10-2019

Concolic testing heap-manipulating programs

Long H. PHAM

Quang LOC LE

Quoc-Sang PHAN

Jun SUN
Singapore Management University, junsun@smu.edu.sg

Citation

PHAM, Long H.; LOC LE, Quang; PHAN, Quoc-Sang; and SUN, Jun. Concolic testing heap-manipulating programs. (2019). *International Symposium on Formal Methods (FM 2019), Porto, Portugal, 2019 October 7-11*.
Available at: https://ink.library.smu.edu.sg/sis_research/4638

Concolic Testing Heap-Manipulating Programs (Technical Report)

Long H. Pham[1]*, Quang Loc Le[2], Quoc-Sang Phan[3], and Jun Sun[4]

[1] Singapore University of Technology and Design, Singapore
[2] School of Computing & Digital Technologies, Teesside University, UK
[3] Synopsys, Inc., USA
[4] Singapore Management University, Singapore

Abstract. Concolic testing is a test generation technique which works effectively by integrating random testing generation and symbolic execution. Existing concolic testing engines focus on numeric programs. Heap-manipulating programs make extensive use of complex heap objects like trees and lists. Testing such programs is challenging due to multiple reasons. Firstly, test inputs for such programs are required to satisfy non-trivial constraints which must be specified precisely. Secondly, precisely encoding and solving path conditions in such programs are challenging and often expensive. In this work, we propose the first concolic testing engine called CSF for heap-manipulating programs based on separation logic. CSF effectively combines specification-based testing and concolic execution for test input generation. It is evaluated on a set of challenging heap-manipulating programs. The results show that CSF generates valid test inputs with high coverage efficiently. Furthermore, we show that CSF can be potentially used in combination with precondition inference tools to reduce the user effort.

1 Introduction

Unit testing is essential during the software development process. To automate unit testing effectively, we are required to generate *valid* test inputs which exercise program behaviors *comprehensively* and *efficiently*. Many techniques for automating unit testing have been proposed, including random testing [18] and symbolic execution [45]. A recent development is the concolic testing technique [32,40]. Concolic testing works by integrating random testing and symbolic execution to overcome their respective limitations [46]. It has been shown that concolic testing often works effectively [47].

Existing concolic testing engines focus on numeric programs, i.e., programs which take numeric type variables as inputs. In contrast, *heap-manipulating* programs make extensive use of heap objects and their inputs are often dynamically allocated data structures. Test input generation for heap-manipulating programs is hard for two reasons. Firstly, the test inputs are often heap objects with complex structures and strict requirements over their shapes and sizes. Secondly, the inputs have unbounded domains. Ideally, test generation for heap-manipulating programs must satisfy three requirements.

1. *(Validity)* It must generate valid test inputs.

* Corresponding author. Email: longph1989@gmail.com

2. *(Comprehensiveness)* It must exercise program behaviors comprehensively, e.g., maximizing certain code coverage.
3. *(Efficiency)* It must be efficient.

Existing approaches often overlook one or more of the requirements. The state-of-the-art approaches are based on classical symbolic execution [26] with lazy initialization [45]. To achieve comprehensiveness and efficiency, lazy initialization postpones the initialization of reference type symbolic variables and fields until they are accessed. However, lazy initialization has limited support to capture constraints on the shapes of the input data structures. As a result, invalid test inputs are generated, which are not only wasteful but also lead to the exploration of infeasible program paths. Furthermore, because the values of un-accessed fields are not initialized, the generated test inputs need to be further concretized. Subsequent works on improving lazy initialization [15,16,21,45] share the same aforementioned problems. To address the validity requirement, Braione *et al.* [11] introduced a logic called HEX as a specification language for the input data structures. However, HEX has limited expressiveness and thus cannot describe many data structures (unless using additional user-provided methods called $triggers$).

Inspired by the recent success of concolic execution (e.g., [1,41]), we aim to develop a concolic execution engine for heap-manipulating programs. Developing a concolic execution engine which achieves validity, comprehensiveness and efficiency is however highly non-trivial. For validity, we need a specification language which is expressive enough to capture constraints over the shapes and sizes of heap objects. We thus adopt a recently proposed fragment of separation logic which is shown to be expressive and decidable [30]. For comprehensiveness and efficiency, we propose a novel concolic testing strategy which combines specification-based testing and concolic execution. That is, we first generate test inputs according to the specification in a black-box manner and then apply concolic execution to cover those uncovered program parts.

In summary, we make the following contributions. Firstly, we propose a concolic execution engine for heap-manipulation programs based on separation logic. Secondly, we combine specification-based testing with concolic execution in order to reduce the cost of constraint solving. Thirdly, we implement the proposal in a tool called Concolic StarFinder (CSF) and evaluate it in multiple experiments.

The rest of this paper is organized as follows. Section 2 illustrates our approach through an example. Section 3 describes our specification language and specification-based test input generation. Next, we present our concolic execution engine in Section 4. We show the implementation and experiments in Section 5. Section 6 discusses related works and finally Section 7 concludes.

2 Approach at a Glance

We illustrate our approach using method $remove$ in class $BinarySearchTree$ from the SIR repository [7]. The method is shown in Fig. 1. It checks if a binary search tree object contains a node with a specific value and, if so, removes the node. To test the method, we must generate two inputs, i.e., a *valid* binary search tree object t and an integer x, and then execute *t.remove(x)*. Note that a valid binary search tree object must

```java
public class BinarySearchTree {
  public BinaryNode root;
  public void remove(int x) {
    root = remove(x, root);
  }
  private BinaryNode remove(int x, BinaryNode t) {
    if (t == null) return t;
    if (x < t.element)
      t.left = remove(x, t.left);
    else if (x > t.element)
      t.right = remove(x, t.right);
    else if (t.left != null && t.right != null){
      t.element = findMin(t.right).element;
      t.right = remove(t.element, t.right);
    } else
      t = (t.left != null) ? t.left : t.right;
    return t;
  }
  private BinaryNode findMin(BinaryNode t) {
    if (t == null) return null;
    else if (t.left == null) return t;
    return findMin(t.left);
  }
}

public class BinaryNode {
  int element; BinaryNode left; BinaryNode right;
}
```

Fig. 1. Sample program

satisfy strict requirements. Firstly, all *BinaryNode* objects must be structured in a binary tree shape. Secondly, for any *BinaryNode* object in the tree, its *element* value must be greater than all the *element* values of its *left* sub-tree and less than those of the *right* sub-tree. One way to define valid binary search tree objects is through programming a $repOK$ method [9,45], as shown in App. 1.

If a $repOK$ method is provided, we can use the black-box enumeration (BBE) approach [45] to generate test inputs. BBE performs symbolic execution with lazy initialization on the $repOK$ method. Although BBE can generate valid test inputs, it also generates many invalid ones, e.g., the generated input is a cyclic graph instead of a tree[5]. In our experiment with BBE for this method, a total of 225 test inputs are generated and only 9 of them are valid. Moreover, because BBE generates test inputs based on the $repOK$ method only, it may not generate a high coverage test suite.

One way to obtain a high coverage test suite is to use the white-box enumeration approach [45]. First, white-box enumeration performs symbolic execution on the method under test to create some partially initialized data structures. Then, these data structures are used as initial inputs to perform symbolic execution with the $repOK$ method. However, because the approach still uses lazy initialization, many invalid test inputs may be generated. Moreover, white-box enumeration requires the availability of a conservative $repOK$ method in the first step, which is not easy to derive. Another approach is to use the HEX logic [12] as a language to specify valid data structures. During lazy initializa-

[5] When BBE runs, we count the structures that the $repOK$ method returns `true` as valid ones, and the structures that the $repOK$ method returns `false` as invalid ones.

Formula	$\Phi ::= \Delta \mid \Phi_1 \vee \Phi_2$	
Symbolic heap	$\Delta ::= \exists \bar{v}.\ (\kappa \wedge \pi)$	
Spatial formula	$\kappa ::= \texttt{emp} \mid x {\mapsto} c(\bar{v}) \mid \texttt{P}(\bar{v}) \mid \kappa_1 * \kappa_2$	
Pure formula	$\pi ::= \texttt{true} \mid \alpha \mid \neg\pi \mid \pi_1 \wedge \pi_2$	$\alpha ::= a_1 = a_2 \mid a_1 \leq a_2$
	$a ::= \texttt{null} \mid k \mid v \mid k{\times}a \mid a_1{+}a_2 \mid -a$	
Data structure	$Node ::= \texttt{data}\ c_i\{\tau_1\ f_{i_1}; ...; \tau_j\ f_{i_j}\}$	$\tau ::= \texttt{bool} \mid \texttt{int} \mid c$
Predicate definition	$Pred ::= \texttt{pred}\ \texttt{P}_\texttt{i}(\bar{v}_i) \equiv \Phi_i$	

Fig. 2. Specification language, where k is a 32-bit integer constant, $\bar{v}$ is a sequence of variables

tion, the exploration is pruned when the heap configuration violates the specification. However, HEX has limited expressiveness, e.g., HEX cannot capture the property that the nodes in the binary search tree are sorted due to the lack of arithmetic constraints.

