Efficient Synthesis of Method Call Sequences for Test Generation and Bounded Verification

Yunfan Zhang, Ruidong Zhu, Yingfei Xiong, Tao Xie
Key Laboratory of High Confidence Software Technologies, Ministry of Education (Peking University)
School of Computer Science, Peking University
Beijing, PR China
{zyf0726,zhurd,xiongyf,taoxie}@pku.edu.cn

ABSTRACT

Modern programs are usually heap-based, where the programs manipulate heap-based data structures to perform computations. In software engineering tasks such as test generation and bounded verification, we need to determine the existence of a *reachable* heap state that satisfies a given specification, or construct the heap state by a sequence of calls to the public methods. Given the huge space combined from the methods and their arguments, the existing approaches typically adopt static analysis or heuristic search to explore only a small part of search space in the hope of finding the target state and target call sequence early on. However, these approaches do not have satisfactory performance on many real-world complex methods and specifications. In this paper, we propose an efficient synthesis algorithm for method call sequences, including an offline procedure for exploring all reachable heap states within a scope, and an online procedure for generating a method call sequence from the explored heap states to satisfy the given specification. To improve the efficiency of state exploration, we introduce a notion of *abstract heap state* for compactly representing heap states of the same structure and propose a strategy of merging *structurally-isomorphic* states. The experimental results demonstrate that our approach substantially outperforms the baselines in both test generation and bounded verification.

确定存在

或者通过对公共方法的一系列调用来构造堆状态

一种有效的方法调用序列的合成算法

本文的任务：
测试生成+有界验证

CCS CONCEPTS

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

KEYWORDS

test generation, bounded verification, program synthesis, symbolic execution

ACM Reference Format:

Yunfan Zhang, Ruidong Zhu, Yingfei Xiong, Tao Xie. 2022. Efficient Synthesis of Method Call Sequences for Test Generation and Bounded Verification. In *37th IEEE/ACM International Conference on Automated Software Engineering (ASE '22), October 10–14, 2022, Rochester, MI, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3551349.3556951

1 INTRODUCTION

Modern programs are usually *heap-based*, where the programs manipulate heap-based data structures, such as linked lists and graphs, to perform computations. There are two major software engineering tasks (test generation and bounded verification) where we need to construct a target heap state that satisfies a given specification, or determine the existence of this target heap state. First, in the task of test generation, we may need to construct different heap states in unit tests before invoking a method, such that different paths of the target method could be reached or specific types of errors could be triggered. For example, to cover a specific path in a method for sorting a list, we may need to construct a linked list in the heap with two elements, where the first is larger than the second. Typically, we can obtain the specifications for reaching different paths or triggering errors by symbolic execution [4, 9, 29, 32, 34] or by manual writing of specifications [7, 21, 26, 27], and yet we still need to construct the heap states to satisfy the specifications. Second, in the task of bounded verification, we need to check whether all heap states concerned (within a user-provided finite scope) satisfy a given property. If the property is violated, the verifier [10, 13, 14, 18, 19, 32] generates a counterexample, i.e., a heap state violating the property.

对于测试生成和有界验证，需要构造一个满足规范状态的目标堆状态或者确定目标状态的存在性

Existing approaches [7, 9] face two major limitations when resorting to directly constructing heap states, by directly assigning values to the fields of the heap objects, where the values are usually inferred by a constraint solver [6, 12, 15, 22, 30]. First, modern object-oriented programming languages such as Java and C++ have accessibility control. Direct construction may require assignments to private fields, violating the accessibility rules and resulting in compilation errors. To overcome this issue, some existing approaches [9] utilize low-level language features such as the reflection APIs. However, tests generated in such a manner break encapsulation and are often unacceptable to the developers. Second, more importantly, direct construction may produce invalid (or unreachable) heap states that could never appear in normal program execution (e.g., a cyclic binary tree). To address this issue, some existing approaches [7, 9] require the user to provide representation invariants to verify the validity of a heap state. e.g., in the form of a `repOK()` Boolean function. However, manually writing accurate representation invariants is a laborious and sometimes complex task, while automatic inference of representation invariants shows some initial promise [24] but still remains as an open problem for real-world cases.

直接构造的缺点

Yingfei Xiong is the corresponding author.

ASE '22, October 10–14, 2022, Rochester, MI, USA
© 2022 Association for Computing Machinery.
ACM ISBN 978-1-4503-9475-8/22/10…$15.00
https://doi.org/10.1145/3551349.3556951

通过综合在数据结构类中对公共方法的一系列调用

To avoid the problem of direct construction, some other approaches [8, 28] resort to synthesizing a sequence of calls to the public methods in the data structure classes. Synthesizing call sequences overcomes the two limitations of direct construction: (1) the generated code does not violate the accessibility rules, and (2) public methods of data structures are often carefully implemented and lead to only valid heap states. A basic approach to synthesizing call sequences is to enumerate all possible call sequences, up to a limit of sequence length, and check whether a sequence leads to a heap state satisfying the specification. However, given the huge space combined from the methods and their arguments, it is infeasible to enumerate all method call sequences. To overcome this issue, some existing approaches adopt static analysis [28] or heuristic search [8] to explore only a small part of search space in the hope of reaching the target heap state early on. However, even the state-of-the-art approach, SUSHI [8], does not have satisfactory performance: for a simple specification shown in Figure 2, SUSHI fails to generate a method call sequence for constructing a heap state to satisfy the specification within 10 hours.

In this paper, we aim to develop an efficient synthesis algorithm for method call sequences with our key insight: a heap state can be separated into two parts: (1) the structure of the heap including the objects and their reference fields pointing to other objects, and (2) primitive values such as integers stored in the primitive fields of the objects. The space of the former is relatively small and can be enumerated. The space of the latter is large, but can be reasoned using a constraint solver. In this way, we can switch from the enumeration of the method call sequences to the enumeration of the heap structures, and use a constraint solver to fill the primitive values.

实验方法概述

Based on this insight, we first introduce the notion of *abstract heap state*, which preserves the heap structure but replaces all primitive values with symbolic variables and a constraint over the symbolic variables. Based on the notion of abstract heap state, we propose an offline algorithm to exhaustively explore reachable abstract heap states, including the following key components: (1) a procedure for exploring new abstract heap states based on symbolic execution of the data structure methods, and (2) a procedure for reducing redundant exploration by merging abstract heap states that are *structurally isomorphic*. We also maintain a graph structure named state transformation graph to record the process of state exploration. Finally, we build the algorithm for synthesizing method call sequences by using the preceding offline algorithm for enumerating abstract heap states and an online procedure for filling the symbolic variables.

实验结果概述

We have developed a prototype based on our proposed approach, and conduct experiments on a set of Java classes implementing complex data structures. For test generation, we compare our approach with SUSHI [8], the state-of-the-art test generator based on call sequence generation. The experimental results show that our approach is more than 100X faster than SUSHI for solving each test generation task, and achieves 20% more branches on the subject programs. For bounded verification, since there is no existing program verifier that exactly addresses our problem, we construct a baseline that extends a symbolic execution engine [32] to partially address our problem by writing driver programs [33]. The experimental results show that when the state space is large, our approach can still verify heap-based properties within 2 minutes, while the baseline cannot verify within 30 minutes.

```
class Node {
    private Node next;
    private int value;
    private Node(Node n, int v) {
        this.next = n; this.value = v;
    }
    public static Node create(int v, boolean b) {
        if (b == true)
            return new Node(null, v * 2 + 1);
        else return new Node(null, v * 2);
    }
    public Node getNext() { return this.next; }
    public int getValue() { return this.value; }
    public void addAfter(int v) {
        this.next = new Node(null, v);
    }
    public Node addBefore(int v) {
        return new Node(this, v);
    }
}
```

Figure 1: A sample Java class implementing a list node

In summary, this paper makes the following main contributions:

- We develop an efficient algorithm for exploring reachable heap states based on state abstraction and state merging.
- We develop an efficient algorithm for synthesizing method call sequences, by combining enumerative techniques and symbolic techniques.
- We implement a prototype and conduct experiments for evaluating the effectiveness and efficiency of our approach.

The rest of the paper is organized as follows. Section 2 presents an overview of our approach. Section 3 formalizes the problem of targeted heap state construction. Section 4 presents the main algorithms. Section 5 presents the evaluation results. Section 6 discusses the limitations of our approach. Section 7 presents the related work, and finally Section 8 concludes.

2 OVERVIEW

This section illustrates the workflow of our approach and the limitations of the SUSHI approach. Figure 1 shows a simple Java class implementing a node of a list-like data structure. The class Node has two private fields: a reference field next and a primitive field value. There are many public methods in this class, including a static factory method create, two getter methods getNext and getValue, and two instance methods addAfter and addBefore that construct a new node and link it after or before this node.

Now let us consider a test generation task where we would like to generate object instances of this Node class to satisfy a specification TEST in Figure 2, in the form of a Boolean function. The object instances are generated by calling the public methods of Node.

The SUSHI approach regards this task as an optimization problem whose goal is to find a sequence of method calls optimizing an objective function, and designs an objective function estimating the distance of the current call sequence from satisfying the specification. SUSHI then solves the optimization problem by applying a genetic algorithm. However, the specification shown in Figure 2

SUSHI的缺陷：搜索算法，将原问题视作最优化问题，难以解决数值类型的约束

```
boolean TEST(Node o) {
    return o.value - o.next.value == 100 &&
        o.next.value - o.next.next.value == 200 &&
        o.value + o.next.next.value == 800
}
```

C_1 : o_1 = create(125, false); C_3 : o_3 = o_2.addBefore(550);
C_2 : o_2 = o_1.addBefore(450); C_4 : **assert**(TEST(o_3));

Figure 2: A specification TEST and a solution C

involves complex numeric constraints that are hard to be solved by just searching. Therefore, SUSHI performs inefficiently and fails to generate an expected method call sequence within 10 hours.

