached with alternative values of the primitive inputs in these method sequences [35, 44, 53, 54]. These approaches may improve over pure search-based approaches, but share the same limitations of search-based techniques in exploring program branches that depend on complex input structures that require different method sequences.

Modern symbolic execution approaches deal with input data structures, but do not adequately address the synthesis of method calls that instantiate complex input data structures yet [10, 20, 48, 53]. They either instantiate the input data structures by directly manipulating the heap memory [1, 10], or rely on static analysis to build sequences of interface method invocations that produce the input structures that satisfy the path conditions [50, 51]. Approaches that rely on static analysis successfully identify some combinations of constructors and methods that instantiate simple input structures, but often fail in the presence of inter-procedural relations that depend on complex input structures, due to imprecision and inefficiency of static analysis.

In this paper, we propose SUSHI, a new approach that combines symbolic execution and search-based approaches to automatically generate executable test cases with complex data structures as inputs. We symbolically execute the target program to produce path conditions that characterise the structure of the inputs, and convert the path conditions into the objective function of an optimisation problem that we solve with a search-based approach. An optimal solution is a sequence of method calls that builds the input structures that exercise the identified path conditions, thus resulting in test cases that execute the corresponding program paths.

Differently from the approaches that exploit search-based techniques to generate test cases and symbolic execution to explore alternative values of the primitive inputs [35, 53, 54], we take full advantage of the power of symbolic execution to precisely characterise complex input structures that trigger the execution of relevant paths in the programs, and then exploit search-based approaches to generate concrete method sequences that instantiate the data structures in the path conditions.

In summary, this paper contributes (i) a complete symbolic execution approach to generate concrete test inputs for programs that manipulate complex input structures, (ii) an automatic approach to convert path conditions into optimisation problems that can be heuristically solved with search-based techniques to generate sequences of method invocations that produce input data structures that satisfy the path conditions, (iii) a prototype implementation of SUSHI, and (iv) initial empirical evidence that the proposed approach outperforms current approaches in generating concrete test cases with complex data structures as inputs.

This paper is organised as follows. Section 2 discusses in detail the limitations of search-based and symbolic execution test case generation approaches through a simple example. Section 3 describes our approach, SUSHI, and provides the essential details of the prototype implementation that we used to gather experimental evidence of its effectiveness. Section 4 presents the experimental results that confirm the ability of SUSHI to automatically generate test cases for programs with complex structured inputs, and reports some representative case studies that indicate the advances of SUSHI with respect to state-of-the-art approaches. Section 5 surveys the most relevant related work. Section 6 summarises the main contribution of this paper and indicates our current research directions.

2 MOTIVATING EXAMPLE

We illustrate the achievements and limitations of the current approaches for generating concrete test cases for programs with input data structures by referring to the simple code in Figure 1. Method *sample* takes in input a doubly linked list *list* and a generic object *obj* (line 2), appends *obj* to *list* (line 3), removes the element at some index N from *list* (line 4), and enters an error state if the removed element r and *obj* refer to the same object in memory (lines 5–6).

Although method *sample* is only a few lines long and does not contain any reference to peculiar or esoteric language features, current state-of-the-art test generators fail in producing test inputs that exercise the faulty statement at line 6.

Random test case generators cannot cope with complex data structures. As a representative example, the popular random test case generator Randoop [45] fails to generate a test case to execute the error statement at line 6, even with a time budget of 12 hours. In particular, Randoop fails to generate a test case that executes line 6 of the program because such test cases consist of lists with exactly N elements or with *obj* at N-th position, which are unlikely to be generated randomly even for low values of N.

Search-based approaches fail in generating test cases that execute line 6 of the program since they miss information on the complex data structure required to exercise the branch at line 5 [28].

Symbolic executors effectively generate the input conditions that characterise the paths that execute the branch, but do not generate valid test cases that satisfy the path conditions [51]. In the next subsections, we illustrate in detail the limits of search-based and symbolic execution approaches, referring to the example in Figure 1.

```
1 | fresh(Obj) &&
2 | fresh(List) &&
3 | fresh(List.header) &&
4 | fresh(List.header.previous) &&
5 | fresh(List.header.next) &&
6 | fresh(List.header.next.next) &&
7 | fresh(List.header.next.next.next) &&
8 | List.header.next.next.next.next = List.header.previous &&
9 | List.size > 3
```

Obj and *List* are the symbolic values for *obj* and *list*

Figure 3: A path condition to reach line 6 of program *sample* in Figure 1 (with $N = 4$)

```
1  | Object obj = new Object();
2  | LinkedList list = new LinkedList();
3  | list.header = new LLEntry();
4  | list.header.previous = new LLEntry();
5  | list.header.next = new LLEntry();
6  | list.header.next.next = new LLEntry();
7  | list.header.next.next.next = new LLEntry();
8  | list.header.next.next.next.next = list.header.previous;
9  | list.size = 7;
10 | sample(list, obj);
```

Figure 4: A test case derived from the path condition of Figure 3 by direct object manipulation (with $N = 4$)

2.1 Search-Based Software Testing

Search-based software testing (SBST) exploits search-based algorithms to automatically generate test cases [28, 46, 52]. SBST formulates the problem of generating test cases as the problem of searching for inputs that maximise a test adequacy criterion, for instance branch coverage. SBST uses a search-based algorithm to maximise the value of an *objective function* that quantifies the distance of a candidate solution from the optimal one. A well-established objective function is the *branch distance* that heuristically quantifies the distance of test inputs from executing a given condition to maximise branch coverage [44]. SBST produces test cases that increase the value of the objective function by iteratively improving the solutions, until reaching either full branch coverage or a timeout.

EvoSuite is one of the most popular tools for SBST of Java programs [28] and generates test cases as sequences of method and constructor invocations. For each parameter, EvoSuite generates primitive values through random sampling, and produces object references through the invocation of constructors and methods that set up the object state. EvoSuite relies on the class APIs to instantiate and modify the internal object state, ultimately producing readable and valid test cases. The large body of experiments reported in the literature witnesses a mature and effective tool, able to generate test suites that consistently achieve high code coverage and reveal software failures [26, 27, 29, 30].

However, when challenged to cover the branch that leads to the error statement at line 6 in Figure 1, EvoSuite cannot produce a method sequence that exercises the branch if $N \geq 15$. EvoSuite cannot generate a valid test case for this branch because its objective function focuses on approaching and satisfying the target branch conditions which, in our example, does not contain enough information about the relationship between the branch and the input data structure.

2.2 Symbolic Execution with Heap Inputs

Symbolic execution builds conditions on symbolic inputs that characterise the input values required to execute the program paths (*path conditions*). A solution of a path condition is a set of concrete values for the input variables to execute the path. Modern symbolic execution approaches deal with complex structured inputs by augmenting the path condition with clauses that predicate on the relations between the object references in the input state [9, 20, 39, 49, 53].

The most popular technique used in symbolic execution to handle structured inputs is *lazy initialisation* [39]. Lazy initialisation starts by executing the program with uninitialised symbolic references and assumes their possible values only upon their dereference. When dereferencing a symbolic reference, lazy initialisation enumerates the possible legal structures for the object, and represents each of them as a constraint in the path condition, thus generating a set of path conditions, one for each possible legal structure. A symbolic reference can be initialised to (i) *null*, (ii) the address of a new object of any compatible type (*fresh*), or (iii) the address of any compatible objects created during a prior initialisation step (*alias*).