In comparison, our approach works as follows. We use separation logic to define a predicate $\texttt{bst}(\texttt{root}, minE, maxE)$, which specifies valid binary search trees where $\texttt{root}$ is the root of the tree and $minE$ (resp. $maxE$) is the minimum (resp. maximum) bound of the $element$ values of the tree. We refer the readers to Section 3 for details of the definition. The precondition of method $remove$ is then specified as $\texttt{bst}(this\_root, minE, maxE)$. With the specification, we first apply specification-based testing based on the precondition in a black-box manner. That is, we generate the test inputs according to the precondition using a constraint solver without exploring the method body. After this step, we generate 22 test inputs and they cover 14 over 15 feasible branches of the method $remove$ (including auxiliary method $findMin$). The only branch which is not covered is the $else$ branch at line 21. We then perform concolic execution with the generated test inputs to identify a feasible path which leads to the uncovered branch. After solving that path condition, we obtain the test inputs for 100% branch coverage.

3 Specification-based Testing

Our approach takes as input a heap-manipulating program which has a precondition specified using a language recently developed in [14,30]. In the following, we introduce the language and present the first step of our approach, i.e., specification-based testing based on the provided precondition.

Specification Language The language we adopt supports separation logic, inductive predicates and arithmetical constraints, which is expressive to specify many data structures [14,30]. Its syntax is shown in Fig. 2. In general, the precondition is a disjunction of one or more symbolic heaps. A symbolic heap is an existentially quantified conjunction of a heap formula κ and a pure formula π. While a pure formula is a constraint in the form of the first-order logic, the heap formula is a conjunction of heap predicates which are connected by separating operation $*$. A heap predicate may be the empty predicate $\texttt{emp}$, a points-to predicate $x {\mapsto} c(\bar{v})$ or an inductive predicate $\texttt{P}(\bar{v})$. Reference

Algorithm 1: $\texttt{genFromSpec}(\Gamma, n)$

```
1  if $n = 0$ then
2      $tests \leftarrow \emptyset$
3      foreach $\Delta \in \Gamma$ do
4          $\texttt{r}, \texttt{model} \leftarrow \texttt{sat}(\Delta)$
5          if $\texttt{r} = \texttt{SAT}$ then
6              $tests \leftarrow tests \cup \texttt{toUnitTest}(\texttt{model})$
7      return $tests$
8  else
9      $\Gamma' \leftarrow \emptyset$
10     foreach $\Delta \in \Gamma$ do
11         $\Gamma' \leftarrow \Gamma' \cup \texttt{unfold}(\Delta)$
12     return $\texttt{genFromSpec}(\Gamma', n-1)$
```

types are annotated by the keyword `data`. Variables may have type τ as boolean `bool` or 32-bit integer `int` or user-defined reference type `c`.

Inductive predicates are supplied by the users with the keyword `pred`. They are used to specify constraints on recursively defined data structures like linked lists or trees. Inductive predicates are defined in the same language. For instance, the inductive predicate $\texttt{bst}(\texttt{root}, minE, maxE)$ introduced in Section 2 is defined as follows

$$\begin{aligned}&\texttt{pred}\ \texttt{bst}(\texttt{root}, minE, maxE) \equiv (\texttt{emp} \wedge \texttt{root} = \texttt{null})\\ &\quad \vee (\exists elt, l, r.\ \texttt{root} {\mapsto} BinaryNode(elt, l, r) *\\ &\qquad \texttt{bst}(l, minE, elt) * \texttt{bst}(r, elt, maxE) \wedge minE < elt \wedge maxE > elt)\end{aligned}$$

, where `root` is the root of the tree and $minE$ (resp. $maxE$) is the minimum (resp. maximum) bound of the $element$ values of the tree. Using this definition with $this\_root$ as symbolic value for field $root$ in class $BinarySearchTree$, the precondition of method $remove$ in the preceding section is then specified as $\texttt{bst}(this\_root, minE, maxE)$.

Specification-based Testing If we follow existing concolic testing strategies [18], we would first generate random test inputs before applying concolic execution. However, it is unlikely that randomly generated heap objects are valid due to the strict precondition. Thus, we apply specification-based testing to generate test inputs based on the user-provided precondition instead.

The details are shown in Algorithm 1. The inputs are a set of formulae Γ and a bound on n. The initial value of Γ contains only the precondition of the program under test. The output is a set of test inputs which are both *valid* and *fully initialized*. Algorithm 1 has two phases.

In the first phase, from line 8 to 12, procedure `unfold` is applied to each symbolic heap Δ in Γ (at line 11) to return a set of unfolded formulae. Recall that a symbolic heap is a conjunction of a heap constraint κ and a pure constraint π. If the heap constraint κ contains no inductive predicates (i.e., it is a base formula), κ is returned as it is. Otherwise, each inductive predicate $\texttt{P}_i(\bar{t}_i)$ in κ is unfolded using its definition. Note

1. $\mathtt{emp} \wedge \mathit{this\_root} = \mathtt{null}$
2. $\exists \mathit{elt}, l, r.\ \mathit{this\_root} {\mapsto} \mathit{BinaryNode}(\mathit{elt}, l, r) * \mathtt{bst}(l, \mathit{minE}, \mathit{elt}) * \mathtt{bst}(r, \mathit{elt}, \mathit{maxE}) \wedge$
   $\mathit{minE} < \mathit{elt} \wedge \mathit{maxE} > \mathit{elt}$
3. $\exists \mathit{elt}, l, r.\ \mathit{this\_root} {\mapsto} \mathit{BinaryNode}(\mathit{elt}, l, r) * \mathtt{bst}(r, \mathit{elt}, \mathit{maxE}) \wedge l = \mathtt{null} \wedge$
   $\mathit{minE} < \mathit{elt} \wedge \mathit{maxE} > \mathit{elt}$
4. $\exists \mathit{elt}, l, r, \mathit{elt}1, l1, r1.\ \mathit{this\_root} {\mapsto} \mathit{BinaryNode}(\mathit{elt}, l, r) * l {\mapsto} \mathit{BinaryNode}(\mathit{elt}1, l1, r1) *$
   $\mathtt{bst}(r, \mathit{elt}, \mathit{maxE}) * \mathtt{bst}(l1, \mathit{minE}, \mathit{elt}1) * \mathtt{bst}(r1, \mathit{elt}1, \mathit{elt}) \wedge$
   $\mathit{minE} < \mathit{elt} \wedge \mathit{maxE} > \mathit{elt} \wedge \mathit{minE} < \mathit{elt}1 \wedge \mathit{elt} > \mathit{elt}1$
5. $\exists \mathit{elt}, l, r.\ \mathit{this\_root} {\mapsto} \mathit{BinaryNode}(\mathit{elt}, l, r) * \mathtt{bst}(l, \mathit{minE}, \mathit{elt}) \wedge r = \mathtt{null} \wedge$
   $\mathit{minE} < \mathit{elt} \wedge \mathit{maxE} > \mathit{elt}$
6. $\exists \mathit{elt}, l, r, \mathit{elt}2, l2, r2.\ \mathit{this\_root} {\mapsto} \mathit{BinaryNode}(\mathit{elt}, l, r) * r {\mapsto} \mathit{BinaryNode}(\mathit{elt}2, l2, r2) *$
   $\mathtt{bst}(l, \mathit{minE}, \mathit{elt}) * \mathtt{bst}(l2, \mathit{elt}, \mathit{elt}2) * \mathtt{bst}(r2, \mathit{elt}2, \mathit{maxE}) \wedge$
   $\mathit{minE} < \mathit{elt} \wedge \mathit{maxE} > \mathit{elt} \wedge \mathit{elt} < \mathit{elt}2 \wedge \mathit{maxE} > \mathit{elt}2$

Fig. 3. Unfoldings

that the definition of $\mathtt{P}_i(\bar{t}_i)$ is a disjunction of multiple base cases and inductive cases. During unfolding, κ is split into a set of formulae, one for each disjunct in the definition of every inductive predicate $\mathtt{P}_i(\bar{t}_i)$ in κ. The process ends when n reaches 0.