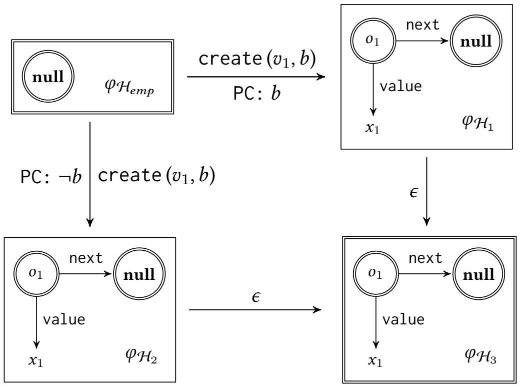
我们的方法：
1、搜索可达的堆状态
2、调用方法序列的合成

抽象堆状态包含：对象、符号变量
对符号变量的一阶状态约束

Next, we explain how our approach addresses the test generation task. There are two phases in the workflow of our approach: exploration of reachable heap states, and synthesis of method call sequences. Due to the huge space of reachable heap states, we introduce the notion of *abstract heap state* to compactly represents multiple concrete heap states with the same structure. An abstract heap state consists of a set of objects, a set of symbolic variables, and a first-order state constraint over the symbolic variables. Each object has a set of reference fields storing references to other objects and a set of primitive fields storing references to symbolic variables. There always exists a special object null containing no field. For example, in the abstract state $\mathcal{H}_1$ shown in Figure 3, the primitive field o_1.value stores a reference to variable x_1, and the possible values of x_1 are characterized by the state constraint $\varphi_{\mathcal{H}_1}$.

1、第一阶段
搜索可达的堆状态

In the first phase, we exhaustively explore the abstract heap states within the pre-specified finite scope, and record the exploration in a data structure called *state transformation graph*. This phase can be offline since the data structure classes are usually from a library and the exploration procedure does not depend on the specification. We start with the empty state $\mathcal{H}_{emp}$, and iteratively explore new abstract heap states by calling the public methods to transform the explored abstract states. Each time we first pick an explored state, and then pick a public method where its receiver object and the object parameters could be assigned from the objects in the state. Since there is no non-null heap object in $\mathcal{H}_{emp}$, the only method that we could call is the static factory method create. We generate two symbolic variables for its primitive parameters, and symbolically execute create over the abstract state $\mathcal{H}_{emp}$. There are two control flow paths in the method create, so there are two succeeding abstract states of $\mathcal{H}_{emp}$ – one is $\mathcal{H}_1$ for executing the then-branch, and the other is $\mathcal{H}_2$ for executing the else-branch. The state constraint of the new state is formed by conjoining the constraint of the original state and the path constraint. However, since by definition a primitive field of an abstract heap state stores only references to symbolic variables and the state constraint constrains only these variables, we make necessary changes to the state constraint by introducing new variables, existential quantifications, and equality constraints, resulting in $\varphi_{\mathcal{H}_1}$ and $\varphi_{\mathcal{H}_2}$.



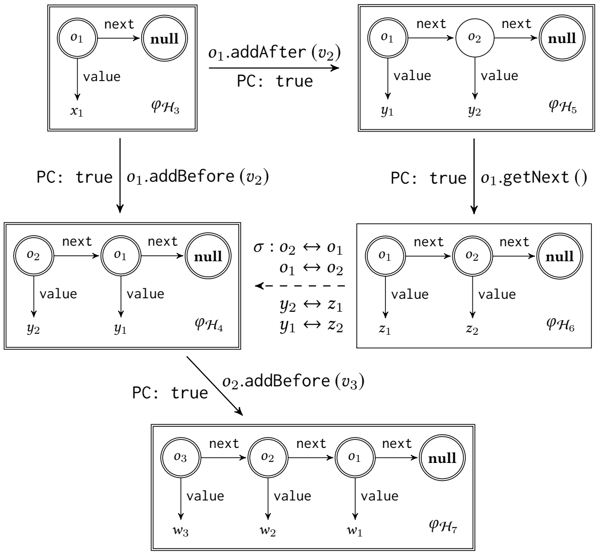
$$\varphi_{\mathcal{H}_{emp}} = \texttt{true}$$
$$\varphi_{\mathcal{H}_1} = \varphi_{\mathcal{H}_{emp}} \wedge \exists v_1. \exists b. (b \wedge x_1 = 2v_1 + 1)$$
$$\varphi_{\mathcal{H}_2} = \varphi_{\mathcal{H}_{emp}} \wedge \exists v_1. \exists b. (\neg b \wedge x_1 = 2v_1)$$
$$\varphi_{\mathcal{H}_3} = \varphi_{\mathcal{H}_1} \vee \varphi_{\mathcal{H}_2}$$

Figure 3: Calling method create on the abstract state $\mathcal{H}_{emp}$

Once a new state is reached, we check whether the graph structure formed by its objects and their reference fields is isomorphic to an existing state (called *structural isomorphism* between the two states). Here $\mathcal{H}_1$ and $\mathcal{H}_2$ are structurally isomorphic. Then we create a new state $\mathcal{H}_3$ to represent the state merged from $\mathcal{H}_1$ and $\mathcal{H}_2$. jects and their fields in $\mathcal{H}_3$ are the same as one of the state merged, and the constraint in $\mathcal{H}_3$ is a disjunction of the con- s in $\mathcal{H}_1$ and $\mathcal{H}_2$. The set of concrete heap states represented by $\mathcal{H}_3$ is the union set of concrete states represented by $\mathcal{H}_1$ and $\mathcal{H}_2$. Next, two special ϵ edges from the original states to the merge state are added. Finally, we mark $\mathcal{H}_1$ and $\mathcal{H}_2$ as *inactive*. An inactive state would not be selected for state exploration or isomorphism comparison. We use single-lined frame and double-lined frame to distinguish between inactive and active states in the figures.

$\mathcal{H}_3$, two new methods can be called, addBefore and addAfter, obtain two new abstract states $\mathcal{H}_4$ and $\mathcal{H}_5$, as shown in 4. Method addAfter creates a new object, but this object eturned and thus is not directly accessible from the stack. To distinguish such objects, we call these objects that are returned from a method call *stack-accessible* as we can own a reference on the stack that refers to the object. We denote stack-accessible objects with double circles and the other objects with single circles. The structural isomorphism also takes stack-accessibility into consideration: stack-accessible objects could only map to stack-accessible objects in an isomorphism. Therefore, states $\mathcal{H}_4$ and $\mathcal{H}_5$ are not isomorphic. Please note that $\varphi_{\mathcal{H}_3}$ contains a free variable x_1 and thus we introduce new existential quantification in $\varphi_{\mathcal{H}_4}$ and $\varphi_{\mathcal{H}_5}$.

Now suppose we call method getNext on the object o_1 in $\mathcal{H}_5$, and result in the abstract state $\mathcal{H}_6$. The abstract states $\mathcal{H}_4$ and $\mathcal{H}_6$ are structurally isomorphic with regard to the bijection σ and could be merged. However, we observe that $\varphi_{\mathcal{H}_6} \rightarrow \varphi_{\mathcal{H}_4}[y_2 := z_1, y_1 := z_2]$ holds for all z_1, z_2, i.e, all concrete heap states represented by $\mathcal{H}_6$ are included in those represented by $\mathcal{H}_4$. Therefore, we do not need to merge the two states but can simply mark $\mathcal{H}_6$ as inactive. We use a dotted arrow in Figure 4 to represent this subsumption relation between $\mathcal{H}_4$ and $\mathcal{H}_6$, but please note this arrow is not part of the state transformation graph. We perform this check with a constraint solver before merging states. We may further call method addBefore on object o_2 in $\mathcal{H}_4$ to obtain $\mathcal{H}_7$.


o_1
next
null
value
x_1
$\varphi_{\mathcal{H}_3}$
o_1.addAfter (v_2)
PC: true
o_1 o_2 null
y_1 y_2
$\varphi_{\mathcal{H}_5}$
PC: true o_1.addBefore (v_2)
PC: true o_1.getNext ()
o_2 o_1 null
y_2 y_1
$\varphi_{\mathcal{H}_4}$
$\sigma : o_2 \leftrightarrow o_1$
$o_1 \leftrightarrow o_2$
$y_2 \leftrightarrow z_1$
$y_1 \leftrightarrow z_2$
o_1 o_2 null
z_1 z_2
$\varphi_{\mathcal{H}_6}$
PC: true o_2.addBefore (v_3)
o_3 o_2 o_1 null
w_3 w_2 w_1
$\varphi_{\mathcal{H}_7}$

$$\varphi_{\mathcal{H}_4} = \exists x_1.\ \big(\varphi_{\mathcal{H}_3} \wedge \exists v_2. (y_1 = x_1 \wedge y_2 = v_2)\big)$$
$$\varphi_{\mathcal{H}_5} = \exists x_1.\ \big(\varphi_{\mathcal{H}_3} \wedge \exists v_2. (y_1 = x_1 \wedge y_2 = v_2)\big)$$
$$\varphi_{\mathcal{H}_6} = \exists y_1. \exists y_2.\ \big(\varphi_{\mathcal{H}_5} \wedge z_1 = y_1 \wedge z_2 = y_2\big)$$
$$\varphi_{\mathcal{H}_7} = \exists y_1. \exists y_2.\ \big(\varphi_{\mathcal{H}_4} \wedge \exists v_3. (w_1 = y_1 \wedge w_2 = y_2 \wedge w_3 = v_3)\big)$$

Figure 4: Further State Exploration

After the state transformation graph is built, we enumerate all permutations of stack-accessible objects in all states to determine whether the objects could form a sequence of arguments to make the specification pass. We illustrate the process with object o_3 in $\mathcal{H}_7$, i.e., check whether Test(o_3) returns true. Since there are symbolic variables in the abstract state, we need to *symbolically* execute the Boolean function Test with the argument o_3, and obtain the path condition α for satisfying the specification.

$$\alpha :\ (w_3 - w_2 = 100) \wedge (w_2 - w_1 = 200) \wedge (w_3 + w_1 = 800)$$

After solving the conjunction of the state constraint $\varphi_{\mathcal{H}_7}$ and the path condition α, we can obtain the target concrete heap state H by replacing the symbolic variables w_1, w_2 and w_3 in $\mathcal{H}_7$ with suitable concrete values – 250, 450, and 550, respectively.

2、第二阶段：合成一系列的方法调用来导致目标的具体状态

At last, we need to synthesize a sequence of method calls that leads to the target concrete state H. This process is non-trivial because there may be multiple paths that can reach the final abstract state, each corresponding to a method sequence, and we do not know which path could produce H. To find the path, we utilize the model (assignments to variables) obtained from solving the final constraint $\varphi_{\mathcal{H}_7} \wedge \alpha$, which represents H. We further notice that $\varphi_{\mathcal{H}_7}$ contains all symbolic variables (possibly under existential quantification) used in previous constraints, and thus the model contains assignments to all these variables and can be converted to a concrete state for any abstract state in the path leading to H. As a result, we search backwardly from the target state. For any state, we check whether the model satisfies the state constraint of any predecessor state and the path condition of corresponding edge. If so, we choose this state and search its predecessors. We repeat this process until we reach the initial state. In this example, we may obtain the following path, backwardly: $\mathcal{H}_7 \leftarrow \mathcal{H}_4 \leftarrow \mathcal{H}_3 \leftarrow \mathcal{H}_2 \leftarrow \mathcal{H}_{emp}$. Finally, we generate the method calls by following this path. Please note the arguments to the method calls are also included in the model. Figure 2 shows the synthesized method sequence.