Symbolic executors for programs with structured inputs successfully generate a path condition that characterises a test case that exercises the faulty statement at line 6 in Figure 1. Differently from search-based approaches, symbolic execution tracks the relationship between branch conditions and constraints on input data structures. Figure 3 reports the path condition that the JBSE symbolic executor [11] produces by executing the path that leads to the error statement at line 6 of method *sample* with $N = 4$.

The path condition of Figure 3 specifies: (i) the conditions on the symbolic input references *Obj* and *List* (lines 1–2), (ii) the conditions on the values that the internal nodes of the doubly linked list can assume when dereferencing *list*, which are new objects (lines 3–7), except for the last element that should refer to the first element in the list (line 8), and (iii) the condition on *List.size*, the symbolic value of variable *size* that corresponds to the size of the list (line 9).

Test case generation approaches based on symbolic execution generate concrete test inputs from path conditions by identifying concrete input states that satisfy all the clauses of a path condition, and by synthesising code to initialise the state and execute the specific paths in the program. While symbolic execution approaches generate values that satisfy the numeric constraints by means of SMT solvers [6, 19, 24, 36], they provide limited solutions for solving constraints that predicate over complex data structures. As a result, symbolic execution is indeed able to generate a path condition that leads to line 6 in Figure 1, but cannot generate a legal method sequence that leads to execute the line by instantiating the necessary input structures.

Current test case generation approaches based on symbolic execution propose partial solutions to build structured inputs by either directly manipulating the data structures (Section 2.2.1) or by using static analysis (Section 2.2.2).

2.2.1 Building Heap Inputs by Direct Manipulation. Symbolic execution tools like JPF [53], jPET [1], JBSE [10, 11] and BLISS [53] straightforwardly instantiate the input objects in the heap memory, and set the initial state by directly manipulating their internal fields.

Figure 4 shows a test case generated from the path condition in Figure 3 by directly instantiating the objects in the heap and manipulating their fields. The test case is composed of a set of statements that build the concrete inputs (lines 1–9) and an invocation of the target method (line 10). The test case mimics the path condition line by line: it builds and sets the structure of the heap (lines 1–8), initialises variable *list.size* with a value computed with an SMT solver (line 9), and invokes the method *sample* with the instantiated *list* and *obj* references as parameters (line 10).

Creating test cases by directly manipulating object references usually violates the information hiding principle, and often result in compilation errors. Some compilation errors simply stem from accessing private fields and can be fixed through low-level language features that override the visibility restriction. For instance, the test case in Figure 4 can be generated by exploiting the Java reflection APIs to access the private class *LLEntry* and the private fields *header*, *next* and *previous*. However, the resulting test case will appear cryptic to developers, who may not be aware of the internal structure of all components (both internally developed and external reusable libraries) referred to in the test case, ultimately questioning the practical usability of this approach.

Direct heap manipulation might also bypass programming interfaces that maintain representation invariants of the data structures. The resulting test cases may thus set the input objects to states not reachable through the programming interfaces [17], and may result in test cases not yielding feasible behaviours of the program. For example, the test code of Figure 4 assigns the numeric value 7 to *list.size* (line 9), which is a correct solution for the constraint *List.size>3* in the path condition, but violates the invariant of the current list structure that contains only 4 nodes (excluding the sentinel node *list.header*).

2.2.2 Building Heap Inputs with Static Analysis. Other symbolic execution approaches, for example Pex [51] and its extension Seeker [50], instantiate complex structured inputs from path conditions by deriving sequences of interface method calls with static analysis.

Pex generates test cases with dynamic symbolic execution (DSE), a modern extension of classic symbolic execution, and solves the path conditions that involve structured inputs by exploiting an intra-procedural static analysis. The analysis identifies a combination of constructors and methods that potentially instantiate the objects to satisfy the path condition.

Seeker iteratively interleaves DSE and static analysis. In essence, Seeker invokes Pex multiple times with *parametric test drivers* built by statically analysing the code to compute both direct and indirect data dependencies of the uncovered branches. Seeker uses static analysis to identify methods that potentially modify a given field, and exploits the identified methods to incrementally build richer test drivers that include gradually longer method sequences. Seeker symbolically executes the test drivers to both filter out unsuccessful drivers and generate parameters that lead to execute new branches.

In both Pex and Seeker, the static analysis falls short with path conditions that depend on non-trivial inter-procedural relations between method parameters and reachable object states. When challenged to generate test cases for the program in Figure 1, Pex and Seeker limit their generation to test cases that involve *LinkedList* structures with at most a single node, resulting in simple test cases that do not exercise the error statement at line 6, as well as many other relevant cases.

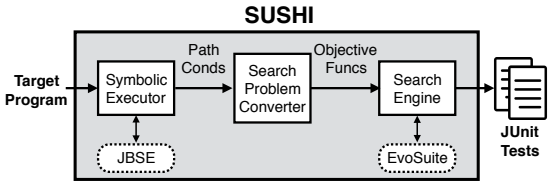

Figure 5: The SUSHI approach

In this paper, we propose *SUSHI*, *Symbolic Unit testing via Search of Heap Inputs*, a new approach that overcomes the limitations of current test case generation techniques for programs with complex data structures as inputs by exploiting an original combination of symbolic execution and search-based techniques. SUSHI combines symbolic execution to generate path conditions that predicate on input data structures with search-based approaches to generate feasible method sequences that produce correct test cases.

3 THE SUSHI APPROACH

SUSHI formulates the problem of finding a sequence of method calls that satisfy a path condition as an optimisation problem over the test space. As shown in Figure 5, SUSHI takes in input a target program and produces test cases that instantiate the input structures in suitable states and execute the target program in such states. The main logical components of SUSHI are a symbolic executor, a search problem converter, and a search engine.

Symbolic executor This component symbolically execute programs with complex structured inputs, for example by using lazy initialisation as discussed in Section 2.2. Current symbolic executors rely on various forms of invariants on the data structures to avoid generating spurious data structures that cannot be instantiated concretely. In this way, they limit the path explosion problem and ultimately enhance the efficiency of the exploration. Popular approaches rely on invariants expressed as executable properties in the form of either *RepOk* methods [20, 53], or declarative properties [10, 48]. This component computes a set of path conditions that address specific test objectives in the target program.

Search problem converter This component converts the path conditions into optimisation problems. For each path condition, it produces an objective function that computes the distance of the current inputs from the optimal solution. The objective function associates a sequence of method calls that instantiate the input objects with a non-negative real value that represents the distance of that input from the optimal solution, that is, an input that satisfies the path condition.

Search engine This component searches for optimal method call sequences by solving the optimisation problem produced with the search problem converter by means of search-based approaches. The identified call sequences represent a test case to exercise the path identified by the path condition.

For program *sample* of Figure 1, SUSHI generates a set of test cases that include the test case illustrated in Figure 2.

The core technical contribution of SUSHI is the *search problem converter*, while the symbolic executor and the search engine are built on top of state-of-the-art symbolic executors and search engines. In the next sections, we formalise the objective functions, the core of the search problem converter, and describe a prototype that instantiates SUSHI for branch testing. Our prototype implementation leverages the JBSE symbolic executor [11] and the EvoSuite search engine [28], as shown in Figure 5.

3.1 The SUSHI Objective Function

The core of the SUSHI approach is the search problem converter component that translates a path condition into the objective function of an optimisation problem over the space of the possible test cases for the target program. A solution of the optimisation problem is a test case that executes the program path represented by the path condition. Here we formalise the objective functions and the related optimisation problems.