Procedure `unfold` is formalized as follows. Given an inductively predicate definition $\mathtt{pred}\ \mathtt{P}_i(\bar{v}_i) \equiv \Phi_i$ and a formula constituted with this predicate, e.g., $\Delta_i * \mathtt{P}_i(\bar{t}_i)$, `unfold` proceeds in two steps. First, it replaces the occurrences of the inductive predicate with its definition as: $\mathtt{unfold}(\Delta_i * \mathtt{P}_i(\bar{t}_i), \mathtt{P}_i(\bar{t}_i)) \equiv \Delta_i * (\Phi_i[\bar{t}_i/\bar{v}_i])$. After that, it applies the following axioms to normalizes the formula into the grammar in Fig. 2:

$$\begin{array}{l} (\kappa_1 \wedge \pi_1) * (\kappa_2 \wedge \pi_2) \equiv (\kappa_1 * \kappa_2) \wedge (\pi_1 \wedge \pi_2) \\ (\exists \bar{w}.\ \Delta_1) * (\exists \bar{v}.\ \Delta_2)\ \equiv \exists \bar{w}, \bar{v}'.\ (\Delta_1 * \Delta_2[\bar{v}'/\bar{v}]) \end{array}$$

The correctness of these axioms could be found in [23,38]. We then use $\mathtt{unfold}(\Delta) \equiv \bigcup_{i=1}^{n} \mathtt{unfold}(\Delta, \mathtt{P}_i(\bar{t}_i)), \mathtt{P}_i(\bar{t}_i) \in \Delta$. For example, given the above-specified precondition for method *remove*, we obtain 6 formulae shown in Fig. 3 after unfolding twice.

Unit Test Generation After unfolding, Γ contains a set of formulae, each of which satisfies the precondition. In the second phase, at lines 1-7, these formulae are transformed into test inputs. First, we check the satisfiability of each formula using a satisfiability solver $\mathtt{S2SAT}_{\mathtt{SL}}$ [28,30] at line 4. The result of the solver is a pair (`r`, `model`) where `r` is a *decision* of satisfiability and `model` is a symbolic model which serves as the evidence of the satisfiability. Intuitively, a symbolic model is a base formula where every variable is assigned a *symbolic* value. Formally, a symbolic model is a quantifier-free base formula Δ_m where Δ_m is satisfiable and for each variable v in Δ_m, if v has a reference type, Δ_m contains $v {\mapsto} c(...)$, or $v = v'$, or $v = \mathtt{null}$; otherwise, Δ_m contains $v = k$ with k is either a boolean or 32-bit integer constant.

At line 6, the symbolic model is transformed into a test input using procedure `toUnitTest`, which we present in details in App. 2. Fig. 4 shows two test inputs generated for the example shown in Fig. 1. These two test inputs correspond to the first two formulae shown in Fig. 3 (where x is assigned the default value 0).

```
public void test_remove1() throws Exception {
  BinarySearchTree obj = new BinarySearchTree();
  obj.root = null; int x = 0;
  obj.remove(x);
}

public void test_remove2() throws Exception {
  BinarySearchTree obj = new BinarySearchTree();
  obj.root = new BinaryNode();
  BinaryNode left_2 = null; BinaryNode right_3 = null;
  int element_1 = 0; int x = 0; obj.root.element = element_1;
  obj.root.left = left_2; obj.root.right = right_3;
  obj.remove(x);
}
```

Fig. 4. Two test inputs

$$
\begin{array}{rcl}
\mathit{datat} & ::= & \texttt{data}\ c\ \{\ (\mathit{type}\ v;)^*\ \} \\
\mathit{type} & ::= & \texttt{bool} \mid \texttt{int} \mid c \\
\mathit{prog} & ::= & \mathit{stmt}^* \\
\mathit{stmt} & ::= & v := e \mid v.f_i := e \mid \texttt{goto}\ e \mid \texttt{assert}\ e \mid \texttt{if}\ e_0\ \texttt{then}\ \texttt{goto}\ e_1\ \texttt{else}\ \texttt{goto}\ e_2 \\
 & & \mid v := \texttt{new}\ c(v_1, ..., v_n) \mid \texttt{free}\ v \\
e & ::= & k \mid v \mid v.f_i \mid e_1\ op_b\ e_2 \mid op_u\ e \mid \texttt{null}
\end{array}
$$

Fig. 5. A core intermediate language

The correctness of the algorithm, i.e., each generated test input is a valid one, is straightforward as each symbolic model obtained from the unfolding satisfies the original precondition, since each one is an under-approximation of a Δ in Γ.

4 Concolic Execution

Specification-based testing allows us to generate test inputs which cover some parts of the program. Some program paths however are unlikely to be covered with such test inputs without exploring the program code [46]. Thus, the second step of our approach is to apply concolic execution to cover the remaining parts of the program.

We take a program, a set of test inputs and a constraint tree as inputs. The constraint tree allows us to keep track of both explored nodes and unexplored nodes. Informally, the concolic execution engine executes the test inputs, expands the tree and then generates new test inputs to cover the unexplored parts of the tree. This process stops when there are no unexplored nodes in the tree or it times out.

For simplicity, we present our concolic engine based on a general core intermediate language. The syntax of the language is shown in Fig. 5, which covers common programming language features. A program in our core language includes several data structures and statements. Our language supports boolean and 32-bit integer as primitive types. Program statements include assignment, memory store, goto, assertion, conditional goto, memory allocation, and memory deallocation. Expressions are side-effect free and consist of typical non-heap expressions and memory load. We use $\mathtt{op_b}$ to represent binary operators, e.g., addition and subtraction, and $\mathtt{op_u}$ to represent unary operators, e.g., logical negation. k is either a boolean or 32-bit integer constant.

We assume the program is in the form of static single assignments (SSA) and omit the type-checking semantics of our language (i.e., we assume programs are well-typed in the standard way). Note that our prototype implementation is for Java bytecode, which in general can be translated to the core language (with unsupported Java language features are abstracted during the translation). The core language is easily extended to interprocedural scenario with method calls.

Execution Engine Our concolic execution engine incrementally grows the constraint tree. Formally, the constraint tree is a pair (V, E) where V is a finite set of nodes and E is a set of labeled and directed edges (v, l, v') where v' is a child of v. Having edge (v, l, v') means that we can transit from v to v' via an execution rule l. Each node in the tree is a concolic state in the form of a 6-tuple $\langle \Sigma, \Delta, s, pc, flag \rangle \iota$ where Σ is the list of program statements; Δ is the symbolic state (a.k.a. the path condition); s is the current valuation of the program variables (i.e., the stack); pc is the program counter; $flag$ is a flag indicating whether the current node has been explored or not and ι is the current statement. Note that Σ and s are mapping functions, i.e., Σ maps a number to a statement, and s maps a variable to its value.

Initially, the constraint tree has only one node $\langle \Sigma, \mathtt{pre}, \emptyset, 0, \mathtt{true} \rangle \iota_0$ where $\emptyset$ denotes an empty mapping function and ι_0 is the initial statement. Note that the initial symbolic state is the precondition. We start with executing the program concretely, with some initial test inputs (at least one), and build the constraint tree along the way. The initial test inputs may come from specification-based testing or be provided by the users. Before each execution, s is initialized with values according to the test input. In the execution process, given a node, our engine systematically identifies an applicable rule (based on the current statement) to generate one or more new nodes. If no rule matches (e.g., accessing a dangling pointer), the execution halts. Note that some of the generated nodes are marked explored whereas some are marked unexplored (depending on the outcome of the concrete execution).

After executing all initial test inputs, the engine searches for unexplored nodes in the tree. If there is one such node with symbolic state Δ, the engine solves Δ using a solver [28,30]. If Δ is satisfiable, the unexplored path is feasible and the symbolic model generated by the solver is transformed into a new test input (as shown in the Sect. 3). The new test input is then executed and the constraint tree is expanded accordingly. If Δ is unsatisfiable, the node is pruned from the tree. This process is repeated until there are no more unexplored nodes or it times out.