3 PROBLEM FORMULATION

In this section, we introduce the terminology that we use in this paper and formalize the problem of targeted heap state construction.

Given a set of classes under test, we denote $\mathcal{M}$ as the set of public methods and $\mathcal{F}$ as the set of fields in the classes. The set of all primitive values is denoted as $\mathcal{V}$.

Definition 3.1 ((concrete) heap state). A (concrete) heap state $H = (O_H, AO_H, \delta_H)$ is a 3-tuple where:

- $O_H \supseteq \{\texttt{null}\}$ is a set of (reachable) heap objects;
- AO_H is a set of stack-accessible objects, where $\{\texttt{null}\} \subseteq AO_H \subseteq O_H$;
- $\delta_H : (O_H \times \mathcal{F}) \to (O_H \cup \mathcal{V})$ is a mapping that maps a field of a heap object to a value, which is either a reference to another heap object or a primitive value such as an integer.

A heap object is called stack-accessible, if we own a reference on the stack that refers to the object, so that we can access the object by this reference. In other words, if a sequence of method calls lead to a heap state H, all the stack-accessible objects $o \in AO_H$ except the special null object are return values of the method calls. In a heap state, the heap objects must be reachable from the stack-accessible objects (otherwise they would be garbage-collected). From this perspective, the stack-accessible objects can be also regarded as "root" objects. The empty heap state, which contains a single null object, is denoted as H_{emp}.

Definition 3.2 (method call). A method call on a heap state H is a 3-tuple $(m, \overline{o}, \overline{a})$ where $m \in \mathcal{M}$ is a method, $\overline{o}$ is a list of object arguments $o \in AO_H$, and $\overline{a}$ is a list of primitive arguments $a \in \mathcal{V}$.

Note that only stack-accessible objects can serve as the arguments of a method call. This definition generically models calls to instance methods, static methods, and constructors. For calls to instance methods, the first argument is a non-null receiver object. After calling a method, the pre-state H will be transformed into a post-state H', which can be different from the pre-state H with regard to the following aspects:

(1) some objects are allocated ($O_{H'} \backslash O_H$), and some objects are freed ($O_H \backslash O_{H'}$) because they are no longer reachable from the stack-accessible objects $AO_{H'}$ (garbage collection);
(2) an object that is the return value of the method call becomes stack-accessible, i.e., $AO_H \subseteq AO_{H'}$ and $AO_{H'} \backslash AO_H$ is either an empty set or a singleton set containing the return value;
(3) $\delta_{H'}$ differs from δ_H because some field values are modified.

Definition 3.3 (state transformation). If executing a method call $c = (m, \overline{o}, \overline{a})$ on a pre-state H would result in a post-state H', we say that (H, c, H') is a state transformation and denote it as $H \xrightarrow{c} H'$.

Definition 3.4 (reachable heap state). For two heap states H and H', we say that H' is reachable from H if there exists a sequence of heap states $H_0, \ldots, H_n$ and a sequence of method calls $c_1, \ldots, c_n$

such that the initial state $H_0 = H$, the final state $H_n = H'$, and $H_{k-1} \xrightarrow{c_k} H_k$ for all $k = 1, \ldots, n$.

Definition 3.5 (specification). A specification Φ is a Boolean function (implemented as a program), and a heap state H satisfies the specification Φ if there exists a list of heap objects $o \in O_H$ and a list of primitive values $a \in \mathcal{V}$ such that $\Phi(\overline{o}, \overline{a})$ returns `true`.

Based on the notion of reachable heap state and specification, we now define the problem of targeted heap state construction.

Definition 3.6 (targeted heap state construction). Given a specification Φ, the problem of targeted heap state construction is to construct or determine the inexistence of a heap state H that is reachable from H_{emp} and satisfies the specification Φ.

4 ALGORITHM

主要算法的介绍

This section presents our main algorithm to address the problem of targeted heap state construction. 1 First, we introduce the notion of abstract heap state and briefly describe the process of symbolic execution on abstract states. 2 Next, we present our algorithm for exploring all reachable heap states within a user-specified scope and building the state transformation graph. 3 Finally, we discuss how to extract a sequence of method calls from the offline-built state transformation graph, so that the call sequence constructs a target heap state satisfying the given specification.

S1：在抽象状态上的符号执行

4.1 Symbolic Execution on Abstract States

4.1.1 Abstract Heap States. In order to compactly represent many heap states with the same structure, we introduce the notion of *abstract heap state*, which replaces all primitive values in the heap state with symbolic variables and a constraint over these variables.

Definition 4.1 (abstract heap state). An abstract heap state $\mathcal{H}$ is a 5-tuple $(O_{\mathcal{H}}, AO_{\mathcal{H}}, Var_{\mathcal{H}}, \delta_{\mathcal{H}}, \varphi_{\mathcal{H}})$ where:

- $O_{\mathcal{H}}$ and $AO_{\mathcal{H}}$ are sets of all heap objects and stack-accessible heap objects, respectively;
- $Var_{\mathcal{H}}$ is a set of symbolic variables;
- $\delta_{\mathcal{H}} : (O_{\mathcal{H}} \times \mathcal{F}) \to (O_{\mathcal{H}} \cup Var_{\mathcal{H}})$ is a mapping that maps a field of a heap object to an abstract value, which is a reference to either another heap object or a symbolic variable;
- $\varphi_{\mathcal{H}}$ is a first-order constraint with existential quantification, where the free variables in $\varphi_{\mathcal{H}}$ are the variables in $Var_{\mathcal{H}}$.

Definition 4.2 (instance). Given a (concrete) heap state H and an abstract heap state $\mathcal{H}$, we say that H is an *instance* of $\mathcal{H}$ if it meets the following conditions:

(1) O_H and AO_H are identical to $O_{\mathcal{H}}$ and $AO_{\mathcal{H}}$, respectively;
(2) there exists an assignment $\pi : Var_{\mathcal{H}} \to \mathcal{V}$ such that $\varphi_{\mathcal{H}}$ is satisfied under the assignment π;
(3) for all objects $o \in O_H$ and fields $f \in \mathcal{F}$, it holds that:
   - $\delta_H(o, f) = o'$ if $\delta_{\mathcal{H}}(o, f) = o' \in O_H$, or
   - $\delta_H(o, f) = \pi(v) \in \mathcal{V}$ if $\delta_{\mathcal{H}}(o, f) = v \in Var_{\mathcal{H}}$.

The heap state instantiated by abstract heap state $\mathcal{H}$ and assignment π is denoted as $\mathcal{H}[\pi]$, and the set containing all instances of $\mathcal{H}$ is denoted as $\mathbf{Inst}(\mathcal{H}) = \{\mathcal{H}[\pi] : \pi \models \varphi_{\mathcal{H}}\}$.

To avoid redundant state exploration, we deal with *structurally isomorphic* abstract states simultaneously by merging them into a single *union* abstract state, such that the instances of the union abstract state are *equivalent* to the union set of instances of the original abstract states. The definition of (concrete) state equivalence and structural isomorphism is shown as follows.

Definition 4.3 (state equivalence). We say that concrete state H_1 is *equivalent* to concrete state H_2 with regard to a bijection $\sigma : O_{H_1} \to O_{H_2}$, denoted as $H_1 \equiv H_2$, if for all objects $o \in O_{H_1}$ and fields $f \in \mathcal{F}$, it holds that

状态相等

- $o \in AO_{H_1}$ iff $\sigma(o) \in AO_{H_2}$, and $\sigma(\texttt{null}) = \texttt{null}$;
- $\delta_{H_1}(o, f) = o' \in O_{H_1}$ iff $\delta_{H_2}(\sigma(o), f) = \sigma(o') \in O_{H_2}$,
- $\delta_{H_1}(o, f) = a \in \mathcal{V}$ iff $\delta_{H_2}(\sigma(o), f) = a \in \mathcal{V}$.

Definition 4.4 (structural isomorphism). For two abstract heap states $\mathcal{H}_1$ and $\mathcal{H}_2$, we say that $\mathcal{H}_1$ is *structurally isomorphic* to $\mathcal{H}_2$ with regard to a bijection $\sigma : (O_{\mathcal{H}_1} \cup Var_{\mathcal{H}_1}) \to (O_{\mathcal{H}_2} \cup Var_{\mathcal{H}_2})$, if for all objects $o \in O_{\mathcal{H}_1}$ and fields $f \in \mathcal{F}$, it holds that:

同构

- $o \in AO_{\mathcal{H}_1}$ iff $\sigma(o) \in AO_{\mathcal{H}_2}$, and $\sigma(\texttt{null}) = \texttt{null}$;
- $\delta_{\mathcal{H}_1}(o, f) = o' \in O_{\mathcal{H}_1}$ iff $\delta_{\mathcal{H}_2}(\sigma(o), f) = \sigma(o') \in O_{\mathcal{H}_2}$;
- $\delta_{\mathcal{H}_1}(o, f) = v \in Var_{\mathcal{H}_1}$ iff $\delta_{\mathcal{H}_2}(\sigma(o), f) = \sigma(v) \in Var_{\mathcal{H}_2}$.

