Software Tools for Technology Transfer manuscript No.
(will be inserted by the editor)

Symbolic Execution with Abstraction

Saswat Anand[1], **Corina S. Păsăreanu**[2], **Willem Visser**[3]

[1] College of Computing, Georgia Institute of Technology e-mail: saswat@gatech.edu
[2] NASA Ames Research Center, Moffett Field, CA 94035 e-mail: pcorina@email.arc.nasa.gov
[3] SEVEN Networks e-mail: willem@gmail.com

The date of receipt and acceptance will be inserted by the editor

Abstract. We address the problem of error detection for programs that take recursive data structures and arrays as input. Previously we proposed a combination of symbolic execution and model checking for the analysis of such programs: we put a *bound* on the size of the program inputs and/or the search depth of the model checker to limit the search state space. Here we look beyond bounded model checking and consider state matching techniques to limit the state space. We describe a method for examining whether a symbolic state that arises during symbolic execution is *subsumed* by another symbolic state. Since the number of symbolic states may be infinite, subsumption is not enough to ensure termination. Therefore, we also consider abstraction techniques for computing and storing abstract states during symbolic execution. Subsumption checking determines whether an abstract state is being revisited, in which case the model checker backtracks – this enables analysis of an *under-approximation* of the program behaviors. We illustrate the technique with abstractions for lists and arrays. We also discuss abstractions for more general data structures. The abstractions encode both the shape of the program heap and the constraints on numeric data. We have implemented the techniques in the Java PathFinder tool and we show their effectiveness on Java programs. This paper is an extended version of [2].

1 Introduction

The problem of finding errors for programs that have heap structures and arrays as inputs is difficult since these programs typically have unbounded state spaces. Among the program analysis techniques that have gained prominence in the past few years are model checking with abstraction, most notably predicate abstraction [6, 7, 15], and static analysis [11, 27]. Both these techniques involve computing a property preserving abstraction that over-approximates all feasible program behaviors. While the techniques are usually used for *proving* properties of software, they are not particularly well suited for error detection – the reported errors may be spurious due to over-approximation, in which case the abstraction needs to be refined. Furthermore, predicate abstraction handles control-dependent properties of a program well, but it is less effective in handling dynamically allocated data structures and arrays [22].

On the other hand, static program analyses, and in particular shape analysis, use powerful *shape* abstractions that are especially designed to model properties of unbounded recursive heap structures and arrays, often ignoring the numeric program data. A drawback is that, unlike model checking, static analyses typically don't report counter-examples exhibiting errors.

We propose an alternative approach that enables discovery of errors in programs that manipulate recursive data structures and arrays, as well as numeric data. The approach uses symbolic execution to execute programs on un-initialized inputs and it uses model checking to systematically explore the program paths and to report counter-examples that are guaranteed to be feasible. We use abstractions to compute *under-approximations* of the feasible program behaviors, hence counter-examples to safety properties are preserved. Our abstractions encode information about the shape of the program heap (as in shape analysis) and the constraints on the numeric data.

We build upon our previous work where we proposed a combination of symbolic execution and model checking for analyzing programs with complex inputs [18, 23]. In that work we put a bound on the input size and (or) the search depth of the model checker. Here we look beyond bounded model checking and we study state matching techniques to limit the state space search. We propose a technique for checking when a symbolic state is sub-

sumed by another symbolic state. The technique handles *un-initialized*, or partially initialized, data structures (e.g. linked lists or trees) as well as arrays. Constraints on numeric program data are handled with the help of an off-the-shelf decision procedure. Subsumption is used to determine when a symbolic state is revisited, in which case the model checker backtracks, thus pruning the state space search.

Even with subsumption, the number of symbolic states may still be unbounded. We therefore define abstraction mappings to be used during state matching. More precisely, for each explored state, the model checker computes and stores an abstract version of the state, as specified by the abstraction mappings. Subsumption checking then determines if an abstract state is being revisited. This effectively explores an under-approximation of the (feasible) paths through the program. We illustrate symbolic execution with abstract subsumption checking for singly linked lists and arrays. Our abstractions are similar to the ones used in shape analysis: they are based on the idea of *summarizing* heap objects that have common properties, for example, summarizing list elements on unshared list segments not pointed to by local variables [22].