The growing of the tree is governed by the execution rules, which effectively defines the semantics of our core language. The detailed execution rules are presented in Fig. 6. One or more rules may be defined for each kind of statements in our core language. Each rule, applied based on syntactic pattern-matching, is of the following form.

$$\frac{\text{conditions}}{\text{current\_state} \leadsto \text{end\_state}_1, ..., \text{end\_state}_n}$$

$$[\textbf{C}{-}\textbf{CONST}]\,\frac{}{s \vdash k \Downarrow k} \quad [\textbf{C}{-}\textbf{VAR}]\,\frac{}{s \vdash v \Downarrow k} \quad [\textbf{C}{-}\textbf{NULL}]\,\frac{}{s \vdash \texttt{null} \Downarrow \texttt{null}}$$

$$[\textbf{C}{-}\textbf{UNOP}]\,\frac{s \vdash e \Downarrow k}{s \vdash op_u\ e \Downarrow\ op_u\ k} \quad [\textbf{C}{-}\textbf{BINOP}]\,\frac{s \vdash e_1 \Downarrow k_1 \quad s \vdash e_2 \Downarrow k_2}{s \vdash e_1\ op_b\ e_2 \Downarrow k_1\ op_b\ k_2}$$

$$[\textbf{C}{-}\textbf{LOAD}]\,\frac{s \vdash v \Downarrow l \quad s \vdash l.f_i \Downarrow k}{s \vdash v.f_i \Downarrow k} \quad [\textbf{C}{-}\textbf{FREE}]\,\frac{s \vdash v \Downarrow l \quad s'{=}s \setminus \{l.f_i \mapsto \_\}\ \forall i{=}1..n \quad \iota = \Sigma(pc+1)}{\langle \Sigma, \Delta, s, pc, \texttt{true}\rangle \texttt{free}\ v \rightsquigarrow \langle \Sigma, \Delta, s', pc+1, \texttt{true}\rangle \iota}$$

$$[\textbf{C}{-}\textbf{ASSIGN}]\,\frac{s \vdash e \Downarrow k \quad s'{=}s[v \leftarrow k] \quad \text{fresh}\ v' \quad e'{=}e[v'/v] \quad \Delta'{\equiv}\exists v'.\Delta[v'/v] \wedge v{=}e' \quad \iota{=}\Sigma[pc+1]}{\langle \Sigma, \Delta, s, pc, \texttt{true}\rangle v := e \rightsquigarrow \langle \Sigma, \Delta', s', pc+1, \texttt{true}\rangle \iota}$$

$$[\textbf{C}{-}\textbf{NEW}]\,\frac{\begin{array}{c}\text{fresh}\ l \quad \text{fresh}\ v' \quad \Delta'{\equiv}\exists v'.\Delta[v'/v] * v{\mapsto}c(v_1, ..., v_n) \\ s'_1{=}s[l.f_i \leftarrow (s \vdash v_i)]\ \forall i{=}1..n \quad s'{=}s'_1[v \leftarrow l] \quad \iota{=}\Sigma(pc{+}1)\end{array}}{\langle \Sigma, \Delta, s, pc, \texttt{true}\rangle v = \texttt{new}\ c(v_1, ..., v_n) \rightsquigarrow \langle \Sigma, \Delta', s', pc{+}1, \texttt{true}\rangle \iota}$$

$$[\textbf{C}{-}\textbf{STORE}]\,\frac{s \vdash v \Downarrow l \quad s \vdash e \Downarrow k \quad s'{=}s[l.f_i \leftarrow k] \quad \Delta'{\equiv}\Delta \wedge v.f_i{:=}e \quad \iota{=}\Sigma(pc+1)}{\langle \Sigma, \Delta, s, pc, \texttt{true}\rangle v.f_i = e \rightsquigarrow \langle \Sigma, \Delta', s', pc+1, \texttt{true}\rangle \iota}$$

$$[\textbf{C}{-}\textbf{GOTO}]\,\frac{s \vdash e \Downarrow k \quad \iota{=}\Sigma(k)}{\langle \Sigma, \Delta, s, pc, \texttt{true}\rangle \texttt{goto}\ e \rightsquigarrow \langle \Sigma, \Delta, s, k, \texttt{true}\rangle \iota}$$

$$[\textbf{C}{-}\textbf{ASSERT}]\,\frac{s \vdash e \Downarrow \texttt{true} \quad \Delta'{\equiv}\Delta \wedge e \quad \iota{=}\Sigma(pc{+}1)}{\langle \Sigma, \Delta, s, pc, \texttt{true}\rangle \texttt{assert}(e) \rightsquigarrow \langle \Sigma, \Delta', s, pc{+}1, \texttt{true}\rangle \iota}$$

$$[\textbf{C}{-}\textbf{TCOND}]\,\frac{s \vdash e_0 \Downarrow \texttt{true} \ \ s \vdash e_1 \Downarrow k_1 \ \ s \vdash e_2 \Downarrow k_2 \ \ \Delta_1{\equiv}\Delta \wedge e_0 \ \ \Delta_2{\equiv}\Delta \wedge \neg e_0 \ \ \iota_1{=}\Sigma(k_1) \ \ \iota_2{=}\Sigma(k_2)}{\begin{array}{c}\langle \Sigma, \Delta, s, pc, \texttt{true}\rangle \texttt{if}\ e_0\ \texttt{then goto}\ e_1\ \texttt{else goto}\ e_2 \rightsquigarrow \\ \langle \Sigma, \Delta_1, s, k_1, \texttt{true}\rangle \iota_1, \langle \Sigma, \Delta_2, s, k_2, \texttt{false}\rangle \iota_2\end{array}}$$

$$[\textbf{C}{-}\textbf{FCOND}]\,\frac{s \vdash e_0 \Downarrow \texttt{false} \ \ s \vdash e_1 \Downarrow k_1 \ \ s \vdash e_2 \Downarrow k_2 \ \ \Delta_1{\equiv}\Delta \wedge e_0 \ \ \Delta_2{\equiv}\Delta \wedge \neg e_0 \ \ \iota_1{=}\Sigma(k_1) \ \ \iota_2{=}\Sigma(k_2)}{\begin{array}{c}\langle \Sigma, \Delta, s, pc, \texttt{true}\rangle \texttt{if}\ e_0\ \texttt{then goto}\ e_1\ \texttt{else goto}\ e_2 \rightsquigarrow \\ \langle \Sigma, \Delta_1, s, k_1, \texttt{false}\rangle \iota_1, \langle \Sigma, \Delta_2, s, k_2, \texttt{true}\rangle \iota_2\end{array}}$$

Fig. 6. Execution rules: $\Sigma[x \leftarrow k]$ updates the mapping Σ by setting x to be k; `fresh` is used as an overloading function to return a new variable/address; $s \vdash e \Downarrow k$ denotes the evaluation of expression e to a concrete value k in the current context s

Intuitively, if the conditions above the line is satisfied, a node matching the current_state generates multiple children nodes.

In the following, we explain some of the rules in detail. In the rule [C−ASSIGN] which assigns the value evaluated from expression e to variable v, for the concrete state our system first evaluates the value of e based on the concrete state s prior to updating the state of v with the new value. For the symbolic state, it substitutes the current value of v to a fresh symbol v' prior to conjoining the constraint for the latest value of v. In the rule [C−NEW] which assigns new allocated object to variable v, for the concrete state our system updates the stack with an assignment of the variable to a fresh location. For the symbolic state, it substitutes the current value of v to a fresh symbol v' prior to spatially conjoining the points-to predicate for the latest value of v.

In the rule [C−LOAD] (resp. [C−STORE]) which reads from (resp. writes into) the field f_i of an object v, in the concrete state we implicitly assume that the corresponding variable of the field is $l.f_i$ where l is the concrete address of v and proceed accordingly. For the symbolic states, checking whether a variable has been allocated before accessed is much more complicated as the path condition (with the precondition) may include occurrences of inductive predicates (which represent unbounded heaps), so our system

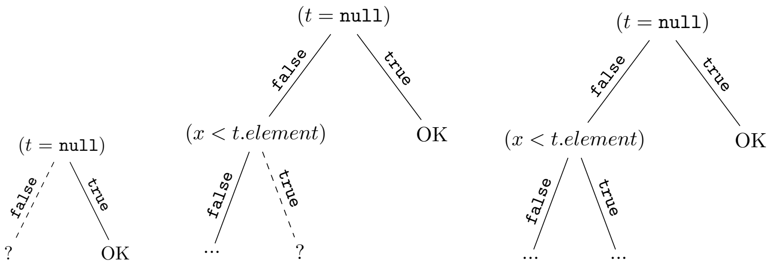

Fig. 7. Constraint trees construction: a question mark represents an unexplored path and OK denotes the execution terminates without error

keeps the constraints with the field-access form (i.e., $v.f_i$) and field-assign form (i.e., $v.f_i := e$) and will eliminate them before sending these formulae to the solver.

In the rule [C−TCOND], two new nodes denoting the `then` branch and the `else` branch of the condition are added into the tree with the current node is their parent. The symbolic states (path conditions) of both nodes are updated accordingly ($\varDelta_1$ and $\varDelta_2$). The concrete state s helps to identify that the execution is going to follow the `then` branch and marks this branch as explored. The remaining node is marked as unexplored. The rule [C−FCOND] is interpreted similarly.