For two isomorphic abstract states $\mathcal{H}_1$ and $\mathcal{H}_2$ with regard to bijection σ, their union abstract state $\mathcal{H} = \mathbf{merge}_{\sigma}(\mathcal{H}_1, \mathcal{H}_2)$ can be obtained by creating a new abstract state $\mathcal{H}$ identical to $\mathcal{H}_2$ but only the state constraint $\varphi_{\mathcal{H}}$ to be the disjunction of $\varphi_{\mathcal{H}_1}[v := \sigma(v)]$ and $\varphi_{\mathcal{H}_2}$. The notation $\varphi_{\mathcal{H}_1}[v := \sigma(v)]$ indicates substitution of $\sigma(v)$ for free variable $v \in Var_{\mathcal{H}_1}$ in formula $\varphi_{\mathcal{H}_1}$. For example, assume that $\varphi_{\mathcal{H}_1}$ is $\exists x. u = 2x$ while $\varphi_{\mathcal{H}_2}$ is $\exists y. v = 2y + 1$, and the bijection σ maps variable u to v. The constraint $\varphi_{\mathcal{H}}$ of the union abstract state can be computed as follows:

$$\varphi_{\mathcal{H}} = \varphi_{\mathcal{H}_1}[u := v] \vee \varphi_{\mathcal{H}_2} = (\exists x. v = 2x) \vee (\exists y. v = 2y + 1)$$

4.1.2 Symbolic Execution. Since we introduce the notion of abstract heap state, we need to know how to obtain the abstract post-state of executing an (abstract) method call on an abstract pre-state. This problem can be solved by means of *symbolic execution.*

Symbolic execution is a program analysis technique that determines what inputs would cause each control flow path of a program to execute. Symbolic execution engines regard the inputs of a program as symbolic variables, explore multiple paths simultaneously, and maintain for each explored path: (1) a *path condition* that describes the conditions satisfied by the branches taken along this path, and (2) a *symbolic store* that maps variables to symbolic expressions or values, which is updated by assignments [5].

Definition 4.5 (abstract method call). An abstract method call on an abstract heap state $\mathcal{H}$ is a 3-tuple $(m, \overline{o}, \overline{x})$ where $m \in \mathcal{M}$ is a method, $\overline{o}$ is a list of object arguments $o \in AO_{\mathcal{H}}$, and $\overline{x}$ is a list of different symbolic variables.

Formally, the result of symbolic execution for an abstract method call $c = (m, \overline{o}, \overline{x})$ on an abstract pre-state $\mathcal{H}$ is a set of path descriptors $(\alpha_p, O_p, r_p, \tau_p)$, where for each path p:

(1) α_p is the path condition, which is a constraint over the arguments x and the variables $v \in Var_{\mathcal{H}}$;
(2) O_p is the set of all heap objects when execution terminates;
(3) r_p is the return value, which is $\perp$ for no return value, a heap object $o \in O_p$, or a symbolic expression over x and $v \in Var_{\mathcal{H}}$;

(4) τ_p is the symbolic store, which maps fields $f \in \mathcal{F}$ of heap objects $o \in O_p$ to other heap objects (for reference fields) or symbolic expressions (for primitive fields).

For each path p, the abstract post-state $\mathcal{H}' = \textbf{PostState}(\mathcal{H}, \mathscr{c}, p)$ can be constructed as follows, according to the result of symbolic exeuction:

(1) $O_{\mathcal{H}'} = O_p$ containing all existent heap objects;
(2) $AO_{\mathcal{H}'} = AO_{\mathcal{H}} \cup \{r_p\}$ if the return value $r_p \in O_p$ is a heap object, otherwise $AO_{\mathcal{H}'} = AO_{\mathcal{H}}$;
(3) for all objects $o \in O_p$ and fields $f \in \mathcal{F}$:
  - $\delta_{\mathcal{H}'}(o, f) = o'$ if $\tau_p(o, f) = o' \in O_p$ is an object, or
  - $\delta_{\mathcal{H}'}(o, f) = v_{o,f}$ if $\tau_p(o, f)$ is an expression, where $v_{o,f}$ is a fresh variable;
(4) $Var_{\mathcal{H}'} = \{v_{o,f} : \tau_p(o, f) \notin O_p\}$ containing all fresh variables;
(5) $\varphi_{\mathcal{H}'} = \exists \bar{v}. \left(\varphi_{\mathcal{H}} \land \exists \bar{x}. \left(\alpha_p \land \bigwedge_{v_{o,f} \in Var_{\mathcal{H}'}} v_{o,f} = \tau_p(o, f)\right)\right)$ formed by introducing existential quantification on the variables $v \in Var_{\mathcal{H}}$ and the call arguments x, and conjoining the constraint $\varphi_{\mathcal{H}}$ of the pre-state $\mathcal{H}$, the path condition α_p, and a set of equality constraints that characterizes the expected values of the variables $v_{o,f} \in Var_{\mathcal{H}'}$.

Note that the non-object return value of a method call is nonessential, since we can always use a literal value to replace it.

The connection between execution on concrete heap states and symbolic execution on abstract heap heaps is depicted in the following lemma, where the notation $\mathscr{c}[\pi]$ indicates substitution of primitive value $\pi(x)$ for symbolic arguments x in $\mathscr{c}$.

Lemma 4.6 (Symbolic execution on abstract states). *Given a path p of an abstract method call $\mathscr{c}$, an abstract pre-state $\mathcal{H}$, and an abstract post-state $\mathcal{H}' = \textbf{PostState}(\mathcal{H}, \mathscr{c}, p)$, we hold that*

$$\textbf{Inst}(\mathcal{H}') = \{H' : \pi \models \varphi_{\mathcal{H}} \land \pi \models \alpha_p \land \mathcal{H}[\pi] \xrightarrow{\mathscr{c}[\pi]} H'\}$$

S2:
探索可到达的堆堆状态

4.2 Exploring Reachable Heap States

Based on abstract heap state, we exhaustively explore all reachable heap states within a user-specified scope in a breadth-first manner. At each iteration, we pick an explored abstract state, enumerate all candidate abstract method calls on the abstract state, and obtain new abstract heap states by means of symbolic execution. If the new abstract state $\mathcal{H}'$ is structurally isomorphic to a pre-explored abstract state $\mathcal{H}$, then we will construct a union abstract state in place of $\mathcal{H}$ and $\mathcal{H}'$ for further exploration. We also maintain a graph structure called state transformation graph to record the process of state exploration, so that we can recover a method call sequence for the reachable heap state. Each vertex in this graph represents an abstract heap state, and each (directed) edge represents either an operation of state merging, or a control flow path of an abstract method call, as shown in Figure 3.

The pseudo-code of our algorithm for exploring reachable heap states and building the state transformation graph is shown in Algorithm 1. Starting with the graph containing a single abstract heap state $\mathcal{H}_{emp}$ that represents the empty heap state (Line 1-2), we iteratively expand the graph by invoking the function `ExploreStates`, until the iteration times is over the given maximum sequence length $maxL$ or no new heap states could be found (Line 39-43). During state exploration, we maintain a set V_{act} containing all *active* abstract states, i.e., abstract states that are not merged into a union state and could be used to further explore new states.

Algorithm 1: The pseudo-code of `BuildGraph`.

```
Input: A set of methods $\mathcal{M}$, a mapping $hs$ specifying the
       heap scope, and a maximum sequence length $maxL$
Output: A state transformation graph $\mathcal{G} = (V_{act}, V, E)$
1  $\mathcal{H}_{emp} \leftarrow (\{\texttt{null}\}, \{\texttt{null}\}, \emptyset, \emptyset, \texttt{true})$;
2  $V \leftarrow \{\mathcal{H}_{emp}\}$; $V_{act} \leftarrow \{\mathcal{H}_{emp}\}$; $E \leftarrow \emptyset$;
3  Function AddState($\mathcal{H}_{new}, \mathcal{H}_{old}, \mathscr{c}, p$):
4      if $\mathcal{H}_{new}$ is out of the heap scope $hs$ then return $(\bot, \bot)$;
5      $V \leftarrow V \cup \{\mathcal{H}_{new}\}$; $E \leftarrow E \cup \{(\mathcal{H}_{old}, \mathscr{c}, p, \mathcal{H}_{new})\}$;
6      for each abstract state $\mathcal{H} \in V_{act}$ do
7          $\sigma \leftarrow$ DecideIsomorphism($\mathcal{H}, \mathcal{H}_{new}$);
8          if $\sigma \neq \bot$ then
9              if $\varphi_{\mathcal{H}_{new}} \rightarrow \varphi_{\mathcal{H}}[v := \sigma(v)]$ always holds then
10                 return $(\bot, \bot)$
11             $\mathcal{H}_u \leftarrow \textbf{merge}_{\sigma}(\mathcal{H}, \mathcal{H}_{new})$;
12             $V \leftarrow V \cup \{\mathcal{H}_u\}$;
13             $E \leftarrow E \cup \{(\mathcal{H}, \epsilon, \sigma, \mathcal{H}_u)\} \cup \{(\mathcal{H}_{new}, \epsilon, \sigma_{\textbf{id}}, \mathcal{H}_u)\}$;
14             $V_{act} \leftarrow V_{act} \backslash \{\mathcal{H}\} \cup \{\mathcal{H}_u\}$;
15             return $(\mathcal{H}_u, \mathcal{H})$;
16     $V_{act} \leftarrow V_{act} \cup \{\mathcal{H}_{new}\}$;
17     return $(\mathcal{H}_{new}, \bot)$;
18 Function ExploreStates($oldStates$):
19     $newStates \leftarrow \emptyset$;
20     while $oldStates \neq \emptyset$ do
21         $\mathcal{H}_{old} \leftarrow$ pop an abstract state from $oldStates$;
22         for each public method $m \in \mathcal{M}$ do
23             for each $\mathscr{c} \in$ GetAbstractCalls($\mathcal{H}_{old}, m$) do
24                 perform symbolic execution for $\mathscr{c}$ and obtain
                    a set of path descriptors $\{(\alpha_p, O_p, r_p, \tau_p)\}$;
25                 for each path $p$ do
26                     if $\varphi_{\mathcal{H}_{old}} \land \alpha_p$ is unsatisfiable then
27                         continue
28                     $\mathcal{H}_{new} \leftarrow \textbf{PostState}(\mathcal{H}_{old}, \mathscr{c}, p)$;
29                     $\mathcal{H}_u, \mathcal{H} \leftarrow$ AddState($\mathcal{H}_{new}, \mathcal{H}_{old}, \mathscr{c}, p$);
30                     if $\mathcal{H}_u = \bot$ then continue;
31                     if $\mathcal{H}_u = \mathcal{H}_{new}$ then
32                         $newStates \leftarrow newStates \cup \{\mathcal{H}_{new}\}$;
33                         continue;
34                     if $\mathcal{H} \in oldStates$ then
35                         $oldStates \leftarrow oldStates \backslash \{\mathcal{H}\} \cup \{\mathcal{H}_u\}$;
36                     else
37                         $newStates \leftarrow newStates \backslash \{\mathcal{H}\} \cup \{\mathcal{H}_u\}$;
38     return $newStates$;
39 $oldStates \leftarrow \{\mathcal{H}_{emp}\}$;
40 for $L \leftarrow 1 \ldots maxL$ do
41     $oldStates \leftarrow$ ExploreStates($oldStates$);
42     if $oldStates = \emptyset$ then break;
43 return $\mathcal{G} = (V_{act}, V, E)$;
```

The function `ExploreStates` accepts a set of old abstract states *oldStates*, and explores new abstract states by calling public methods $m \in \mathcal{M}$ to transform the old states. Concretely, for each abstract state $\mathcal{H}_{old} \in oldHeaps$, we first enumerate all candidate abstract method calls on $\mathcal{H}_{old}$, and perform symbolic execution to obtain a set of control flow paths with their path descriptors (Line 22-24). The function `GetAbstractCalls`$(\mathcal{H}, m)$ returns a stream of abstract method calls by enumerating all stack-accessible objects (that are also compatible with the type signature of m) in $AO_{\mathcal{H}}$ as object arguments, and allocating fresh symbolic variables as primitive arguments. For each path p, if the path condition α_p is compatible with the state constraint $\varphi_{\mathcal{H}_{old}}$, we compute the new abstract heap state $\mathcal{H}_{new}$ into which $\mathcal{H}_{old}$ would be transformed after executing the path p, and add $\mathcal{H}_{new}$ into the state transformation graph by invoking the function `AddState` (Line 25-29).