SUSHI translates a path condition PC into the objective function $\text{DISTANCE}_{PC}(t)$ that associates a test case t with a non-negative value d that evaluates to 0 if t satisfies *all* the clauses of the path condition, and to a positive integer that grows proportionally with the amount of clauses of the path condition that t does not satisfy. Thus, the objective function $\text{DISTANCE}_{PC}(t)$ defines the optimisation problem as the problem of finding a test case that satisfies the path condition, that is a test case on which the distance function evaluates to zero. The problem of optimising (minimising) the objective function is formally defined as:

$$\min_t \text{DISTANCE}_{PC}(t) = \sum_{i=1}^{k} \delta_{c_i}(t)$$

where the path condition PC is a conjunction of clauses $PC = c_1 \wedge c_2 \wedge \ldots \wedge c_k$, t is a test case, and $\delta_{c_i} \in [0, 1]$ quantifies the distance of t from the values that satisfy the clause c_i.

A minimum of the objective function is a test case for which all $\delta_{c_i}(t)$ evaluate to zero, and each $\delta_{c_i}(t)$ evaluates to zero only for test cases that satisfy the path condition clause c_i. Thus, the objective function evaluates to decreasing values for test cases that satisfy an increasing number of clauses.

$\delta_{c_i}(t)$ is computed by evaluating the clause c_i against the input state produced by executing the test case t as defined in Figure 6, which distinguishes three types of clauses:

- *numeric clauses* in the form $n_1 \bowtie n_2$, where n_1 and n_2 are arithmetic expressions over numeric input values and $\bowtie$ is some comparison operator,
- *reference equality clauses* in form $r_1 = r_2$, where r_1 and r_2 are input references, and
- *fresh object clauses* in the form $\textbf{\textit{fresh}}(r)$, where r is an input reference that must refer to a fresh object in the input state.

We now discuss the distance function in detail, by denoting as $t(exp)$ the evaluation of the expressions exp in the state produced by executing the test case t, where exp can be either a numeric expression or a reference.

Numeric clauses The distance $\delta_{n_1 \bowtie n_2}(t)$ yields 0 if the comparison $n_1 \bowtie n_2$ yields *true*. Otherwise, it is a positive value that corresponds to the absolute value of the difference between n_1 and n_2 normalised in the interval $[0, 1]$.

We add a small positive quantity ϵ to the difference between n_1 and n_2, to handle the case of strict inequality comparisons. For example, the clause $List.size > 3$ of Figure 3 is not satisfied by a test case t that builds a list with exactly 3 nodes, and this is reflected by the distance $\delta_{List.size>3}(t)$ that yields $(3 - List.size + \epsilon)/(1 + 3 - List.size + \epsilon) = \epsilon/(1 + \epsilon)$ which is a non-zero albeit small value.

When either or both n_1 and n_2 refer to input values that depend on objects that do not exist in the input state, yielding an indefinite value ($\perp$) when evaluated, the distance function conventionally evaluates to the maximal value 1.

This distance function favors test cases that better approximate the solutions of the path condition. This often yields a smooth objective function, ultimately improving the search of an optimal solution. For example, a test case t' that builds a list with a single item has a distance $\delta_{List.size>3}(t')$ strictly greater than a test case t'' that builds a list with two items, and therefore the search process will consider t'' over t', thus progressing towards the optimal solution.

Reference clauses The distance function $\delta_{r_1=r_2}(t)$ yields 0 if $t(r_1)$ and $t(r_2)$ refer to the same object (the two references alias each other) or are both *null*. Otherwise, the distance function yields 1, including when the evaluation of either r_1 or r_2 is undefined.

Fresh object clauses The distance function $\delta_{\textbf{\textit{fresh}}(r)}(t)$ addresses the satisfaction of clauses that predicate on the *freshness* of the object referred with r. The semantics of freshness depends on the lazy initialisation algorithm that initialises a symbolic reference to a *fresh* object if it assumes a reference to a (symbolically fresh) input object that is distinct from any other input object previously dereferenced.

For instance, the clause ***fresh****(List.header.previous)* in Figure 3 states the freshness of the reference *List.header.previous*. This clause is satisfied if:

$$t(List.header.previous) \neq t(List.header) \wedge$$
$$t(List.header.previous) \neq t(List) \wedge$$
$$t(List.header.previous) \neq t(Obj)$$

since the previous clauses in the path condition state that the references *List.header*, *List* and *Obj* point to other objects dereferenced at previous statements.

To evaluate freshness, we exploit the temporal order in which the symbolic executor generates the clauses in the path condition. We constrain the symbolic executor JBSE to produce path conditions in which a clause c_i precedes a clause c_j if and only if the clause c_i is assumed earlier than c_j. Under this condition, we evaluate the freshness of a clause c with respect to the objects already referred to in any other clause that precedes c in the path condition.

Figure 6 defines the distance function $\delta_{\textbf{\textit{fresh}}(r)}(t)$ referring to the set of the previously *referred objects* (referred_objects) in the prefix of the path condition. The distance function $\delta_{\textbf{\textit{fresh}}(r)}(t)$ yields 0 if r is defined and does not refer to any referred objects in previous clauses, 1 otherwise.

The two-value distance functions for clauses over references are smoothed by the objective function that sums the similarities of many different path condition clauses over the references in the input state.

$$\delta_{c_i}(t) = \begin{cases} \delta_{n_1 \bowtie n_2}(t) = \begin{cases} 0 & \text{if } t(n_1) \bowtie t(n_2) \\ 1 & \text{if } t(n_1) = \bot \text{ or } t(n_2) = \bot \\ 1 - \dfrac{1}{1 + |t(n_1) - t(n_2)| + \epsilon} & \text{otherwise} \end{cases} \\ \delta_{r_1 = r_2}(t) = \begin{cases} 0 & \text{if } t(r_1) = t(r_2) \\ 1 & \text{if } t(r_1) = \bot \text{ or } t(r_2) = \bot \text{ or } t(r_1) \neq t(r_2) \end{cases} \\ \delta_{\mathit{fresh}(r)}(t) = \begin{cases} 0 & \text{if } t(r) \notin \text{referred\_objects}(t)\langle c_1, \ldots, c_{i-1}\rangle \\ 1 & \text{if } t(r) = \bot \text{ or } t(r) \in \text{referred\_objects}(t)\langle c_1, \ldots, c_{i-1}\rangle \end{cases} \end{cases}$$

$$\text{referred\_objects}(t)\langle\rangle = \emptyset$$

$$\text{referred\_objects}(t)\langle c_1, \ldots, c_k\rangle = \begin{cases} \text{referred\_objects}(t)\langle c_1, \ldots, c_{k-1}\rangle \cup \{t(r)\} & \text{if } c_k = \mathit{fresh}(r) \\ \text{referred\_objects}(t)\langle c_1, \ldots, c_{k-1}\rangle & \text{otherwise} \end{cases}$$

Figure 6: $\delta_{c_i}(t)$: distance of a test t from the values that satisfy a clause c_i of a path condition

3.2 Prototype

Exploring all possible paths of a program is practically infeasible, and thus a practical implementation of SUSHI shall embed a strategy to select a set of *testing-relevant* path conditions. We have implemented a Java prototype that instantiates SUSHI to optimize branch coverage.[1] Our prototype symbolically executes the subject to identify a set of paths that cover as many branches as possible, and concretizes the corresponding path conditions to test cases.

We focus on branch testing to both limit the amount of paths to be explored, and to produce results comparable with competing techniques. The branch selection strategy is embedded in the *symbolic executor*. The prototype implementation of the core parts of our approach—the *search problem converter* and the *search engine*—are general, and the prototype can be adapted to any other selection strategy by simply adapting the symbolic executor.

Our implementation exploits the symbolic executor JBSE to 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 comprehensive set of techniques to account for structural invariants of the input data structures during symbolic execution [9, 10].