For example, Fig. 7 show the constraint trees constructed during the concolic execution of the example in Fig. 1 with two initial test inputs in Fig. 4. The input of the first test case is an empty tree. The condition of the $\texttt{if} - \texttt{statement}$ at line 7 evaluates to `true`, satisfying the rule [C−TCOND]. The constraint tree in Fig. 7(a) is constructed. The input of the second test case is a tree with one node and x is 0. Thus the node is to be removed as its $element$ is 0. The rule [C−FCOND] is applied, which results in the tree in Fig. 7(b). The condition $x < t.element$ is then used to generate a new test input with $x = 0$ and $t.element = 1$. Executing this new test input triggers the rule [C−TCOND] at line 9, and updates the constraint tree as in Fig.7(c).

Path Condition Transformation Note that the path conditions generated according to the execution rules may contain field-access and field-assign expressions which are beyond the syntax in Fig. 2 and the support of the solver [28,30]. Thus, these expressions need to be eliminated. The details of the transformation are presented in the Algorithm 2. The input of the algorithm is a path condition which may contain field-access and field-assign expressions. The output are multiple path conditions, i.e., a disjunction of path conditions, without field-access and field-assign expressions.

The algorithm begins by recording all symbolic values for all fields of points-to predicates (lines 1-3). Then it considers each conjunct, which in form of a binary expression with left-hand side and right-hand side, in the path condition (line 4). In general, the field-access expression is substituted by symbolic value of the field. For each field-access expression $v.f_i$ in the conjunct (line 5), if the current path condition implies v is `null`, the path condition is unsatisfiable and is discarded (lines 6-7). In case the path condition implies v is constrained by a points-to predicate, it substitutes $v.f_i$

Algorithm 2: $\texttt{preprocess}(\Delta)$

```
1  $map \leftarrow \emptyset$
2  foreach $v \mapsto c(v_1, ..., v_n) \in \Delta$ do
3      $map \leftarrow map \cup \{v.f_i \leftarrow v_i\}$
4  foreach $(lhs\ op\ rhs) \in \Delta$ do
5      foreach $v.f_i \in (lhs\ op\ rhs)$ do
6          if $\Delta \Longrightarrow v = \texttt{null}$ then
7              return $\emptyset$
8          else if $map(v.f_i) = v_i \,\|\, map(x.f_i) = v_i \,\&\&\, \Delta \Longrightarrow v = x$ then
9              $(lhs\ op\ rhs) \leftarrow (lhs\ op\ rhs)[v_i/v.f_i]$
10         else if $\texttt{P}(\bar{v}) \in \Delta \,\&\&\, (v \in \bar{v} \,\|\, x \in \bar{v} \,\&\&\, \Delta \Longrightarrow v = x)$ then
11             $\Delta_s \leftarrow \texttt{unfold}(\Delta, \texttt{P}(\bar{v}))$, $\Gamma \leftarrow \emptyset$
12             foreach $\Delta_i \in \Delta_s$ do
13                 $\Gamma \leftarrow \Gamma \cup \texttt{preprocess}(\Delta_i)$
14             return $\Gamma$
15         else
16             return $\emptyset$
17     if $op\ is\ :=$ then
18         Substitute $lhs$ with a fresh symbolic name
19         Update the field in $map$ to the new name
20         Substitute $:=$ with $=$
21 return $\{\Delta\}$
```

with the corresponding symbolic name for the field in the predicate (lines 8-9). Otherwise, if v is constrained by an inductive predicate, it unfolds the predicate to find points-to predicate for v (lines 10-14). In the last case (lines 15-16), it considers the current path condition does not have enough information to resolve $v.f_i$ and simply returns empty. For field-assign expression $v.f_i := e$, after transforming the expression with above steps, it substitutes the left-hand side with a fresh symbolic name f_i', update the mapping from $v.f_i$ (or $x.f_i$ in case $\Delta \Longrightarrow x = v$) to f_i', then change $:=$ to $=$ (lines 17-20). Note that the update at line 19 may override the update at line 9 for left-hand side. Similar to Algorithm 1, the correctness of Algorithm 2 follows from the fact that each final path condition is an under-approximation of the original path condition because of the unfolding process. For instance, the path condition $\texttt{bst}(this\_root, minE, maxE) \wedge t = this\_root \wedge t \neq \texttt{null} \wedge x < t.element$ has field-access expression $t.element$ which need to be transformed. Using Algorithm 2, we get the final path condition which can be passed to the solver:

$$\exists elt, l, r.\ this\_root \mapsto BinaryNode(elt, l, r) * \texttt{bst}(l, minE, elt) * \texttt{bst}(r, elt, maxE) \wedge \\ minE < elt \wedge maxE > elt \wedge t = this\_root \wedge t \neq \texttt{null} \wedge x < elt$$

The solver verifies that the path condition is satisfiable and then returns a model which is a $BinarySearchTree$ with 1 node. The $element$ field of the node has value 1 and the value of parameter x is 0. The details of the transformation are shown in App. 3.

5 Implementation and Experiments

We have implemented our proposal in a tool, named Concolic StarFinder (CSF), with 6770 lines of Java code as a module inside the Java PathFinder framework. In the following, we conduct three experiments and contrast CSF's performance with existing approaches. All experiments are conducted on a laptop with 2.20GHz and 16 GB RAM. Instructions to obtain the artifact, which contains the tool source code, benchmarks and test scripts to replicate the experiments, are included in App. 4.

First Experiment In this experiment, we assume CSF is used as a stand-alone tool to generate test inputs for heap-manipulating programs. That is, the users provide a program and a precondition, then apply CSF to automatically generate a set of test inputs. The experimental subject is a comprehensive set of benchmark programs collected from previous publications, which includes *Singly-Linked List* (SLL), *Doubly-Linked List* (DLL), *Stack*, *Binary Search Tree* (BST), *Red Black Tree* (RBT) from SIR [7], *AVL Tree*, *AA Tree* (AAT) from Sireum/Kiasan [8], *Tll* from [27], the motivation example from SUSHI [10], the TSAFE project [17], and the Gantt project [3]. In total, we have 74 methods whose line of codes range from dozens to more than one thousand. For each method, the precondition according to the original publication is adopted for generating test inputs using CSF. In the specification-based testing stage, CSF is configured to generate all test inputs with a depth of 1 (e.g., unfolding the precondition once).

We compare CSF with two state-of-the-art tools, e.g., JBSE [12] and BBE [45]. JBSE uses HEX for specifying the invariants of valid test inputs and generates test inputs accordingly. We use the same invariants reported in [12] in our experiments. Note that because the HEX invariants for *SLL*, *Stack*, *BST*, *AA Tree* and *Tll* are not available[6], we skip running JBSE with these test subjects. BBE is explained in Section 2. In the following, we answer multiple research questions (RQ) through experiments.

RQ1: Does CSF generate valid test inputs? We apply CSF to generate test inputs for the 74 methods. To check whether the generated test inputs are valid, we validate the generated test inputs with the $repOK$ method in the data structures. The results are shown in the columns named $\#Tests$ in Table 1 for each test subject. The entries for JBSE and BBE are in the form of the number of valid test inputs over the total number of test inputs. As expected, all test inputs generated by CSF are valid. In comparison, JBSE generates 4.65% valid test inputs and BBE generates 7.83% valid test inputs. The reason for the poor results of JBSE and BBE is that the reference variables/fields are initialized with the wrong values or never initialized if they are not accessed. Note that by default, JBSE generates partially initialized test inputs, so we additionally call method $repOK$ to concretize them. CSF solves the path conditions, which contain the precondition, to generate test inputs, which are guaranteed to be valid. We thus conclude that using an expressiveness language is important in achieving validity.

RQ2: Can CSF achieve high code coverage? We use JaCoCo [4] to measure the branch coverage of the generated test inputs. The results are shown in the sub-columns named

[6] and it is unclear to us whether HEX is capable to specify them.