In the function `AddState`, we first check whether $\mathcal{H}_{new}$ is within the heap scope *hs*, where *hs* is a mapping that specifies the maximum number of objects of each class in the heap state, and add the new state $\mathcal{H}_{new}$ with a new edge between $\mathcal{H}_{old}$ and $\mathcal{H}_{new}$ into the graph. Then we find whether there exists an active abstract heap state $\mathcal{H} \in V_{act}$ that is structurally isomorphic to $\mathcal{H}_{new}$.

In the case that such an isomorphic abstract state $\mathcal{H}$ exists (Line 9-15), we first check that whether $\varphi_{\mathcal{H}_{new}} \to \varphi_{\mathcal{H}}[v := \sigma(v)]$ always holds. If it always holds, all concrete heap states represented by $\mathcal{H}_{new}$ are already included in those represented by $\mathcal{H}$; therefore, we simply discard $\mathcal{H}_{new}$ and would not use it for further exploration. Otherwise, we merge the two abstract states $\mathcal{H}$ and $\mathcal{H}_{new}$ into a new union state $\mathcal{H}_u$ (Line 11), and add the union state $\mathcal{H}_u$ with two ϵ edges into the graph (Line 12-13). Finally, we mark $\mathcal{H}_u$ to be active but the original state $\mathcal{H}$ that has been merged to be inactive (Line 14), meaning that the union state $\mathcal{H}_u$ in place of $\mathcal{H}$ and $\mathcal{H}_{new}$ would be used for further exploration (Line 34-37). The identity mapping is denoted as $\sigma_{\mathbf{id}}$, and the bijections $\sigma, \sigma_{\mathbf{id}}$ are attached to the ϵ edges for recording the isomorphic relation between the union state and the original states, i.e., $O_{\mathcal{H}_u} = \sigma(O_{\mathcal{H}}) = \sigma_{\mathbf{id}}(O_{\mathcal{H}_{new}})$. In the case that there is no active isomorphic abstract state (Line 16), we mark $\mathcal{H}_{new}$ to be active meaning it could be used to explore new states (Line 32).

Finally, we discuss the function `DecideIsomorphism`, which is supposed to decide whether two abstract heap states $\mathcal{H}_1, \mathcal{H}_2$ are structurally isomorphic, and return a bijection σ if they are indeed isomorphic. We implement this function by using a backtracking algorithm. The algorithm incrementally builds candidate bijections mapping objects/variables in $\mathcal{H}_1$ to objects/variables in $\mathcal{H}_2$, and backtracks as soon as it determines that the (partial) candidate bijection cannot meet the conditions shown in Definition 4.4. To optimize the algorithm, we apply several pruning strategies as follows. First, we search a bijection only between the two sets of stack-accessible objects, because other objects/variables must be reachable from the stack-accessible objects, and the relation between other objects/variables can be derived by $\sigma(o_1) = o_2 \implies \sigma(\delta_{\mathcal{H}_1}(o_1, f)) = \delta_{\mathcal{H}_2}(o_2, f)$. Second, we extract several features to quickly exclude some cases that two abstract heap states cannot be isomorphic (under the current partial bijection). For example, if $\mathcal{H}_1$ and $\mathcal{H}_2$ contain different numbers of (stack-accessible) heap objects, they must be non-isomorphic; if the reference counts (i.e., in-degrees if we regard heap states as object graphs) of two objects o_1, o_2 are different, the bijection σ cannot map o_1 to o_2.

The soundness and completeness of algorithm `BuildGraph`$(\mathcal{M}$, *hs*, *maxL*), which explores reachable heap states and builds a state transformation graph $\mathcal{G} = (V_{act}, V, G)$, are depicted as follows.

Theorem 4.7 (soundness). *For all abstract heap states $\mathcal{H} \in V$ and instances $H \in \mathbf{Inst}(\mathcal{H})$, we hold that H is reachable from H_{emp}.*

Theorem 4.8 (weak completeness). *For all heap states H_n reachable from H_{emp} along a sequence of intermediate heap states $H_1, \ldots, H_{n-1}$, if $n \leq maxL$ and the heap states H_i are within the heap scope hs for all $i = 1, \ldots, n$, we hold that there exists an active abstract state $\mathcal{H} \in V_{act}$ and an instance $H \in \mathbf{Inst}(\mathcal{H})$ such that $H \equiv H_n$.*

4.3 Synthesizing Method Call Sequences

S3：合成方法调用序列

After exploring the reachable states and building the state transformation graph $\mathcal{G} = (V_{act}, V, E)$, we can address the problem of targeted heap state construction by simply enumerating all explored (active) abstract states $\mathcal{H} \in V_{act}$, and checking whether there is an instance of $\mathcal{H}$ satisfying the given specification. If we find such a target state satisfying the specification, the remaining task is to synthesize a sequence of method calls that leads to the target state. This task can be finished by traversing the state transformation graph backwardly, and collecting the method calls along the path from $\mathcal{H}_{emp}$ to $\mathcal{H}$. Since there may be multiple paths that can reach the abstract state $\mathcal{H}$, we need to figure out which path is expected.

The pseudo-code of our algorithm for synthesizing method call sequences is shown in Algorithm 2. We first enumerate the active abstract states $\mathcal{H} \in V_{act}$, and symbolically execute the Boolean function Φ with all possible arguments to obtain the (path) conditions for satisfying the specification Φ (Line 16-20). Then we check whether the conjunction of the state constraint $\varphi_{\mathcal{H}}$ and the condition α_p is satisfiable under an assignment π by invoking the function `CheckSat`. The function `CheckSat` accepts a first-order formula, and returns an assignment if the formula is satisfiable, which maps all variables in the formula, including both bound variables and free variables, to their expected values[1]. After obtaining the assignment π, we successfully find a concrete target state $\mathcal{H}[\pi]$ satisfying the specification Φ with arguments $\overline{o}, \overline{\pi(x)}$. Then we need to synthesize a sequence of method calls that leads to the target state. We notice that the state constraint $\varphi_{\mathcal{H}}$ contains all symbolic arguments of the previous abstract calls leading to $\mathcal{H}$ (under existential quantification), and thus the assignment π contains all assignments to these variables. As a result, we traverse the state transformation graph backwardly to collect method calls, using the assignment π to figure out which path to take and instantiate the abstract method calls along this path (Line 1-15).

At each step, we check the type of the current abstract state $\mathcal{H}$. If $\mathcal{H}$ is a union abstract state (Line 5-10), there are two incoming ϵ edges and two predecessors $\mathcal{H}_1, \mathcal{H}_2$. We obtain the correct predecessor by checking whose state constraint (after transformed by a bijection) is satisfied under the assignment π. Then we need to update the method call sequence S, the object arguments $\overline{o}$, and the assignment π according to the bijection $\sigma_{1/2}$ (Line 10), because we

[1] There might be a problem of name collision, and we might need to rename the bound variables in the formula. The renaming mechanism is omitted for brevity.

Algorithm 2: The pseudo-code of `SynthCallSeq`.

```
   Input: A state transformation graph $\mathcal{G} = (V_{act}, V, E)$, and a
          specification $\Phi$ in the form of a Boolean function
   Output: a sequence of method calls with their return values
           $S$, and two lists of arguments $\overline{o}$ and $\overline{a}$ such that
           $\Phi(\overline{o}, \overline{a})$ returns true
1  Function Traverse($\mathcal{H}, \overline{o}, \overline{x}, \pi$):
2      $S \leftarrow [\,]$; $\overline{a} = \overline{\pi(x)}$;
3      while $\mathcal{H} \neq \mathcal{H}_{emp}$ do
4          if $\mathcal{H}$ is a union abstract state then
5              $(\mathcal{H}_1, \epsilon, \sigma_1, \mathcal{H}), (\mathcal{H}_2, \epsilon, \sigma_2, \mathcal{H}) \leftarrow$ the two
                incoming $\epsilon$ edges of $\mathcal{H}$;
6              if $\varphi_{\mathcal{H}_1}[v := \sigma_1(v)]$ is satisfied under $\pi$ then
7                  $\mathcal{H} \leftarrow \mathcal{H}_1$; $\sigma \leftarrow \sigma_1$;
8              else
                   // $\varphi_{\mathcal{H}_2}[v := \sigma_2(v)]$ is satisfied under $\pi$
9                  $\mathcal{H} \leftarrow \mathcal{H}_2$; $\sigma \leftarrow \sigma_2$;
10             $S \leftarrow \sigma^{-1}(S)$; $\overline{o} \leftarrow \sigma^{-1}(\overline{o})$; $\pi \leftarrow \sigma^{-1}(\pi)$;
11         else
12             $(\mathcal{H}', \mathscr{c}, p, \mathcal{H}) \leftarrow$ the incoming edge of $\mathcal{H}$;
13             insert $(\mathscr{c}[\pi], r_p)$ at the front of $S$;
14             $\mathcal{H} \leftarrow \mathcal{H}'$;
15     return $S, \overline{o}, \overline{a}$;
16 for each abstract heap state $\mathcal{H} \in V_{act}$ do
17     for each $\mathscr{c} = (\Phi, \overline{o}, \overline{x}) \in$ GetAbstractCalls($\mathcal{H}, \Phi$) do
18         perform symbolic execution for $\mathscr{c}$ and obtain a set of
            path descriptors $\{(\alpha_p, O_p, r_p, \tau_p)\}$;
19         for each path $p$ do
20             if $r_p \neq$ true then continue;
21             $\pi \leftarrow$ CheckSat($\varphi_{\mathcal{H}} \wedge \alpha_p$);
22             if $\pi \neq \bot$ then
23                 return Traverse($\mathcal{H}, \overline{o}, \overline{x}, \pi$);
24 return UNSAT;
```

are going to the predecessor $\mathcal{H}_{1/2}$ in the next step, and we need to "translate" the objects and variables in $\mathcal{H}$ into those in $\mathcal{H}_{1/2}$. The notation $\sigma(\overline{o})$ indicates substitution of $\sigma(o)$ for all object $o \in \mathbf{dom}(\sigma)$, and the notations $\sigma(S), \sigma(\pi)$ are similar. If $\mathcal{H}$ is a post-state via an abstract method call (Line 12-14), we simply instantiate the abstract call $\mathscr{c}$ by π, add it with its return value into the sequence S, and move to the pre-state.