When JBSE terminates, the prototype processes the information on the branches traversed during the symbolic execution by identifying the minimal subset of paths that achieves maximum branch coverage, using a linear programming library,[2] and translates the corresponding path conditions into *evaluator programs* that compute the objective function discussed in Section 3.1.

The prototype then leverages the search-based test generator EvoSuite to build the test cases by using the evaluator programs of the selected path conditions as fitness functions. EvoSuite generates test cases by means of a genetic algorithm that randomly mutates and combines a population of generated test cases to create new solutions, aiming to progressively minimize the value returned by the evaluator program (the fitness function), until it eventually succeeds in generating a test case that makes the evaluator program return zero, that is, a test case that satisfies the given path condition.

Some path conditions can be spurious due to unsound computations during symbolic execution, and thus SUSHI cannot instantiate those path conditions into test cases. Our prototype iterates from the path selection phase to select a new set of path conditions that can execute the branches that were not covered yet, until a maximum time budget for the test search phase is reached.

[1] https://github.com/pietrobraione/sushi
[2] https://www.gnu.org/software/glpk/

4 EVALUATION

We experimentally evaluated our hypothesis that, by augmenting symbolic execution with a search-based engine, SUSHI effectively generates test cases for programs that depend on complex structured inputs in the form of sequences of method invocations. In particular, SUSHI is more effective than alternative approaches that combine symbolic execution with either direct heap manipulation or static analysis approaches. We present the results of the experiments that we conducted to answer three research questions:

RQ_1: Does SUSHI generate test cases more effectively than symbolic execution approaches with direct heap manipulation?

RQ_2: Does SUSHI generate test cases more effectively than symbolic execution approaches with static analysis?

RQ_3: To what extent do SUSHI test suites exercise program paths that depend on complex data structures as inputs?

We evaluated SUSHI by executing the prototype described in Section 3.2 on a set of benchmark programs, which work on complex structured inputs, and quantified the effectiveness of the test suites in terms of both *soundness* and *branch coverage*. We measure soundness as the ability of generating only valid test data, that is, test cases that execute reachable program states, and branch coverage as the fraction of executed program branches.

We address RQ_1 by comparing the tests generated with SUSHI and JBSE/DHM, respectively. JBSE/DHM extends the JBSE symbolic executor with direct heap manipulation. The results show that SUSHI outperforms JBSE/DHM in both soundness and coverage.

We address RQ_2 by comparing SUSHI with Pex and Seeker, which combine dynamic symbolic execution with different static analyses to synthesise test cases composed of legal method sequences. We found compelling evidence that SUSHI outperforms both Pex and Seeker in terms of branch coverage.

We address RQ_3 by inspecting the program branches executed only by SUSHI, and not executed by any of a representative sample of state-of-the-art test generators: Pex, Seeker, and EvoSuite. The results confirm the unique characteristics of SUSHI to identify and instantiate the data structures required to execute such branches.

4.1 Subject Programs

We considered a set of Java classes with paths that involve complex inter-procedural dependencies and result in complex conditions on input data structures during symbolic execution:

sample The sample program of Figure 1 with $N = 15$.
treemap An implementation of red-black trees [23].
avl An implementation of AVL trees [32].
caching An implementation of a doubly linked list, which caches node objects to improve efficiency [32].
tsafe Class *TrajectorySynthesizer* of the TSAFE prototype [21] that computes plane trajectories based on position and flight plan.
gantt Class *DependencyGraph* of the GanttProject software (commit `6d9a001`) that handles dependency edges for a Gantt graph.[3]
closure01 Class *RemoveUnusedVars* of Google Closure compiler that removes unused variables from parse trees.[4]
closure72 Class *RenameLabels* of the Google Closure compiler that processes the *LABEL* nodes out of parse trees.

The programs *tsafe*, *closure01*, and *closure72* include known faults that depend on non-trivial structured inputs. The fault in *tsafe* was revealed while verifying a set of correctness properties [9], whereas the faults in *closure01* and *closure72* are part of Defects4J, (bugs 1 and 72, respectively) [37].

Table 1 reports the subject programs (column *Subject*) with the number of branches of the class under analysis (column *Branches*), and the executable lines of code (column *LOC*) computed as the lines of code in both the class under analysis and the transitive closure of all its dependencies, which indicatively represent the complexity of code that the symbolic execution has to analyze.

4.2 Experimental Setup

We executed the SUSHI prototype against the subject programs, considering all public methods of the classes under analysis as entry points for the symbolic execution, and challenging SUSHI to generate test cases that cover the program branches. For each program, we configured the symbolic executor JBSE in SUSHI with the minimum bound values that produce the highest coverage, and encoded heap invariants in the HEX language [10].[5]

An independent variable of our experiments is the accuracy of the data structure invariants that we feed to the symbolic executor. The less accurate the invariants are, the more likely the symbolic executor produces spurious path conditions, which correspond to unreachable program states.

We considered two different configurations of the invariants of the input data structures: *partial* and *accurate* invariants. Partial invariants specify only a subset of the properties required to precisely symbolically analyse the subject programs, thus forcing SUSHI to cope with many spurious path conditions and waste time in trying to concretise them. Accurate invariants specify the properties that fully characterise the data structures to discard all spurious path conditions, thus challenging SUSHI to concretise valid path conditions for as many program branches as possible.

We selected partial invariants as common invariants of the basic data structures: lists and unbalanced trees. In particular, we referred to the doubly linked list invariants when applying SUSHI to the subjects *sample*, *caching*, *tsafe* and *gantt*, and we referred to the (unbalanced) binary and n-ary trees invariants when applying SUSHI to the subjects *treemap*, *avl*, *closure01* and *closure72*. The accurate invariants systematically account for the additional data structures and properties of each benchmark program. In particular, we specified the full invariants of caching linked lists, red-black trees, AVL trees, and compiler parse trees [10].

We augmented the JBSE symbolic executor with direct heap manipulation (JBSE/DHM) as follows: Given a path condition to be instantiated, JBSE/DHM queries the Z3 solver to solve the numeric constraints, and exploits the Java reflection APIs to set the object states according to the path condition.

We executed each experiment with a total time budget of two hours, with at most one hour allocated to symbolic execution. We set one-hour timeout for the symbolic execution phase in both SUSHI and JBSE/DHM to foster controlled experiments on identical sets of path conditions. We then allocated further 60 minutes for test generation to both tools. We ran all the experiments 10 times to gain statistical strength.

We executed the test generators Pex [51] (Visual Studio 2015) and Seeker [50] (v 1.0) against an equivalent C# porting of *sample*, *treemap*, *avl*, *caching*, *tsafe*, and *gantt*. We compared SUSHI with the best results achieved by executing Pex and Seeker with and without invariants encoded as invariant checking (repOk) methods. To comply with Pex requirements, we used invariant checking (repOk) methods equivalent to the HEX invariants used in JBSE.

We compared the branches covered by SUSHI, Pex, Seeker and EvoSuite [28] searching for evidence that SUSHI test suites execute branches hard to cover with state-of-the-art test generators,[6] and that the branches uniquely covered with SUSHI depend on complex data structures. To this end, we considered the best result of EvoSuite version 1.0.1 out of 10 runs in the default EvoSuite configuration that optimises branch coverage. We set a time bound of two hours that is twice larger than the time at which all tools saturate coverage, resulting in the best hand-tuning for all tools used in the experiments. Notice that we only compare coverage.

For all generated test suites, we measured branch coverage with the JaCoCo library,[7] and manually cross-checked whether the test cases reveal the known faults in the subjects *tsafe*, *closure01*, and *closure72*. We executed all experiments on a Ubuntu 16.04 machine equipped with 64 CPUs at 2.6 GHz and 512 GB of RAM.