Table 1. Experiment 1 & 2: Results

Program	CSF				JBSE				BBE			
	#Tests	Cov.(%)	#Calls	T(s)	#Tests	Cov.(%)	NCov.(%)	T(s)	#Tests	Cov.(%)	NCov.(%)	T(s)
DLL	75	**100**	40/58	32	121/5146	56	100	206	0/35	0	21	21
AVL	62	**100**	36/654	274	76/295	**100**	100	48	17/117	70	89	69
RBT	133	**99**	14/1106	2403	137/291	87	91	38	14/380	26	53	333
SUSHI	5	**100**	3/38	8	0/900	0	100	24	2/27	25	25	8
TSAFE	16	**59**	1/595	1190	0/32	0	5	10	0/1	0	0	1
Gantt	22	**100**	2/156	25	17/887	55	90	24	0/6	0	5	2
SLL	29	**100**	21/8	11	-	-	-	-	16/50	66	71	19
Stack	18	**100**	16/2	7	-	-	-	-	11/14	84	84	6
BST	47	**100**	16/33	14	-	-	-	-	19/260	69	86	131
AAT	46	**99**	21/352	277	-	-	-	-	3/166	6	43	111
Tll	6	**100**	2/4	2	-	-	-	-	1/4	38	50	2
Math	320	**88**	576/0	73	-	-	-	-	128/320	75	79	95

$Cov.(\%)$ (which is the coverage achieved by valid test inputs) and $NCov.(\%)$ (which is the coverage achieved by all test inputs including the invalid ones) in Table 1. The winners are highlighted in bold. Note that for CSF, because all the test inputs are valid, we omit the column $NCov.(\%)$. The results show that CSF achieves nearly 100% branch coverage for almost all programs except TSAFE, whose coverage is 59.46%. For 70 out of 74 methods, CSF can obtain 100% branch coverage (including branches for auxiliary methods and excluding infeasible branches). CSF fails to cover 1 branch in two methods (i.e., $remove$ for RBT and $remove$ for AAT) and 3 branches in one method (i.e., put for RBT). The reason is that although the path conditions leading to those branches are satisfiable, the solver times out. For method TS_R_3, CSF achieves 59.46% branch coverage because in the execution, some native methods are invoked and applying symbolic execution to those paths are infeasible. Moreover, some of the path conditions contain string constraints which are not supported by the solver. For JBSE and BBE, the average coverage is 68.54% and 37.85% respectively if we consider valid test inputs only. If all test inputs are considered, the average coverage increases to 95.59% for JBSE and 54.66% for BBE. Note that the coverage is inflated with invalid test inputs.

RQ3: Is CSF sufficiently efficient? We measure the time needed to generate test inputs (sub-columns $T(s)$ in the Table 1). The results show that CSF needs 57.34 seconds on average for each program. The numbers for JBSE and BBE are 8.75 and 9.50 seconds respectively. Both JBSE and BBE are faster than CSF since they solve simpler constraints (e.g., without inductive predicates). However, their efficiency has a cost in term of the validity of the generated test inputs and the achieved code coverage. To conclude, we believe CSF is sufficiently efficient to be used in practice. We further show the number of solver calls used in CSF, i.e., the sub-column $\#Calls$ in the Table 1. The results are represented in form of the number of solver calls for specification-based testing over that of concolic execution. The results show that CSF needs 43 calls in average. Note that the number of solver calls in the specification-based testing stage varies according to the number of disjuncts in the precondition.

Second Experiment One infamous limitation of symbolic execution testing approach is it cannot handle programs with complex numerical conditions. On the other hand,

```
public boolean withCos(Node root) {
  while (root != null) {
    if (Math.cos(root.elem) == 1) return true;
    root = root.next; }
  return false;
}
```

Fig. 8. An example in the second experiment

specification-based testing approach does not suffer this limitation because it generates test inputs independently of programs under test. In this experiment, we aim to show the usefulness of specification-based testing in CSF, especially for programs with complex numerical conditions. To do that, we systematically compose a set of programs which travel a singly-linked list, apply a method from *java.lang.Math* library to the list elements, and check if the result satisfies some condition. One example is shown in Fig. 8 with method *cos*, which returns the cosin value of an integer. In total, we have 32 programs with 32 different methods from *java.lang.Math* library. We run CSF with only specification-based testing (to generate 10 test inputs) and compare the results with BBE. We cannot compare with JBSE because we do not have the HEX invariant for singly-linked list. However, we note that JBSE is a symbolic execution engine, which means it has difficulties in handling complex numerical conditions. The list elements has random values from -32 to 31 for all the tools. Due to randomness, we repeat the experiment 10 times for each program.

In average, while CSF obtains 88.28% branch coverage, BBE obtains 75.31%. The average number of solver calls is 18 and the average time is 2.27 seconds for each program. For BBE, it generates 10 test inputs for each program but only 4 of them satisfy *repOK* in 2.97 seconds. From the results, we conclude that the specification-based testing phase is useful, especially for programs with complex numerical conditions.

Third Experiment Although having a specification language based on separation logic allows us to precisely specify preconditions of the programs under test and generate valid test inputs, it could be non-trivial for ordinary users to use such a language. This problem has been recognized by the community and there have been multiple approaches to solve this problem [2,27,31,39]. One noticeable example which has made industrial impact is the Infer static analyzer [2], which infers preconditions of programs through bi-abduction [13]. In this experiment, we show that CSF can be effectively combined with Infer so that CSF can be applied without user-specified preconditions.

We first apply Infer to generate preconditions of the programs under test and then apply CSF to generate test inputs accordingly. The test subject is PLEXIL [5], i.e., NASA's plan automation and execution framework. Specifically, we analyze its verification environment PLEXIL5 [6] with Infer, and collect 88 methods that have explicit preconditions returned by Infer.

The experimental results are shown in Table 2, which are categorized based on the number of initial test inputs generated from Infer's preconditions (column $\#Init\,Tests$). The second column $\#Methods$ shows the number of methods in the category. The column $\#Tests$ shows the number of generated test inputs and the column $\#Exceptions$

Table 2. Experiment 3 with Infer: Results

#Init Tests	#Methods	#Tests	#Exceptions	#Calls	Time(s)
1	8	10	10	8/14	16
2	51	130	119	102/206	167
3	29	152	132	87/254	161

```
public void test_integerValue1() throws Exception {
  PlexilTreeParser obj = new PlexilTreeParser();
  plexil.PlexilASTNode _t = new plexil.PlexilASTNode();
  obj.ASTNULL = new antlr.ASTNULLType();
  int ttype_1 = 0;
  plexil.PlexilASTNode right_3 = null;
  plexil.PlexilASTNode down_2 = null;
  _t.ttype = ttype_1; _t.down = down_2; _t.right = right_3;
  obj.integerValue(_t);
}
```

Fig. 9. A test input which leads to $RuntimeException$

shows the number of exceptions in the category. Lastly, two columns $\#Calls$ and $Time(s)$ show the number of solver calls and the time needed to generate the test inputs respectively. In summary, CSF generates 292 test inputs in 344 seconds which achieved 58.36% branch coverage in average. Our investigation shows that all of these test inputs are valid according to the inferred preconditions. Interestingly, 261 out of the 292 test inputs (i.e., 89%) lead to $RuntimeException$ during execution. The interpretation can be either (1) the inferred preconditions are too weak to capture all the necessary conditions for valid test inputs generation, or (2) there are potential bugs in the programs.

To give an example, method $integerValue$ receives an Abstract Syntax Tree (AST) as input and the AST must contain an *INT* token. The inferred precondition only says that the input should not be `null`. One of the test inputs generated by CSF is shown in Fig. 9. The execution result is $RuntimeException$ because the value of field *ttype* does not match with the value of *INT* token, which is 108.

It would be interesting to develop a full integration of CSF and the recent bi-abduction for erroneous specification inference [39] so that we can generate meaningful test inputs automatically to witness bugs for any program.

6 Related Work

We review closely related work in the following, emphasis is given to approaches that generate test inputs for heap-manipulating programs.

Concolic testing programs with heap inputs This work is the first work that uses separation logic for concolic testing. The engineering design of our tool is based on that of JDart [32]. However, JDart, like most concolic execution engines, e.g., [18,19,24,33,42], does not support data structures as symbolic input for testing methods. Our work is related to CUTE [40] and Pex [43]. CUTE [40] does support data structures as input by using the so-called *logical input map* to keep track of input memory graph. However, CUTE cannot handle unbounded inputs nor capture the shape relations between pointers, which leads to imprecision. Pex [43] uses a type system [44] to describe disjointness of memory regions. But again, Pex cannot handle unbounded inputs. Moreover, the type system can only reason about the *global* heap, which leads to complex constraints and hence poor scalability. In comparison, our work handles unbounded inputs and shape relations are well-captured by separation logic predicates.

Lazy initialization As far as we know, lazy initialization [25] is the only way to handle unbounded inputs. However, most works in this direction, e.g., [15,16,21,45], did not address the problem of generating invalid test inputs due to the lack of constraints on the shapes of the input data structures. This work is related to the tool JSF presented in [35,36]. While JSF uses separation logic for specifying preconditions and apply classical symbolic execution, ours relies on concolic execution. Moreover, to support memory access, JSF unfolds those heaps accessed by reference variables in advance, our work prepares heap accesses via lazy unfolding which helps to encode both executed/not-yet-executed paths and heap accesses together. Another related work is [11] by Braione *et al.*, which we have discussed extensively in previous sections. The logic presented in [11], HEX, is not expressive enough to describe many popular data structures, including the binary search tree in our motivating example.