5 EVALUATION

To evaluate our proposed approach, we develop a prototype and conduct experiments to answer the following research questions:

- **RQ1:** Does our approach generate method call sequences more effectively than other test generators?
- **RQ2:** Does our approach verify heap-based programs and properties more effectively than other program verifiers?
- **RQ3:** To what extent does the strategy of state merging improve the efficiency of exploring reachable heap states?

5.1 Experimental Setup

5.1.1 Subject Programs. We collect 14 data structure classes implemented in Java as the subject programs, including 4 classes from the SUSHI benchmark [3], 6 classes from the Sireum/Kiasan benchmark, 2 classes from SIR [2], and 2 classes from the JavaScan website [1][2]. All the 14 subject programs are used in the test generation experiment. The subject programs in the Sireum/Kiasan benchmark have been equipped with some class invariants; therefore, we select these programs to be used in the bounded verification experiment.

5.1.2 Prototype Implementation. We implemented our proposed approach in a prototype named MSeqSynth, which uses JBSE [9] for performing symbolic execution, and the Java binding of Z3 SMT solver [12] for checking the satisfiability of first-order constraints.

5.1.3 Configuration. We conduct three experiments to answer the three research questions. All the experiments are executed on a Windows 10 machine equipped with Intel Core i7-10700 CPU at 2.90GHz and 16 GB RAM. The JVM is configured with `-Xmx4096m`.

In the first experiment, we evaluate the effectiveness of our approach on generating test cases (method call sequences) to cover the program branches of the 14 subject programs, comparing with the state-of-the-art test generator SUSHI. The frontend of SUSHI is a symbolic-execution-based path selector, for identifying a set of paths that should be exercised to cover program branches. Each path corresponds to a *test generation task*, and is then emitted to the backend of SUSHI – a genetic algorithm for searching a method call sequence that satisfies the path condition. Since our approach is designed for generating a sequence satisfying a given specification, we integrate our approach into SUSHI as the backend, and use the received path condition as the specification. We compare our approach with SUSHI on the achieved branch coverage and the average generation time.

We configure the global time budget to be 1 hour, and the time budget for each test generation task to be 3 minutes. SUSHI's frontend requires the user to provide invariants (written in the HEX language [9]) of the data structure classes for reducing exploration of unreachable paths, and bounds such as maximum number of lazily-initialized objects N_{init} and maximum depth of the explored state N_{depth} for avoiding infinite exploration. For the subject programs from the SUSHI benchmark, we use the same accurate invariants provided in the benchmark, which discard all unreachable paths. (The subject programs in SUSHI benchmark are slightly different from the programs with the same name but in other benchmark, because they have been instrumented with several shadow fields for writing accurate invariants.) For the remaining subject programs, we manually write partial invariants to discard only a part of unreachable paths. There are two major reasons for writing partial rather than accurate invariants for these programs: (1) it might be difficult to write accurate invariants in the HEX language, and (2) we would like to have a fair comparison against SUSHI by providing different configurations of invariants. To improve the statistical significance of experimental results, we configure as large as possible bounds for SUSHI, $N_{init} = 5$ and $N_{depth} = 50$, such that the

[2] Not all classes in the preceding benchmarks are collected, because at present our prototype does not support some language features such as arrays, strings, floating numbers, generics, and subtype polymorphism.

frontend of SUSHI could explore many paths and produce many test generation tasks, but does not straightforwardly run out of the time budget. To configure our approach, we need to specify a maximum sequence length *maxL* and a finite heap scope *hs*. In this experiment on test generation, we set $maxL = +\infty$ meaning no limit on the sequence length, but set an extra time budget for state exploration to be 30 minutes. The heap scope *hs* is specified as follows. For the internal classes (e.g. `AvlNode`), which are manipulated inside the data structure classes and do not appear in the parameters of the public methods, we set the maximum number of heap objects of these classes to be $N_{obj} = 6$. For other classes, we set the maximum number of heap objects to be the maximum number of parameters of these classes in all methods' parameter lists.

In the second experiment, we evaluate the effectiveness of our approach on verifying heap-based programs and properties. Because the existing program verifiers do not exactly address our problem – whether all heap states reachable from calling public methods satisfy a given property, we construct a baseline $\mathrm{SE_{seq}}$ that extends symbolic execution to (partially) address our problem by writing *driver programs* for each target class, which is inspired by Visser et al. [33]. The driver program constructs a receiver object of the target class, and non-deterministically executes all sequences of method calls on the receiver object. The baseline $\mathrm{SE_{seq}}$ then performs symbolic execution on the driver program to check whether all possible paths (within the user-specified scope) leading to heap states satisfying the property. We select JPF [32] as the underlying symbolic executor for $\mathrm{SE_{seq}}$, because JPF supports APIs for non-deterministic execution.

Because the driver program constructs only the receiver object, it does not support any method with object arguments, we select the 3 classes whose methods do not contain object arguments from the Sireum/Kiasan benchmark, and use the 7 invariants written in these classes as the properties to be verified. We configure the time budget for verifying each property to be 30 minutes. The heap scope *hs* is also set to 1 to be consistent with the baseline, and the maximum sequence length *maxL* is configured with two different values (7 and 8). The driver programs are written to also explore all method sequences within *maxL*.

In the third experiment, we evaluate the efficiency improvement of our state matching strategy for exploring reachable heap states. We compare the two versions of our approach, one with state merging and one without state merging (remove Line 6-15 of the Algorithm 1), on the elapsed time of state exploration for the 14 subject classes. We use the same heap scope specified in the first experiment, three different maximum sequence lengths (5, 6, and 7), and configure the time budget to be 30 minutes.

5.2 RQ1: Effectiveness on Test Generation

Table 1 shows the results of the first experiment for answering RQ1. As shown in the table, the overall elapsed time of our approach is less (or equal) than SUSHI on all the subject programs, even if we count in the time for offline exploration of reachable states. After offline exploration, the average generation time of our approach for each test generation task is more than 100X faster than SUSHI, especially on the tasks that cannot be solved, because SUSHI cannot determine the *inexistence* of a solution but mostly only reports a timeout. Due to the same reason, on several complex data structures such as `Avl`, `RBT`, `AATree` in the Kiasan benchmark, and `Binom` in the JavaScan website, the number of tasks ($N_{solve} + N_{fail}$) processed by our approach is much more than SUSHI; therefore, our approach achieves higher branch coverages on these subject classes. There is one subject program `CList` that SUSHI outperforms our approach, because the uncovered branches require constructing a list with length more than 20, which is out of the heap scope that we configure. The total branches covered by our approach on all subjects (1094) is 20% more than those covered by SUSHI (913). Finally, we answer RQ1 by stating that **our approach is overall more efficient than SUSHI and could generate more test cases to cover more program branches**.

5.3 RQ2: Effectiveness on Bounded Verification

Table 2 shows the results of the second experiment for answering RQ2. As shown in the table, the verification time of our approach is shorter than the baseline $\mathrm{SE_{seq}}$. When the maximum sequence length is configured to be 8, $\mathrm{SE_{seq}}$ fails to verify any property in a 30-minute time budget; however, our approach could still verify 4 properties of the subject class `Avl` and `BST`. Finally, we answer RQ2 by stating that in our problem setting, **our approach can verify heap-based programs and properties more efficiently than the baseline implemented with a symbolic execution engine.**

5.4 RQ3: Improvement by State Merging

Table 3 shows the results of the third experiment for answering RQ3. As shown in the table, the elapsed time of state exploration with state merging is much shorter than it without state merging, especially when the maximum sequence length *maxL* is large. We answer RQ3 by stating that **the strategy of state matching significantly improves the efficiency of exploring all reachable heap states within a user-specified scope**.

6 LIMITATIONS AND DISCUSSION

There are two major limitations of our approach as we will discuss in this section.

Limitations about applicability. As stated in Section 4, our approach needs to perform symbolic execution on data structure methods, for computing the abstract post-states into which an abstract pre-state would be transformed after executing an abstract call. Therefore, our approach inherently have the limitations of symbolic execution, e.g., path explosion, unsolvable path conditions, and interactions with the outside environment; these limitations make our approach hard to apply to complicated and large scale projects.

Limitations about configuration. Our approach requires an explicit configuration of a heap scope and a maximum sequence length, for bounding the state space to explore. In the one hand, this configuration enables users to explicitly control the scope and length of the generated sequences, so that they could obtain relatively small test cases to debug their programs. But in the other hand, it might be tricky to provide an appropriate configuration: too small configuration might lead to failure in generating an expected sequence, while too large configuration might lead to high time and

Table 1: Comparative evaluation of MSeqSynth and SUSHI.