4.3 RQ1/RQ2: SUSHI wrt Symbolic Approaches

Table 1 reports the number of tests generated by SUSHI and JBSE/DHM configured with partial and accurate invariants (columns *Test cases*), the branch coverage of the generate tests (columns *Branch coverage*), and the execution time required by SUSHI and JBSE/DHM to generate the tests as median over 10 runs (columns

[3] https://github.com/bardsoftware/ganttproject
[4] https://github.com/google/closure-compiler
[5] We observe that SUSHI is not sensitive to either the chosen symbolic executor, or the specification language used to express the data structure invariants. For instance, using the JPF symbolic executor with equivalent invariants encoded as repOk methods [53] would lead to identical path conditions and then identical coverage results in our experiments, though the performance of the symbolic execution phase may vary.

[6] We also experienced with Randoop [45] and EvoSuite with DSE [31], and they achieved worse coverage than standard EvoSuite on every subject. The tools Symstra [54] and Evacon [35] were not available at the time of the writing.
[7] http://eclemma.org/jacoco/

Table 1: Effectiveness of JBSE/DHM and SUSHI configured with partial (DHM^P, SUSHI^P) and accurate (DHM^A, SUSHI^A) data structure invariants (OK = valid outcomes, FP = False Positives)

Subjects			Test cases					Branch coverage					Generation time (mm:ss)			
			DHM^P		SUSHI^P	DHM^A	SUSHI^A	DHM^P		SUSHI^P	DHM^A	SUSHI^A	DHM^P	SUSHI^P	DHM^A	SUSHI^A
	Branches	LOC	OK	FP	OK	OK	OK	OK	FP	OK	OK	OK				
sample	2	8	2	0	2	2	2	2	0	2	2	2	00:01	00:06	00:01	00:06
treemap	164	514	21	9	50	33	33	101	48	138	*144	*144	01:47	‡120:00	00:13	05:26
avl	40	530	9	2	16	14	14	18	22	40	40	40	00:09	10:16	00:02	00:13
caching	36	211	9	1	15	10	10	25	6	*31	*31	*31	00:02	06:24	00:01	04:17
tsafe	36	607	§2	0	+12	§2	+10	5	0	*35	5	*35	04:33	05:49	00:44	03:11
gantt	26	404	3	2	5	8	8	5	4	16	*24	*24	†60:01	†60:55	00:08	01:28
closure01	279	7,766	3	0	4	§7	+13	11	0	11	45	104	†60:01	†66:13	14:05	39:27
closure72	23	5,972	0	3	+2	3	+3	0	#0	16	#0	18	00:28	‡120:00	00:03	20:33

+ Test suite reveals the known faults
§ Test cases for linear path conditions only
# Test cases throw runtime exceptions
* Missed branches are infeasible
† Symbolic execution terminated by timeout (1h)
‡ Test generation terminated by timeout (2h)

Generation time). Columns DHM^P, SUSHI^P, DHM^A, and SUSHI^A reports the results for JBSE/DHM and SUSHI with partial and accurate invariants, respectively.

Test suites with partial invariants (Table 1, columns DHM^P, SUSHI^P). When executed with partial invariants, SUSHI and JBSE/DHM must deal with sets of path conditions that may include spurious conditions, that is, path conditions that correspond to unreachable program states. In Table 1, columns *OK* report the number of valid test cases (respectively, branches), that is, test cases instantiated from satisfiable path conditions for both DHM^P and SUSHI^P. Columns *FP* report the number of false positives (respectively, branches), that is, test cases that correspond to spurious path conditions for DHM^P (SUSHI^P generates only valid test cases). We detected the false positives by manually inspecting the data structures instantiated in the tests cases. For example, for `treemap` we found 9 test cases that execute the program with *unbalanced* trees in input, thus violating the invariants of the data structure.

In these experiments, SUSHI^P largely outperformed DHM^P in soundness: DHM^P instantiates a test case for every selected path condition (both satisfiable and spurious), and thus generates false positives for 5 out of 8 subject programs (red values in column *Test cases*/DHM^P/FP of Table 1). SUSHI^P produces test cases by including only legal method calls, and thus does not generate false positives by construction, albeit wasting time searching for concrete test cases for the spurious conditions, as shown in columns *Generation time/DHM^P* and *Generation time/SUSHI^P*. The SUSHI^P performance penalties are paid back with sound results.

False positive test cases produce unsound coverage data that limit the practical relevance of the results. Column *Branch coverage*/DHM^P/FP of Table 1 reports in red the amount of false positive branches, that is, branches covered only with false positive test cases. For `treemap`, DHM^P reports a total amount of 149 covered branches (101 + 48) which is greater than the 144 feasible ones, as we deduced by inspecting the program.

SUSHI^P largely outperforms DHM^P in branch coverage too. SUSHI^P benefits from the soundness of the generated test cases to enhance branch coverage, while DHM^P suffers from false positives that obfuscate the validity of the coverage results. SUSHI^P generates only test cases that cover feasible path conditions, and thus incrementally considers path conditions to cover still missed branches, while DHM^P generates both valid and false positive test cases, thus missing the opportunity of covering branches that it wrongly considers as covered. Columns *Branch coverage*/DHM^P/OK and *Branch coverage*/SUSHI^P/OK indicate that SUSHI^P obtained a better branch coverage than DHM^P in 6 out of 8 subject programs.

Test suites with accurate invariants (Table 1, columns DHM^A, SUSHI^A). When executed with accurate invariants, both DHM^A and SUSHI^A work with the same non-spurious path conditions, and should thus produce test suites that obtain the same coverage. Surprisingly, our results (columns DHM^A, SUSHI^A) indicate subtle, but relevant differences for *tsafe*, *closure01* and *closure72*.

In *tsafe* and *closure01*, DHM^A generated fewer test cases than SUSHI^A, since DHM^A fails to instantiate path conditions with non-linear constraints not solved with Z3. These examples highlight a core limitation of direct heap manipulation to deal with path conditions that escape SMT theories. SUSHI robustly handles non-linear path conditions in the search phase. SUSHI^A generated 10 and 13 test cases that cover 35 and 104 branches in *tsafe* and *closure01*, respectively, while DHM^A correspondingly generated only 2 and 7 test cases that cover only 5 and 45 branches, respectively.

In *closure72*, DHM^A generated 3 test cases that assign values to program variables declared as constants. These test cases throw runtime exceptions and terminate before executing the target program. This example highlights another limitation of direct heap manipulation that can produce test cases that conflict with the compile time assumptions of the program under test.

Accurate invariants largely improve the performance of SUSHI^A over SUSHI^P, since SUSHI^A does not waste time with spurious path conditions. Although DHM^A may outperform SUSHI^A in efficiency, since SUSHI^A executes a search algorithms after symbolic execution, DHM^A is intrinsically less sound, since it generates tests that may raise exceptions, and cannot instantiate non-linear path conditions. Moreover, the degree of accuracy of the invariants may not be easily ascertained, thus jeopardising the trustability of the results of JBSE/DHM, but not the ones of SUSHI.

In summary, our experiments provide empirical evidence that SUSHI improves on approaches that combine symbolic execution with direct heap manipulation in both soundness and coverage.