Specification-based testing has been an active research area for decades. Depending on the testing goals, different types of logic have been used as the specification languages to generate test inputs, for example Alloy [34], Java predicates [9], and temporal logic [20,22]. However, we are not aware of any existing work that generate test inputs from the specification in separation logic like ours.

Separation logic Research in separation logic focuses on static verification [13,14,27,29,37], which may return false positives and are not able to generate test inputs.

7 Conclusion

We have presented a novel concolic execution engine for heap-manipulating programs based on separation logic. Our engine starts with generating a set of initial test inputs based on preconditions. It concretely executes, monitors the executions and generates new inputs to drive the execution to unexplored code. We have implemented the proposal in CSF and evaluated it over benchmark programs. The experimental results show CSF's effectiveness and practical applications.

Acknowledgments. This research is supported by MOE research grant MOE2016-T2-2-123.

Appendix 1: Method *repOK* for *BinarySearchTree*

```
public class BinarySearchTree {
  public boolean repOK(BinaryNode t) {
    return repOK(t, new Range());
  }
  boolean repOK(BinaryNode t, Range range) {
    if (t == null) return true;
    if (!range.inRange(t.element)) return false;
    boolean ret = true;
    ret = ret && repOK(t.left, range.setUpper(t.element));
    ret = ret && repOK(t.right, range.setLower(t.element));
    return ret;
  }
}
```

Appendix 2: Symbolic model to test input

In the following, we show how to transform the symbolic model into a test input using procedure `toUnitTest`. In this transformation, we maintain a list of initialized variables. The transformation has three steps. Firstly, for each points-to predicate $v{\mapsto}c(...)$, we create a new object of type c and assign the new object to v. Similarly, for each predicate $v = \texttt{null}$ or $v = k$, we assign `null` or k to v respectively. After that, we add v into the list of initialized variables. Secondly, for each equality predicate $v_1 = v_2$, in case either v_1 or v_2 is not initialized, we find an initialized alias v for v_1 and v_2 in the model, then assign v to v_1 and v_2. In case v_1 and v_2 are not alias with any initialized variable, we create a new object with compatible type and assign it to v_1 and v_2. After that, both v_1 and v_2 are added into the list of initialized variables. Lastly, for each points-to predicate $v{\mapsto}c(v_1, ..., v_n)$, we assign v_i to $v.f_i$ for $i = 1..n$. Note that before this step, all variables v, v_1, ..., v_n are already initialized given the previous two steps.

Appendix 3: Path condition transformation example

We show the details of tranformation for the path condition

$$\texttt{bst}(\mathit{this\_root}, \mathit{minE}, \mathit{maxE}) \wedge t = \mathit{this\_root} \wedge t \neq \texttt{null} \wedge x < t.\mathit{element}$$

with field-access expression $t.\mathit{element}$.

From the path condition, we know that t is alias with $\mathit{this\_root}$ and is constrained by the predicate $\texttt{bst}(\mathit{this\_root}, \mathit{minE}, \mathit{maxE})$, so we unfold the predicate and get two new path conditions:

1. $\texttt{emp} \wedge \mathit{this\_root} = \texttt{null} \wedge t = \mathit{this\_root} \wedge t \neq \texttt{null} \wedge x < t.\mathit{element}$
2. $\exists \mathit{elt}, l, r.\ \mathit{this\_root}{\mapsto}\mathit{BinaryNode}(\mathit{elt}, l, r) * \texttt{bst}(l, \mathit{minE}, \mathit{elt}) * \texttt{bst}(r, \mathit{elt}, \mathit{maxE}) \wedge$
   $\mathit{minE} < \mathit{elt} \wedge \mathit{maxE} > \mathit{elt} \wedge t = \mathit{this\_root} \wedge t \neq \texttt{null} \wedge x < t.\mathit{element}$

In the first case, $\mathit{this\_root} = \texttt{null} \wedge t = \mathit{this\_root}$ so we cannot have symbolic value for $t.\mathit{element}$ and get rid of this path condition. Note that in this case, the path condition is unsatisfiable because it also contains $t \neq \texttt{null}$. In the second case, t is alias with $\mathit{this\_root}$, which points to a $\mathit{BinaryNode}$ with the symbolic value for field

$element$ is elt, so we substitute $t.element$ with elt and get the final path condition which can be passed to the solver:

$$\exists elt, l, r.\ this\_root \mapsto BinaryNode(elt, l, r) * \texttt{bst}(l, minE, elt) * \texttt{bst}(r, elt, maxE) \wedge \\ minE < elt \wedge maxE > elt \wedge t = this\_root \wedge t \neq \texttt{null} \wedge x < elt$$

Appendix 4: Instructions to replicate the experiments

We provide a Docker image containing the tool source code, benchmarks and test scripts to replicate our experiments. Instructions to install Docker on various platforms can be found in this link: `https://docs.docker.com/install/`.

Depending on the way Docker is installed, all the following Docker commands may need to be run with **sudo**.

1. Pulling the Docker image from Docker Hub
   ```
   docker pull artifact2019/concolic
   ```
2. Creating a Docker container
   ```
   docker run -ti artifact2019/concolic /bin/bash
   ```
   after this step, you will be inside the container and the current directory is
   ```
   /tools/jpf-costar
   ```
3. Running all the examples from the current directory
   ```
   bin/testAll.sh
   ```
4. The generated test inputs will be in the directory
   ```
   src/output
   ```

References

1. A fuzzer and a symbolic executor walk into a cloud. `https://blog.trailofbits.com/2016/08/02/engineering-solutions-to-hard-program-analysis-problems/`
2. Facebook Infer. `https://fbinfer.com/`
3. GanttProject. `https://github.com/bardsoftware/ganttproject`
4. JaCoCo. `https://www.eclemma.org/jacoco/`
5. PLEXIL. `http://plexil.sourceforge.net`
6. PLEXIL5. `https://github.com/nasa/PLEXIL5`
7. SIR. `http://sir.unl.edu/portal/index.php`
8. Sireum. `https://code.google.com/archive/p/sireum/downloads`
9. Boyapati, C., Khurshid, S., Marinov, D.: Korat: Automated Testing Based on Java Predicates. In: Frankl, P.G. (ed.) ISSTA 2002, pp. 123–133. ACM (2002). https://doi.org/10.1145/566172.566191
10. Braione, P., Denaro, G., Mattavelli, A., Pezzè, M.: Combining Symbolic Execution and Search-based Testing for Programs with Complex Heap Inputs. In: Bultan, T., Sen, K. (eds.) ISSTA 2017, pp. 90–101. ACM (2017). https://doi.org/10.1145/3092703.3092715
11. Braione, P., Denaro, G., Pezzè, M.: Symbolic Execution of Programs with Heap Inputs. In: Nitto, E.D., Harman, M., Heymans, P. (eds.) FSE 2015, pp. 602–613. ACM (2015). https://doi.org/10.1145/2786805.2786842