				MSeqSynth							**SUSHI**					
	Subject	$\lvert\mathcal{M}\rvert$	B_{all}	T_{all}	$T_{explore}$	N_{solve}	N_{fail}	T_{solve}	T_{fail}	B_{cov}	T_{all}	N_{solve}	N_{fail}	T_{solve}	T_{fail}	B_{cov}
SUSHI	Avl	7	59	51.5	30	15	0	0.02	-	59	120.8	15	0	6	-	59
	RBT	10	191	399.4	300	34	0	0.22	-	**191**	3600*	30	14	35.1	175.5	162
	DList	38	136	486	328	49	0	0.05	-	**136**	3600*	41	16	11.9	>180	111
	CList	7	80	147	62	11	43	0.02	1.42	78	500.7	11	2	19.3	129.2	**80**
Kiasan	Avl	7	55	64.1	36	15	203	0.01	<0.01	**55**	3600*	10	20	5.6	>180	29
	RBT	10	180	438.7	295	35	983	0.08	0.04	**175**	3600*	20	21	16.2	>180	101
	BST	8	51	68.2	49	14	0	0.01	-	51	100.1	14	0	5.6	-	51
	AATree	8	58	3600*	1800*	16	563	0.02	2.95	**56**	3600*	12	18	5.6	>180	40
	Leftist	7	31	339.1	317	10	5	0.01	0.73	31	1000	10	5	5.5	>180	31
	Stack	8	17	28.3	12	10	0	0.01	-	17	67	10	0	5.5	-	17
SIR	DList	22	81	206	151	33	3	0.03	0.01	81	1018	33	3	8.4	>180	81
	SList	13	41	199.2	167	13	1	0.01	<0.01	41	302.7	13	1	5.5	>180	41
JavaScan	Skew	6	25	43.2	29	8	0	0.01	-	25	54.6	8	0	5.5	-	25
	Binom	9	114	3600*	1298	16	1419	0.05	0.02	**98**	3600*	14	16	5.6	>180	85

For each subject program, we report the number of public methods ($\lvert\mathcal{M}\rvert$) and the number of all program branches to cover (B_{all}). The number of test generation tasks that are successfully solved or failed to solve is respectively reported as N_{solve} or N_{fail}. Note N_{fail} contains the tasks that are unsolvable. The average generation time for the solved tasks or failed tasks is reported as T_{solve} or T_{fail}. The elapsed time of MSeqSynth for (possibly offline) state exploration is $T_{explore}$. The overall elapsed time for each subject program is T_{all}, and the number of covered program branches is B_{cov}. All the time statistics are reported in seconds. An asterisk (*) indicates the execution timed out.

Table 2: Comparative evaluation of MSeqSynth and SE$_{\text{seq}}$.

		maxL = 7		**maxL** = 8	
Subject	**Property**	T_{synth}	T_{SE}	T_{synth}	T_{SE}
Avl	balanced	60.5	258	66.8	N/A
	ordered	31.1	260	39.5	N/A
	wellFormed	44.4	255	52.2	N/A
BST	ordered	39.1	1743	61.1	N/A
AATree	ordered	790.8	1630	N/A	N/A
	wellLevel	775.7	890	N/A	N/A
	wellFormed	1162	1637	N/A	N/A

The verification time of MSeqSynth and SE$_{\text{seq}}$ for each property is reported as T_{synth} and T_{SE} in seconds (N/A means out of time budget or memory exhausted). T_{synth} includes the state exploration time.

Table 3: Experimental results for evaluating the efficiency improvement of the state merging strategy.

	maxL = 5		**maxL** = 6		**maxL** = 7	
Subject	T_{merge}	T_{not}	T_{merge}	T_{not}	T_{merge}	T_{not}
Avl	13	27	13	748	18	N/A
RBT	88	N/A	90	N/A	98	N/A
DList	141	N/A	178	N/A	251	N/A
CList	16	633	22	N/A	29	N/A
Avl	18	26	18	638	20	N/A
RBT	47	N/A	48	N/A	54	N/A
BST	11	31	12	1253	18	N/A
AATree	69	107	126	N/A	733	N/A
Leftist	9	15	9	611	13	N/A
Stack	9	17	9	246	9	N/A
DList	60	N/A	77	N/A	116	N/A
SList	25	400	35	N/A	56	N/A
Skew	7	8	8	20	9	190
Binom	143	189	148	N/A	176	N/A

T_{merge} is the elapsed time of state exploration with state merging, and T_{not} is the elapsed time without state merging, all in seconds (N/A means out of time budget or memory exhausted).

memory consumption because the number of reachable (abstract) heap states becomes large. In future work, we plan to alleviate this limitation by making our procedure of state exploration incremental and on-demand, and the configuration self-adjusted instead of set by users.

7 RELATED WORK

7.1 Test Generation

Various automatic test generation approaches have been proposed for heap-based programs. These approaches can be broadly classified into two categories: direct construction [4, 7, 9, 21, 26, 32] and sequence generation [8, 11, 16, 17, 23, 25, 28].

Direct construction approaches, such as specific symbolic executors [4, 9, 32] and specification-based test generators [7, 21, 26], construct heap states by directly assigning values to the fields of heap objects. These approaches either model the whole heap and convert the problem as a constraint solving problem, or enumerate the heap structure in some way and use constraint solving to fill the primitive values. As discussed in the introduction, these approaches have two major limitations: breaking encapsulation and possibly producing invalid inputs; these limitations can be addressed by sequence generation approaches.

Sequence generation approaches indirectly produce desired input heap states by generating and executing a sequence of calls to the public methods in the classes under test. Some approaches use random sampling [11, 25] or heuristic search [16, 23]. For example, JCrasher [11] randomly traverses a pre-built type graph to generate type-correct test inputs. Randoop [25] builds the method call sequence incrementally by maintaining an object pool and randomly selecting objects from the pool to form the next call. EvoSuite [16] exploits a Genetic Algorithm (GA) to search for a set of method call sequences that maximize a chosen coverage criterion (e.g., branch coverage). EvoObj [23] facilitates EvoSuite by creating test templates that leave only some slots for GA to mutate their values. However, these preceding approaches are not applicable in our work's target setting (i.e., generating specific test inputs for satisfying the given specification), and typically fall short in covering corner cases of coverage targets.

Given that random sampling and heuristic search tend to generate ineffective method call sequences, Xie et al. [34] and Visser et al. [33] propose to generate only method sequences that lead to either a different heap structure or a different path condition not subsumed by already explored path conditions. To achieve this goal, their approaches also enumerate heap states in a way similar to our approach. However, there are two main important differences: (1) these approaches deal with a single receiver object, and cannot generate sequences that involve public methods from multiple classes or public methods with non-primitive parameters, and (2) since their approaches' goal is to explore different path conditions, these approaches do not merge states, whereas state merging is critical to the effectiveness of our approach as shown in our evaluation.

Sharing the same goal as our approach, Seeker [28] and SUSHI [8] aim to generate method call sequences that produce heap states that satisfy a specific condition. Seeker [28] first uses a static analysis to build a skeleton of the call sequence with missing primitive values, and then fill the primitive values using dynamic symbolic execution. Due to the imprecision of static analysis, Seeker often fails to build a desirable skeleton. Therefore, Seeker cannot be used for the bounded verification task, and for the test generation task, Seeker is outperformed by SUSHI [8], a previous state-of-the-art approach. SUSHI uses a genetic search algorithm to find a method call sequence. As stated earlier, the genetic search often fails when there is a complex constraint between primitive values. Thus, SUSHI also cannot be used for the bounded verification task, and for the test generation task, SUSHI is outperformed by our approach as shown in our evaluation.

7.2 Bounded Verification

There are various existing approaches [10, 13, 14, 19, 20, 32] for verifying a heap-based program against the given properties within a user-specified finite bound. The approaches proposed by Dennis et al. [13] and Dolby et al. [14] encode the program in a relational logic and use a relational constraint solver to find specification violations. JayHorn [19, 20] is a verification tool for Java programs. It uses the Soot [31] Java optimization framework as a front-end, and generates a set of constrained Horn clauses to encode the verification condition, and sends the Horn clauses to a Horn engine for solving. JayHorn is particularly useful for verifying programs with unbounded data structures and unbounded iteration due to its way of encoding the heap, but it can also be configured to deal with bounded verification tasks. JPF [32] and JBMC [10] are symbolic-execution-based verifiers, which symbolically explore the program within the user-specified bound, and check whether there exists a program path leading to a state violating the given properties. However, in contrast to our approach, these approaches do not focus on verifying the program behaviors under only valid input states reachable from calling public API methods, and they cannot generate an API call sequence that instantiates an input state and results in property violations.

8 CONCLUSION

In this paper, we have proposed an efficient algorithm for synthesizing method call sequences, by combining enumerative techniques and symbolic techniques. The synthesis algorithm includes an offline procedure for exploring all reachable heap states within a user-specified scope and building the state transformation graph, and an online procedure for extracting a method call sequence from the state transformation graph, such that the method call sequence satisfies the given specification. To improve the efficiency of offline state exploration, we introduce the notion of abstract heap state and the strategy of state merging. The evaluation results show that our synthesis algorithm performs efficiently in both test generation tasks and bounded verification tasks.

ACKNOWLEDGMENTS

We acknowledge Peiyi Sun, Dan Hao, and Lu Zhang for the discussion during the early phase of the work. This work is sponsored by the National Key Research and Development Program of China under Grant No. 2019YFE0198100, the Innovation and Technology Commission of HKSAR under Grant No. MHP/055/19, National Natural Science Foundation of China under Grant No. 61922003 and 62161146003.

REFERENCES

[1] [n.d.]. *JavaScan.* https://www.javascan.com/chapter/data-structures

[2] [n.d.]. *Software-artifact Infrastructure Repository.* https://sir.csc.ncsu.edu/portal/index.php

[3] [n.d.]. *SUSHI Experiments.* https://github.com/pietrobraione/sushi-experiments

[4] Elvira Albert, Israel Cabanas, Antonio Flores-Montoya, Miguel Gómez-Zamalloa, and Sergio Gutierrez. 2011. jPET: An Automatic Test-Case Generator for Java. In *18th Working Conference on Reverse Engineering, WCRE 2011, Limerick, Ireland, October 17-20, 2011*, Martin Pinzger, Denys Poshyvanyk, and Jim Buckley (Eds.). IEEE Computer Society, 441–442. https://doi.org/10.1109/WCRE.2011.67

[5] Roberto Baldoni, Emilio Coppa, Daniele Cono D'Elia, Camil Demetrescu, and Irene Finocchi. 2018. A Survey of Symbolic Execution Techniques. *ACM Comput. Surv.* 51, 3 (2018), 50:1–50:39. https://doi.org/10.1145/3182657

[6] Clark W. Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanovic, Tim King, Andrew Reynolds, and Cesare Tinelli. 2011. CVC4. In *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings (Lecture Notes in Computer Science, Vol. 6806)*, Ganesh Gopalakrishnan and Shaz Qadeer (Eds.). Springer, 171–177. https://doi.org/10.1007/978-3-642-22110-1_14

[7] Chandrasekhar Boyapati, Sarfraz Khurshid, and Darko Marinov. 2002. Korat: automated testing based on Java predicates. In *Proceedings of the International Symposium on Software Testing and Analysis, ISSTA 2002, Roma, Italy, July 22-24, 2002*, Phyllis G. Frankl (Ed.). ACM, 123–133. https://doi.org/10.1145/566172.566191