Comparing SUSHI with Pex and Seeker. Table 2 compares SUSHI^P and SUSHI^A against the best results of Pex, Seeker, and EvoSuite. For each tool, the table reports the size of the generated test suites

Table 2: Comparative evaluation of SUSHI, Pex, Seeker, and EvoSuite

	SUSHI^{P}			SUSHI^{A}			Pex			Seeker			EvoSuite		
Subject Program	**Test cases**	**Branch coverage**		**Test cases**	**Branch coverage**		**Test cases**	**Branch coverage**		**Test cases**	**Branch coverage**		**Test cases**	**Branch coverage**	
sample	2	2	(100%)	2	2	(100%)	1	0	(0%)	2	0	(0%)	1	0	(0%)
treemap	50	138	(84%)	33	144	(88%)	10	11	(7%)	51	50	(25%)	30	144	(88%)
avl	16	40	(100%)	14	40	(100%)	11	25	(63%)	17	28	(70%)	14	40	(100%)
caching	15	31	(86%)	10	31	(86%)	9	11	(31%)	54	27	(75%)	8	29	(81%)
tsafe	$^{+}$12	35	(97%)	$^{+}$10	35	(97%)	10	13	(36%)	10	13	(36%)	7	24	(67%)
gantt	5	16	(62%)	8	24	(92%)	7	8	(31%)	7	8	(31%)	8	24	(92%)
closure01	4	11	(4%)	$^{+}$13	104	(37%)		-			-		7	43	(15%)
closure72	$^{+}$2	16	(70%)	$^{+}$3	18	(78%)		-			-		8	10	(43%)

$^{+}$ These test suites reveal the known fault in the corresponding subject programs.

(columns *Test cases*), and the target branches that each test suite exercises (columns *Branch coverage*). The grey background highlights the best result for each subject.

The first four columns of Table 2 report the results of the experiments conducted on the six subject programs for which we had an equivalent C# version. Both SUSHI^{P} and SUSHI^{A} generated test suites with better coverage than the test suites generated by Pex and Seeker: SUSHI^{P} generated 100 test cases that cover 95% of the 276 feasible branches of the six subjects, SUSHI^{A} generated 77 test cases that cover 100% of the feasible branches, Pex generated 48 test cases that cover 25% of the feasible branches, and Seeker generated 141 test cases that cover 46% of the feasible branches.

The test suites generated with Pex and Seeker suffer from the difficulty of static analysis in dealing with complex data structures. Pex and Seeker succeed in generating test cases for simple data structures but not for the complex ones, while SUSHI can effectively generate test cases that exercise a larger set of data structures.

Our experiments thus provide preliminary evidence that SUSHI generates test suites with a higher branch coverage than the test suites generated with two popular approaches that combine symbolic execution with static analysis.

4.4 RQ3: SUSHI wrt Complex Heap Inputs

We studied the data structures that uniquely characterise the test cases generated with SUSHI as follows: (i) We extended the benchmark of comparative approaches to EvoSuite to include a representative test case generators that exploit approaches beyond symbolic execution (column *EvoSuite* in Table 2), (ii) identified the branches covered solely by SUSHI test suites, and (iii) inspected in details the SUSHI test cases that executes those branches.

Table 2 shows that the SUSHI^{A} test suites achieve better branch coverage than all competitors for the subject programs, with EvoSuite reaching the same branch coverage for *treemap*, *avl*, and *gantt*.

We manually inspected the SUSHI^{A} test cases that traverse the branches covered solely by SUSHI and confirmed that SUSHI generates test cases that build the complex data structures required to exercise important portions of the programs. For example, the SUSHI^{P} and SUSHI^{A} test suites are the only suites that exercise the branches of *caching* for removing an element from a list with a full cache. Such branches require a test case that builds a list bigger than the cache size (more than 20 nodes), removes enough nodes to fill the cache (at least 20), and finally removes a further node that exercises the considered case. Only SUSHI^{P} and SUSHI^{A} generate test cases that instantiated such complex data structures, while none of the competing approaches do.

EvoSuite achieves the best branch coverage among the competing approaches, but still fails in generating test cases for branches that depend on input structures that are not explicitly captured in the branch conditions, e.g., the one in Figure 1. SUSHI^{A} achieves the same coverage of EvoSuite on *treemap*, *avl* and *gantt*, and outperforms EvoSuite on *sample*, *caching*, *tsafe*, *closure01* and *closure72*.

Overall SUSHI^{A} generated test suites that execute most executable branches and reveal the known faults. In detail, SUSHI^{A} generated test cases that cover all branches of *sample* and *avl* (100% branch coverage), and all feasible branches of *treemap*, *caching*, *tsafe*, and *gantt*. We manually verified that the uncovered branches correspond to either dead code or unreachable code elements. The SUSHI^{A} test cases cover 37% and 78% of the branches of *closure01* and *closure72*, respectively. We were not able to manually investigate the reachability of the uncovered branches, since the classes under test are coupled with too many other classes with non-trivial size and behaviour, thus the missed branches might either be unreachable branches or depend on paths and inputs that exceeded the symbolic execution bounds set in the experiments.

A closer examination of the experiment with *closure01* indicates that the low coverage achieved is strongly affected by the symbolic execution bounds indeed. We executed SUSHI^{A} on *closure01* with an upper bound of six nodes in the input parse trees, which makes the symbolic executor JBSE explore 86,141 execution paths in 14 minutes. When executed with larger upper bounds, JBSE does not terminate after several hours. This result indicates that the execution space of *closure01* is likely very large and deep, and that the JBSE depth-first path exploration may be a bottleneck for the analysis. Different exploration strategies, like the ones discussed by Cadar et al. [14], may improve the effectiveness of symbolic execution and lead to better results. Despite such limitation, SUSHI explores more branches in *closure01* than the competing tools, and is the only one that generates test cases that reveal the known fault.

A core technical contribution of this paper is the objective function (Section 3.1) that SUSHI uses to steer the generation of test cases that instantiate path conditions with concrete inputs. In the experiments with accurate invariants, SUSHI successfully found the optimal solution for all objective functions for the subject programs, converging consistently in all 10 re-executions within reasonable time budgets, thus indicating the effectiveness of the objective function. Column *Generation time*/SUSHI^{A} of Table 1 shows that in 6 out of 8 subjects the median time that SUSHI took to converge on

all path conditions is less than 6 minutes, while for *closure01* and *closure72* the median time is 54 and 21 minutes, respectively.

In summary, SUSHI can indeed instantiate path conditions that derive from the symbolic execution of programs with complex data structures, and thus generate concrete test cases for test objectives that depend on complex inputs and that exercise relevant behaviours in the programs. As for any other symbolic execution approach, the results of SUSHI depend on the path exploration strategy. Our evaluation provides additional evidence of the differences and the complementarity between symbolic execution and search-based software testing approaches [2, 22, 31, 35, 55].

4.5 Threats to validity

Threats to external validity may derive from the selection of case studies. Our results are obtained on a relatively limited sample of programs that provide some preliminary evidence of the feasibility of the approach, but are far from being a solid set of experiments with significant statistical validity. Nonetheless, the sample programs include commonly used data structures, and are representative of the class of programs that could mostly benefit from SUSHI, that is, programs that use complex data structures as input. We believe that our results clearly indicate a high potential of SUSHI to automate the generation of concrete test cases for these programs.

We limited our comparison to Pex, Seeker, and EvoSuite, in some cases due to the unavailability of other candidate tools. This threat is mitigated by the representativeness of the choices: two mature approaches, Pex and EvoSuite, and a tool that stresses the combination of symbolic execution and static analysis, Seeker. The comparison indicates the high potential of SUSHI.

Like all state-of-the-art symbolic execution approaches on data structures [10, 48, 53], the efficiency of SUSHI depends on the accuracy of the invariant specifications used. We reported on experiments with partial invariant specifications, where we assume that testers afford low or no cost for writing suboptimal invariants. Our results show that SUSHI does not generate invalid test cases, thus suffering less than other approaches when working with partial invariant. We also reported on experiments with accurate invariant specifications to investigate the ability of SUSHI of executing program paths that depend on complex input structures.