12. Braione, P., Denaro, G., Pezzè, M.: JBSE: A Symbolic Executor for Java Programs with Complex Heap Inputs. In: Zimmermann, T., Cleland-Huang, J., Su, Z. (eds.) FSE 2016, pp. 1018–1022. ACM (2016). https://doi.org/10.1145/2950290.2983940
13. Calcagno, C., Distefano, D., O'Hearn, P.W., Yang, H.: Compositional Shape Analysis by Means of Bi-Abduction. JACM **58**(6), 26:1–26:66 (2011). https://doi.org/10.1145/2049697.2049700
14. Chin, W.N., David, C., Nguyen, H.H., Qin, S.: Automated Verification of Shape, Size and Bag Properties via User-defined Predicates in Separation Logic. Sci. Comput. Program. **77**(9), 1006–1036 (2012). https://doi.org/10.1016/j.scico.2010.07.004
15. Deng, X., Lee, J., Robby: Bogor/Kiasan: A K-bounded Symbolic Execution for Checking Strong Heap Properties of Open Systems. In: ASE 2006, pp. 157–166. IEEE Computer Society (2006). https://doi.org/10.1109/ASE.2006.26
16. Deng, X., Robby, Hatcliff, J.: Towards A Case-Optimal Symbolic Execution Algorithm for Analyzing Strong Properties of Object-Oriented Programs. In: SEFM 2007. IEEE Computer Society (2007). https://doi.org/10.1109/SEFM.2007.43
17. Dennis, G.D.: TSAFE : Building a Trusted Computing Base for Air Traffic Control Software. Master's thesis, Massachusetts Institute of Technology, USA (2003)
18. Godefroid, P., Klarlund, N., Sen, K.: DART: Directed Automated Random Testing. In: Sarkar, V., Hall, M.W. (eds.) PLDI 2005, pp. 213–223. ACM (2005). https://doi.org/10.1145/1065010.1065036
19. Godefroid, P., Levin, M.Y., Molnar, D.: SAGE: Whitebox Fuzzing for Security Testing. Queue **10**(1), 20:20–20:27 (2012). https://doi.org/10.1145/2090147.2094081
20. Heimdahl, M.P.E., Rayadurgam, S., Visser, W., Devaraj, G., Gao, J.: Auto-generating Test Sequences Using Model Checkers: A Case Study. In: Petrenko, A., Ulrich, A. (eds.) FATES 2003, pp. 42–59. Springer (2003). https://doi.org/10.1007/978-3-540-24617-6_4
21. Hillery, B., Mercer, E., Rungta, N., Person, S.: Exact Heap Summaries for Symbolic Execution. In: Jobstmann, B., Leino, K.R.M. (eds.) VMCAI 2016, pp. 206–225. Springer (2016). https://doi.org/10.1007/978-3-662-49122-5_10
22. Hong, H.S., Lee, I., Sokolsky, O., Ural, H.: A Temporal Logic Based Theory of Test Coverage and Generation. In: Katoen, J.P., Stevens, P. (eds.) TACAS 2002, pp. 327–341. Springer (2002). https://doi.org/10.1007/3-540-46002-0_23
23. Ishtiaq, S.S., O'Hearn, P.W.: BI as an Assertion Language for Mutable Data Structures. In: Hankin, C., Schmidt, D. (eds.) POPL 2001, pp. 14–26. ACM (2001). https://doi.org/10.1145/360204.375719
24. Jayaraman, K., Harvison, D., Ganesh, V., Kiezun, A.: jFuzz: A Concolic Whitebox Fuzzer for Java. In: Denney, E., Giannakopoulou, D., Pasareanu, C.S. (eds.) NFM 2009, pp. 121–125 (2009)
25. Khurshid, S., Păsăreanu, C.S., Visser, W.: Generalized Symbolic Execution for Model Checking and Testing. In: Garavel, H., Hatcliff, J. (eds.) TACAS 2003, pp. 553–568. Springer (2003). https://doi.org/10.1007/3-540-36577-X_40
26. King, J.C.: Symbolic Execution and Program Testing. Commun. ACM **19**(7), 385–394 (1976). https://doi.org/10.1145/360248.360252
27. Le, Q.L., Gherghina, C., Qin, S., Chin, W.N.: Shape Analysis via Second-Order Bi-Abduction. In: Biere, A., Bloem, R. (eds.) CAV 2014, pp. 52–68. Springer (2014). https://doi.org/10.1007/978-3-319-08867-9_4
28. Le, Q.L., Sun, J., Chin, W.N.: Satisfiability Modulo Heap-Based Programs. In: Chaudhuri, S., Farzan, A. (eds.) CAV 2016, pp. 382–404. Springer (2016). https://doi.org/10.1007/978-3-319-41528-4_21
29. Le, Q.L., Sun, J., Qin, S.: Frame Inference for Inductive Entailment Proofs in Separation Logic. In: Beyer, D., Huisman, M. (eds.) TACAS 2018, pp. 41–60. Springer (2018). https://doi.org/10.1007/978-3-319-89960-2_3

30. Le, Q.L., Tatsuta, M., Sun, J., Chin, W.: A Decidable Fragment in Separation Logic with Inductive Predicates and Arithmetic. In: Majumdar, R., Kuncak, V. (eds.) CAV 2017, pp. 495–517. Springer (2017). https://doi.org/10.1007/978-3-319-63390-9_26
31. Le, X.D., Le, Q.L., Lo, D., Le Goues, C.: Enhancing Automated Program Repair with Deductive Verification. In: ICSME 2016, pp. 428–432. IEEE Computer Society (2016). https://doi.org/10.1109/ICSME.2016.66
32. Luckow, K., Dimjašević, M., Giannakopoulou, D., Howar, F., Isberner, M., Kahsai, T., Rakamarić, Z., Raman, V.: JDart: A Dynamic Symbolic Analysis Framework. In: Chechik, M., Raskin, J.F. (eds.) TACAS 2016, pp. 442–459. Springer (2016). https://doi.org/10.1007/978-3-662-49674-9_26
33. Marinescu, P.D., Cadar, C.: Make Test-zesti: A Symbolic Execution Solution for Improving Regression Testing. In: Glinz, M., Murphy, G.C., Pezzè, M. (eds.) ICSE 2012, pp. 716–726. IEEE Computer Society (2012). https://doi.org/10.1109/ICSE.2012.6227146
34. Marinov, D., Khurshid, S.: TestEra: A Novel Framework for Automated Testing of Java Programs. In: ASE 2001, pp. 22–. IEEE Computer Society (2001). https://doi.org/10.1109/ASE.2001.989787
35. Pham, L.H., Le, Q.L., Phan, Q.S., Sun, J., Qin, S.: Enhancing Symbolic Execution of Heap-based Programs with Separation Logic for Test Input Generation. In: ATVA 2019. To appear
36. Pham, L.H., Le, Q.L., Phan, Q.S., Sun, J., Qin, S.: Testing Heap-based Programs with Java StarFinder. In: Chaudron, M., Crnkovic, I., Chechik, M., Harman, M. (eds.) ICSE 2018, pp. 268–269. ACM (2018). https://doi.org/10.1145/3183440.3194964
37. Piskac, R., Wies, T., Zufferey, D.: Automating Separation Logic Using SMT. In: Sharygina, N., Veith, H. (eds.) CAV 2013, pp. 773–789. Springer (2013). https://doi.org/10.1007/978-3-642-39799-8_54
38. Reynolds, J.: Separation Logic: A Logic for Shared Mutable Data Structures. In: LICS 2002, pp. 55–74. IEEE Computer Society (2002). https://doi.org/10.1109/LICS.2002.1029817
39. Santos, J.F., Maksimovi, P., Sampaio, G., Gardner, P.: JaVerT 2.0: Compositional Symbolic Execution for JavaScript. PACMPL **3**(POPL), 66:1–66:31 (2019). https://doi.org/10.1145/3290379
40. Sen, K., Marinov, D., Agha, G.: CUTE: A Concolic Unit Testing Engine for C. In: Wermelinger, M., Gall, H.C. (eds.) FSE 2005, pp. 263–272. ACM (2005). https://doi.org/10.1145/1081706.1081750
41. Stephens, N., Grosen, J., Salls, C., Dutcher, A., Wang, R., Corbetta, J., Shoshitaishvili, Y., Kruegel, C., Vigna, G.: Driller: Augmenting Fuzzing Through Selective Symbolic Execution. In: NDSS 2016. The Internet Society (2016)
42. Tanno, H., Zhang, X., Hoshino, T., Sen, K.: TesMa and CATG: Automated Test Generation Tools for Models of Enterprise Applications. In: Bertolino, A., Canfora, G., Elbaum, S.G. (eds.) ICSE 2015, pp. 717–720. IEEE Computer Society (2015). https://doi.org/10.1109/ICSE.2015.231
43. Tillmann, N., De Halleux, J.: Pex-White Box Test Generation for .NET. In: Beckert, B., Hähnle, R. (eds.) TAP 2008, pp. 134–153. Springer (2008). https://doi.org/10.1007/978-3-540-79124-9_10
44. Vanoverberghe, D., Tillmann, N., Piessens, F.: Test Input Generation for Programs with Pointers. In: Kowalewski, S., Philippou, A. (eds.) TACAS 2009, pp. 277–291. Springer (2009). https://doi.org/10.1007/978-3-642-00768-2_25
45. Visser, W., Păsăreanu, C.S., Khurshid, S.: Test Input Generation with Java PathFinder. In: Avrunin, G.S., Rothermel, G. (eds.) ISSTA 2004, pp. 97–107. ACM (2004). https://doi.org/10.1145/1007512.1007526
46. Wang, X., Sun, J., Chen, Z., Zhang, P., Wang, J., Lin, Y.: Towards Optimal Concolic Testing. In: Chaudron, M., Crnkovic, I., Chechik, M., Harman, M. (eds.) ICSE 2018, pp. 291–302 (2018). https://doi.org/10.1145/3180155.3180177

47. Yun, I., Lee, S., Xu, M., Jang, Y., Kim, T.: QSYM : A Practical Concolic Execution Engine Tailored for Hybrid Fuzzing. In: Enck, W., Felt, A.P. (eds.) USENIX Security 2018, pp. 745–761. USENIX Association (2018)