[8] 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 the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis, Santa Barbara, CA, USA, July 10 - 14, 2017*, Tevfik Bultan and Koushik Sen (Eds.). ACM, 90–101. https://doi.org/10.1145/3092703.3092715

[9] Pietro Braione, Giovanni Denaro, and Mauro Pezzè. 2015. Symbolic execution of programs with heap inputs. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015, Bergamo, Italy, August 30 - September 4, 2015*, Elisabetta Di Nitto, Mark Harman, and Patrick Heymans (Eds.). ACM, 602–613. https://doi.org/10.1145/2786805.2786842

[10] Lucas C. Cordeiro, Pascal Kesseli, Daniel Kroening, Peter Schrammel, and Marek Trtík. 2018. JBMC: A Bounded Model Checking Tool for Verifying Java Bytecode. In *Computer Aided Verification - 30th International Conference, CAV 2018, Held*

as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings, Part I (Lecture Notes in Computer Science, Vol. 10981), Hana Chockler and Georg Weissenbacher (Eds.). Springer, 183–190. https://doi.org/10.1007/978-3-319-96145-3_10

[11] Christoph Csallner and Yannis Smaragdakis. 2004. JCrasher: an automatic robustness tester for Java. *Softw. Pract. Exp.* 34, 11 (2004), 1025–1050. https://doi.org/10.1002/spe.602

[12] Leonardo Mendonça de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings (Lecture Notes in Computer Science, Vol. 4963)*, C. R. Ramakrishnan and Jakob Rehof (Eds.). Springer, 337–340. https://doi.org/10.1007/978-3-540-78800-3_24

[13] Greg Dennis, Felix Sheng-Ho Chang, and Daniel Jackson. 2006. Modular verification of code with SAT. In *Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2006, Portland, Maine, USA, July 17-20, 2006*, Lori L. Pollock and Mauro Pezzè (Eds.). ACM, 109–120. https://doi.org/10.1145/1146238.1146251

[14] Julian Dolby, Mandana Vaziri, and Frank Tip. 2007. Finding bugs efficiently with a SAT solver. In *Proceedings of the 6th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2007, Dubrovnik, Croatia, September 3-7, 2007*, Ivica Crnkovic and Antonia Bertolino (Eds.). ACM, 195–204. https://doi.org/10.1145/1287624.1287653

[15] Niklas Eén and Niklas Sörensson. 2003. An Extensible SAT-solver. In *Theory and Applications of Satisfiability Testing, 6th International Conference, SAT 2003. Santa Margherita Ligure, Italy, May 5-8, 2003 Selected Revised Papers (Lecture Notes in Computer Science, Vol. 2919)*, Enrico Giunchiglia and Armando Tacchella (Eds.). Springer, 502–518. https://doi.org/10.1007/978-3-540-24605-3_37

[16] Gordon Fraser and Andrea Arcuri. 2013. Whole Test Suite Generation. *IEEE Trans. Software Eng.* 39, 2 (2013), 276–291. https://doi.org/10.1109/TSE.2012.14

[17] Kobi Inkumsah and Tao Xie. 2008. Improving Structural Testing of Object-Oriented Programs via Integrating Evolutionary Testing and Symbolic Execution. In *23rd IEEE/ACM International Conference on Automated Software Engineering (ASE 2008), 15-19 September 2008, L'Aquila, Italy.* IEEE Computer Society, 297–306. https://doi.org/10.1109/ASE.2008.40

[18] Daniel Jackson. 2006. *Software Abstractions - Logic, Language, and Analysis.* MIT Press. http://mitpress.mit.edu/catalog/item/default.asp?ttype=2&tid=10928

[19] Temesghen Kahsai, Philipp Rümmer, Huascar Sanchez, and Martin Schäf. 2016. JayHorn: A Framework for Verifying Java programs. In *Computer Aided Verification - 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part I (Lecture Notes in Computer Science, Vol. 9779)*, Swarat Chaudhuri and Azadeh Farzan (Eds.). Springer, 352–358. https://doi.org/10.1007/978-3-319-41528-4_19

[20] Temesghen Kahsai, Philipp Rümmer, and Martin Schäf. 2019. JayHorn: A Java Model Checker - (Competition Contribution). In *Tools and Algorithms for the Construction and Analysis of Systems - 25 Years of TACAS: TOOLympics, Held as Part of ETAPS 2019, Prague, Czech Republic, April 6-11, 2019, Proceedings, Part III (Lecture Notes in Computer Science, Vol. 11429)*, Dirk Beyer, Marieke Huisman, Fabrice Kordon, and Bernhard Steffen (Eds.). Springer, 214–218. https://doi.org/10.1007/978-3-030-17502-3_16

[21] Shadi Abdul Khalek, Guowei Yang, Lingming Zhang, Darko Marinov, and Sarfraz Khurshid. 2011. TestEra: A tool for testing Java programs using alloy specifications. In *26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011), Lawrence, KS, USA, November 6-10, 2011*, Perry Alexander, Corina S. Pasareanu, and John G. Hosking (Eds.). IEEE Computer Society, 608–611. https://doi.org/10.1109/ASE.2011.6100137

[22] Quang Loc Le, Jun Sun, and Wei-Ngan Chin. 2016. Satisfiability Modulo Heap-Based Programs. In *Computer Aided Verification - 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part I (Lecture Notes in Computer Science, Vol. 9779)*, Swarat Chaudhuri and Azadeh Farzan (Eds.). Springer, 382–404. https://doi.org/10.1007/978-3-319-41528-4_21

[23] Yun Lin, You Sheng Ong, Jun Sun, Gordon Fraser, and Jin Song Dong. 2021. Graph-based seed object synthesis for search-based unit testing. In *ESEC/FSE '21: 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Athens, Greece, August 23-28, 2021*, Diomidis Spinellis, Georgios Gousios, Marsha Chechik, and Massimiliano Di Penta (Eds.). ACM, 1068–1080. https://doi.org/10.1145/3468264.3468619

[24] Muhammad Zubair Malik, Aman Pervaiz, and Sarfraz Khurshid. 2007. Generating Representation Invariants of Structurally Complex Data. In *Tools and Algorithms for the Construction and Analysis of Systems*, Orna Grumberg and Michael Huth (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 34–49.

[25] Carlos Pacheco, Shuvendu K. Lahiri, Michael D. Ernst, and Thomas Ball. 2007. Feedback-Directed Random Test Generation. In *29th International Conference on Software Engineering (ICSE 2007), Minneapolis, MN, USA, May 20-26, 2007.* IEEE Computer Society, 75–84. https://doi.org/10.1109/ICSE.2007.37

[26] Long H. Pham, Quang Loc Le, Quoc-Sang Phan, Jun Sun, and Shengchao Qin. 2019. Enhancing Symbolic Execution of Heap-Based Programs with Separation Logic for Test Input Generation. In *Automated Technology for Verification and Analysis - 17th International Symposium, ATVA 2019, Taipei, Taiwan, October 28-31, 2019, Proceedings (Lecture Notes in Computer Science, Vol. 11781)*, Yu-Fang Chen, Chih-Hong Cheng, and Javier Esparza (Eds.). Springer, 209–227. https://doi.org/10.1007/978-3-030-31784-3_12

[27] Jeremias Rößler, Andreas Zeller, Gordon Fraser, Cristian Zamfir, and George Candea. 2013. Reconstructing Core Dumps. In *Sixth IEEE International Conference on Software Testing, Verification and Validation, ICST 2013, Luxembourg, Luxembourg, March 18-22, 2013.* IEEE Computer Society, 114–123. https://doi.org/10.1109/ICST.2013.18

[28] Suresh Thummalapenta, Tao Xie, Nikolai Tillmann, Jonathan de Halleux, and Zhendong Su. 2011. Synthesizing method sequences for high-coverage testing. In *Proceedings of the 26th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2011, part of SPLASH 2011, Portland, OR, USA, October 22 - 27, 2011*, Cristina Videira Lopes and Kathleen Fisher (Eds.). ACM, 189–206. https://doi.org/10.1145/2048066.2048083

[29] Nikolai Tillmann and Jonathan de Halleux. 2008. Pex-White Box Test Generation for .NET. In *Tests and Proofs - 2nd International Conference, TAP 2008, Prato, Italy, April 9-11, 2008. Proceedings (Lecture Notes in Computer Science, Vol. 4966)*, Bernhard Beckert and Reiner Hähnle (Eds.). Springer, 134–153. https://doi.org/10.1007/978-3-540-79124-9_10

[30] Emina Torlak. 2009. *A constraint solver for software engineering: finding models and cores of large relational specifications.* Ph.D. Dissertation. Massachusetts Institute of Technology, Cambridge, MA, USA. http://hdl.handle.net/1721.1/46789

[31] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie J. Hendren, Patrick Lam, and Vijay Sundaresan. 1999. Soot - a Java bytecode optimization framework. In *Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative Research, November 8-11, 1999, Mississauga, Ontario, Canada*, Stephen A. MacKay and J. Howard Johnson (Eds.). IBM, 13. https://dl.acm.org/citation.cfm?id=782008

[32] Willem Visser, Corina S. Pasareanu, and Sarfraz Khurshid. 2004. Test input generation with java PathFinder. In *Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2004, Boston, Massachusetts, USA, July 11-14, 2004*, George S. Avrunin and Gregg Rothermel (Eds.). ACM, 97–107. https://doi.org/10.1145/1007512.1007526

[33] Willem Visser, Corina S. Pasareanu, and Radek Pelánek. 2006. Test input generation for java containers using state matching. In *Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2006, Portland, Maine, USA, July 17-20, 2006*, Lori L. Pollock and Mauro Pezzè (Eds.). ACM, 37–48. https://doi.org/10.1145/1146238.1146243

[34] Tao Xie, Darko Marinov, Wolfram Schulte, and David Notkin. 2005. Symstra: A Framework for Generating Object-Oriented Unit Tests Using Symbolic Execution. In *Tools and Algorithms for the Construction and Analysis of Systems, 11th International Conference, TACAS 2005, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2005, Edinburgh, UK, April 4-8, 2005, Proceedings (Lecture Notes in Computer Science, Vol. 3440)*, Nicolas Halbwachs and Lenore D. Zuck (Eds.). Springer, 365–381. https://doi.org/10.1007/978-3-540-31980-1_24