5 RELATED WORK

This paper presented SUSHI, a test case generation approach that benefits from a novel combination of symbolic execution and search-based techniques, to overcome a crucial limitation of many test generation approaches that rely on directly manipulating the heap memory to construct the input structures [1, 7, 33, 38, 48, 53]. Directly manipulating the heap may result in test cases that the developers can hardly understand and trust, especially when dealing with programs with complex ad-hoc data structures. These approaches can benefit from our SUSHI approach—provided suitable objective functions—to instantiate test cases composed of method sequences. This paper paves the way for future research in this direction.

In Section 2, we discussed the achievements and limitations of the main test generation approaches that rely solely on either symbolic execution or search-based techniques, but not both. For further details, readers can refer to the excellent surveys of Cadar et al. [14, 15] and McMinn [44]. Our SUSHI approach is a significant enhancement to test generation techniques based on symbolic execution, since SUSHI instantiates the path conditions into concrete test cases that initialise the input objects through their APIs. Our experiments suggest that SUSHI outperforms approaches that exploit static analysis to identify legal initialisation sequences [50, 51].

We now briefly survey the main techniques that combine symbolic execution and search-based software testing (SBST). Both Lakhotia et al. and Dinges and Agha exploit meta-heuristic search procedures to find numeric solutions for path conditions that include non-linear arithmetics and floating point variables that cannot be effectively handled with SMT solvers [22, 42]. SUSHI handles non-linear formulas in a similar fashion and extends the target domain to complex data structures.

Symstra and Evacon combine SBST and dynamic symbolic execution (DSE) to maximise branch coverage [35, 54]. They use standard SBST to produce method sequences, turn these method sequences into parametric test drivers by replacing their primitive inputs with symbolic variables, and then exploit DSE to find new relevant inputs. By construction, these approaches cannot execute program paths that depend on data structures that were not handled in the SBST step. Despite the difficulty of directly comparing SUSHI with Symstra and Evacon, due to their unavailability, our results related to EvoSuite suggest that identifying different initialisations of the data structures, as in SUSHI, can lead to significant complementary improvements with respect to just optimising the primitive inputs.

Another line of research exploits symbolic execution to optimise the evolutionary algorithms used in SBST. Baars et al. synthesise the execution conditions of sequences of branches, to improve the fitness function of SBST [2]. Malburg and Fraser, and Galeotti et al. extend search-based test generation with local search operators that apply DSE during the search procedure [31, 43]. Lakhotia et al. use symbolic execution to compute alias information for the local search [41]. Xie et al. and Baluda both apply search-based path selection strategies in symbolic executors to select the paths with higher chances to execute uncovered branches [3, 55].

6 CONCLUSIONS

Modern automated test case generation techniques fall short when the input requires non-trivial data structures. In this paper, we propose a new approach to automatically generate test cases for programs with complex data structures. We use symbolic execution to generate path conditions that precisely characterise the dependencies between the program paths and the input data structures. We then turn the path conditions into optimisation problems that we solve with search-based techniques. A solution of an optimisation problem is a test case that invokes a sequence of method calls that generate the required data structures. Our evaluation shows that the approach is effective in generating test cases for programs with complex data structures as inputs, outperforming current test generation techniques. We are currently working on widening our experimental scope to further consolidate the results.

ACKNOWLEDGMENTS

This work was supported by the GAUSS project (PRIN-MIUR-2015KWREMX) and by the EPSRC grant EP/N007166/1.

REFERENCES

[1] E. Albert, I. Cabanas, A. Flores-Montoya, M. Gómez-Zamalloa, and S. Gutierrez. jPET: An Automatic Test-Case Generator for Java. In *Proceedings of The Working Conference on Reverse Engineering*, WCRE '11, pages 441–442. IEEE Computer Society, 2011.

[2] A. Baars, M. Harman, Y. Hassoun, K. Lakhotia, P. McMinn, P. Tonella, and T. Vos. Symbolic search-based testing. In *Proceedings of the International Conference on Automated Software Engineering*, ASE '11, pages 53–62. IEEE Computer Society, 2011.

[3] M. Baluda. EvoSE: Evolutionary symbolic execution. In *Proceedings of the 6th International Workshop on Automating Test Case Design, Selection and Evaluation*, A-TEST '15, pages 16–19. ACM, 2015.

[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] M. Baluda, G. Denaro, and M. Pezzè. Bidirectional symbolic analysis for effective branch testing. *IEEE Transactions on Software Engineering*, 42(5):403–426, 2015.

[6] C. Barrett, M. Deters, L. Moura, A. Oliveras, and A. Stump. 6 years of SMT-COMP. *Journal of Automated Reasoning*, 50(3):243–277, 2013.

[7] C. Boyapati, S. Khurshid, and D. Marinov. Korat: Automated testing based on java predicates. In *Proceedings of the International Symposium on Software Testing and Analysis*, ISSTA '02, pages 123–133. ACM, 2002.

[8] P. Braione, G. Denaro, A. Mattavelli, M. Vivanti, and A. Muhammad. Software testing with code-based test generators: data and lessons learned from a case study with an industrial software component. *Software Quality Journal*, 22(2):311–333, June 2014.

[9] P. Braione, G. Denaro, and M. Pezzè. Enhancing symbolic execution with built-in term rewriting and constrained lazy initialization. In *Proceedings of the European Software Engineering Conference held jointly with the ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ESEC/FSE '13, pages 411–421. ACM, 2013.

[10] P. Braione, G. Denaro, and M. Pezzè. Symbolic execution of programs with heap inputs. In *Proceedings of the European Software Engineering Conference held jointly with the ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ESEC/FSE '15, pages 602–613, 2015.

[11] P. Braione, G. Denaro, and M. Pezzè. JBSE: a symbolic executor for java programs with complex heap inputs. In *Proceedings of the European Software Engineering Conference held jointly with the ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ESEC/FSE '16, pages 1018–1022, 2016.

[12] J. Burnim and K. Sen. Heuristics for scalable dynamic test generation. In *Proceedings of the International Conference on Automated Software Engineering*, pages 443–446. IEEE Computer Society, 2008.

[13] C. Cadar, D. Dunbar, and D. Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the Symposium on Operating Systems Design and Implementation*, OSDI '08, pages 209–224. USENIX Association, 2008.

[14] C. Cadar, P. Godefroid, S. Khurshid, C. S. Păsăreanu, K. Sen, N. Tillmann, and W. Visser. Symbolic execution for software testing in practice: Preliminary assessment. In *Proceedings of the International Conference on Software Engineering*, ICSE '11, pages 1066–1071. ACM, 2011.

[15] C. Cadar and K. Sen. Symbolic execution for software testing: Three decades later. *Communications of the ACM*, 56(2):82–90, Feb. 2013.

[16] C. Csallner and Y. Smaragdakis. Jcrasher: an automatic robustness tester for java. *Proceedings of the Symposium on Principles and Practice of Parallel Programming*, 34(11):1025–1050, Sept. 2004.

[17] L. Cseppento and Z. Micskei. Evaluating symbolic execution-based test tools. In *Proceedings of the International Conference on Software Testing, Verification and Validation*, ICST '10, pages 1–10. IEEE Computer Society, 2015.

[18] E. Daka, J. Campos, G. Fraser, J. Dorn, and W. Weimer. Modeling readability to improve unit tests. In *Proceedings of the European Software Engineering Conference held jointly with the ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ESEC/FSE 2015, pages 107–118. ACM, 2015.