To the best of our knowledge, this is the first time shape abstractions are used in software model checking, with the goal of error detection. We summarize our contributions as follows:

- Method for comparing symbolic states, which takes into account uninitialized data. The method handles recursive structures, arrays and constraints on numeric data. The method is incorporated in our framework that performs symbolic execution during model checking.
- Abstractions for lists and arrays that encode the shape of the heap and the numeric constraints for the data stored in the summarized objects.
- Implementation in the Java PathFinder tool and examples illustrating the application of the framework on Java programs.

1.1 Related Work

Our work follows a recent trend in software model checking, which proposes under-approximation based abstractions for the purpose of *falsification* [4, 5, 16, 25]. These methods are complementary to the usual over-approximation based abstraction techniques, which are geared towards proving properties. There are some important differences between our work and [4, 5, 16, 25]. The works presented in [16, 25] address analysis of closed programs, not programs with inputs as we do here, and use abstraction mappings for state matching during concrete execution, not symbolic execution. Moreover, the approaches presented in [16,25] do not address abstractions for recursive data structures and arrays. The approach presented in [4, 5] uses predicate abstraction to compute under-approximations of programs. In contrast, we use symbolic execution and shape abstractions with the goal of error detection. And unlike [4, 5] and also over-approximation based predicate abstraction techniques, which require the a priori computation of the abstract program transitions, regardless of the size of the reachable state space, our approach uses abstraction only during state matching and it involves only the *reachable* states under analysis.

In previous work [24] we developed a technique for finding guaranteed feasible counter-examples in abstracted Java programs. That work addresses simple numeric abstractions (not shape abstractions as we do here) and it did not use symbolic execution for program analysis.

Program analysis based on symbolic execution has received a lot of attention recently, e.g. [12,19,28] - however all these approaches don't address state matching. Symstra [30] uses symbolic execution over numeric data and subsumption checking for test generation; we generalize that work with subsumption for un-initialized complex data; in addition, we use abstraction to further reduce the explored symbolic state space.

The works in [22, 31] propose abstractions for singly linked lists that are similar to the one described in this paper; however, unlike ours, these abstractions don't account for the numeric data stored in the summarized list elements. Recent work for summarizing numeric domains [13,14] addresses that in the context of arrays and recursive data structures. The work presented in [8] proposes to use predicate abstraction based model checking to programs that manipulate heap structures. However, these approaches use over-approximation based abstractions and it is not clear how to generate feasible counter-examples that expose errors.

1.2 Paper Layout

The rest of the paper is organized as follows. In the next section we give some background on the Java PathFinder model checker and its symbolic execution capability. Section 3 illustrates our approach on an example. Section 4 presents our algorithm for checking subsumption between symbolic states and Section 5 describes a general model checking procedure that uses symbolic execution with (abstract) subsumption checking. Section 6 describes abstractions for lists and arrays, Section 7 illustrates the application of the presented technique to two non-trivial examples containing lists and arrays and Section 8 concludes the paper.

2 Background

Java PathFinder JPF [17,29] is an explicit-state model checker for Java programs that is built on top of a custom-made Java Virtual Machine (JVM). By default, JPF