[19] L. De Moura and N. Bjørner. Satisfiability modulo theories: Introduction and applications. *Communications of the ACM*, 54(9):69–77, 2011.

[20] X. Deng, J. Lee, and Robby. Bogor/Kiasan: A k-bounded symbolic execution for checking strong heap properties of open systems. In *Proceedings of the International Conference on Automated Software Engineering*, ASE '06, pages 157–166. ACM, 2006.

[21] G. D. Dennis. Tsafe: building a trusted computing base for air traffic control software. Master's thesis, Massachusetts Institute of Technology, 2003.

[22] P. Dinges and G. Agha. Solving complex path conditions through heuristic search on induced polytopes. In *Proceedings of the ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE '14, pages 425–436. ACM, 2014.

[23] H. Do, S. Elbaum, and G. Rothermel. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *Empirical Software Engineering*, 10(4):405–435, 2005.

[24] B. Dutertre. Yices 2.2. In *Proceedings of the International Conference on Computer Aided Verification*, CAV '2014, pages 737–744. Springer, 2014.

[25] J. Edvardsson. A survey on automatic test data generation. In *Proceedings of the Second Conference on Computer Science and Engineering*, pages 21–28. ECSEL, 1999.

[26] G. Fraser and A. Arcuri. 1600 faults in 100 projects: Automatically finding faults while achieving high coverage with evosuite. *Empirical Software Engineering*, 20(3):611–639, 2013.

[27] G. Fraser and A. Arcuri. Evosuite at the sbst 2013 tool competition. In *Proceedings of the 2013 IEEE Sixth International Conference on Software Testing, Verification and Validation Workshops*, ICSTW '13, pages 406–409. IEEE Computer Society, 2013.

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

[29] G. Fraser and A. Arcuri. Evosuite at the sbst 2015 tool competition. In *Proceedings of the International Workshop on Search-Based Software Testing*, SBST '15, pages 25–27. IEEE Computer Society, 2015.

[30] G. Fraser and A. Arcuri. Evosuite at the sbst 2016 tool competition. In *Proceedings of the International Workshop on Search-Based Software Testing*, SBST '16, pages 33–36. ACM, 2016.

[31] J. P. Galeotti, G. Fraser, and A. Arcuri. Improving search-based test suite generation with dynamic symbolic execution. In *Proceedings of the International Symposium on Software Reliability Engineering*, ISSRE '13. IEEE Computer Society, 2013.

[32] J. P. Galeotti, N. Rosner, C. G. López Pombo, and M. F. Frias. Analysis of invariants for efficient bounded verification. In *Proceedings of the International Symposium on Software Testing and Analysis*, ISSTA '10, pages 25–36. ACM, 2010.

[33] M. Gligoric, T. Gvero, V. Jagannath, S. Khurshid, V. Kuncak, and D. Marinov. Test generation through programming in UDITA. In *Proceedings of the International Conference on Software Engineering*, ICSE '10, pages 225–234. ACM, 2010.

[34] P. Godefroid, N. Klarlund, and K. Sen. Dart: directed automated random testing. In *Proceedings of the Conference on Programming Language Design and Implementation*, PLDI '05, pages 213–223. ACM, 2005.

[35] K. Inkumsah and T. Xie. Improving structural testing of object-oriented programs via integrating evolutionary testing and symbolic execution. In *Proceedings of the International Conference on Automated Software Engineering*, ASE '08, pages 297–306. IEEE Computer Society, 2008.

[36] S. Jha, R. Limaye, and S. A. Seshia. Beaver: Engineering an efficient smt solver for bit-vector arithmetic. In *Proceedings of the International Conference on Computer Aided Verification*, CAV '09, pages 668–674. Springer, 2009.

[37] R. Just, D. Jalali, and M. D. Ernst. Defects4J: A database of existing faults to enable controlled testing studies for Java programs. In *Proceedings of the International Symposium on Software Testing and Analysis*, ISSTA '14, pages 437–440. ACM, 2014.

[38] S. A. Khalek, G. Yang, L. Zhang, D. Marinov, and S. Khurshid. Testera: A tool for testing java programs using alloy specifications. In *Proceedings of the International Conference on Automated Software Engineering*, ASE '11, pages 608–611. IEEE Computer Society, 2011.

[39] S. Khurshid, C. S. Păsăreanu, and W. Visser. Generalized symbolic execution for model checking and testing. In *Proceedings of the International Conference on Tools and Algorithms for Construction and Analysis of Systems*, TACAS '03. Springer, 2003.

[40] B. Korel. Automated software test data generation. *IEEE Transactions on Software Engineering*, 16(8):870–879, 1990.

[41] K. Lakhotia, M. Harman, and P. McMinn. Handling dynamic data structures in search based testing. In *Proceedings of the conference on Genetic and Evolutionary Computation*, GECCO '08, pages 1759–1766. ACM, 2008.

[42] K. Lakhotia, N. Tillmann, M. Harman, and J. De Halleux. Flopsy: Search-based floating point constraint solving for symbolic execution. In *Proceedings of the International Conference on Testing Software and Systems*, ICTSS '10, pages 142–157. Springer, 2010.

[43] J. Malburg and G. Fraser. Combining search-based and constraint-based testing. In *Proceedings of the International Conference on Automated Software Engineering*, ASE '11, pages 436–439. IEEE Computer Society, 2011.

[44] P. McMinn. Search-based software test data generation: a survey. *Software Testing, Verification and Reliability*, 14:105–156, 2004.

[45] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball. Feedback-directed random test generation. In *Proceedings of the International Conference on Software Engineering*, ICSE '07, pages 75–84. ACM, 2007.

[46] R. Pargas, M. J. Harrold, and R. Peck. Test-data generation using genetic algorithms. *Software Testing, Verification and Reliability*, 9(4):263–282, 1999.

[47] M. Pezzè and M. Young. *Software Testing and Analysis: Process, Principles and Techniques*. Wiley, 2007.

[48] 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.

[49] K. Sen and G. Agha. CUTE and jCUTE: Concolic unit testing and explicit path model-checking tools. In *Proceedings of the International Conference on Computer Aided Verification*, CAV '06, pages 419–423. Springer, 2006.

[50] S. Thummalapenta, T. Xie, N. Tillmann, J. de Halleux, and Z. Su. Synthesizing method sequences for high-coverage testing. In *Proceedings of the Conference on Object-Oriented Programming Systems and Applications*, OOPSLA '11, pages 189–206. ACM, 2011.

[51] N. Tillmann and J. de Halleux. Pex: White box test generation for .NET. In *Proceedings of the International Conference on Tests and Proofs*, TAP '08, pages 134–153. Springer, 2008.

[52] P. Tonella. Evolutionary testing of classes. In *Proceedings of the International Symposium on Software Testing and Analysis*, ISSTA '04, pages 119–128. ACM, 2004.

[53] W. Visser, C. S. Păsăreanu, and S. Khurshid. Test input generation with java pathfinder. In *Proceedings of the International Symposium on Software Testing and Analysis*, ISSTA '04, pages 97–107. ACM, 2004.

[54] T. Xie, D. Marinov, W. Schulte, and D. Notkin. Symstra: a framework for generating object-oriented unit tests using symbolic execution. In *Proceedings of the International Conference on Tools and Algorithms for Construction and Analysis of Systems*, TACAS '05, pages 365–381. Springer, 2005.

[55] T. Xie, N. Tillmann, P. de Halleux, and W. Schulte. Fitness-guided path exploration in dynamic symbolic execution. In *Proceedings of the International Conference on Dependable Systems and Networks*, DSN '09, pages 359–368, 2009.