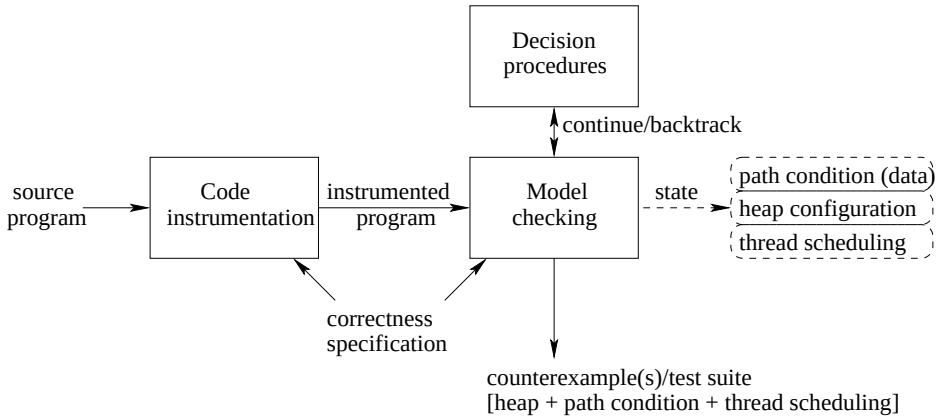

Fig. 1. Symbolic Execution in Java PathFinder.

stores all the explored states, and it backtracks when it visits a previously explored state. Alternatively, the user can customize the search (by forcing the search to backtrack on user-specified conditions) and it can specify what part of the state (if any) to be stored and used for matching. We used these features to implement (abstract) subsumption checking.

2.1 Symbolic Execution in Java PathFinder

Symbolic execution [20] allows one to analyze programs with unknown inputs. The main idea is to use *symbolic values*, instead of actual (concrete) data, as input values and to represent the values of program variables as symbolic expressions. As a result, the outputs computed by a program are expressed as a function of the symbolic inputs.

The state of a symbolically executed program includes the (symbolic) values of program variables, a *path condition* (`PC`) and a program counter. The path condition accumulates constraints which the inputs must satisfy in order for an execution to follow the corresponding path.

In previous work [18, 23], we extended JPF to perform symbolic execution for Java programs. The approach handles recursive data structures, arrays, numeric data and concurrency. The approach is illustrated in Figure 1. Programs are transformed to enable JPF to perform symbolic execution – concrete types are replaced with corresponding symbolic types and concrete operations, such as arithmetic and logical operations, are replaced with calls to methods that implement corresponding operations on symbolic expressions[1]. The path condition is updated at every branch statement in the program that compares symbolic values of program variables. Whenever the path condition is updated, it is checked for satisfiability using an appropriate decision procedure. In this work, we used the Omega library [26] for linear integer constraints, but other decision procedures can also be used [3]. If the path condition is unsatisfiable, the model checker backtracks. Note that if the *satisfiability* of the path condition cannot be determined as the problem of checking satisfiability is undecidable in general, the model checker still backtracks. Therefore, the model checker explores only *feasible* program behaviors, and all counterexamples to safety properties are preserved.

As described in [18], the approach is used for finding counterexamples to safety properties and for test input generation. For every counterexample, the model checker reports the input heap configuration (encoding constraints on reference fields and array elements), the numeric path condition (and a satisfying solution), and thread scheduling, which can be used to reproduce the error.

2.2 Lazy Initialization

Symbolic execution for complex data uses *lazy initialization*. The execution of a method that takes structurally complex inputs starts with inputs that have *uninitialized* fields of reference types. These fields are initialized lazily when they are first accessed during the method's symbolic execution. This allows symbolic execution of methods without requiring an a priori bound on the size of the structure of input.

When the execution accesses an un-initialized reference field, the framework nondeterministically initializes the field to

- *null*, or
- a reference to a new object with uninitialized fields of reference types, or
- a reference of an object created during a prior field initialization

This systematically accounts for all possible aliasing that may exist in the input structure. If a new object is cre-

[1] The interested reader is referred to [1, 18] for a detailed description of the code transformation.

```
   class Node {
    int elem;
    Node next; ...

    Node find(int v){
1:    Node n = this;
2:    while (n != null){
3:      if (n.elem > v)
          return n;
4:      n = n.next;
      }
5:    return null;
   }}
```

Fig. 2. Example illustrating symbolic execution with abstract subsumption checking

ated, and it has a field of primitive type then the field is assigned an symbolic value of appropriate type.

Arrays are handled in a similar way. During symbolic execution, an array is represented by a pair consisting of a symbolic value representing array's length and an association list of array cells. Each cell in the list consists of a symbolic integer representing the cell's index in the array and a symbolic value representing the value stored in the corresponding index in the array. When an array element is accessed (during read or write to the array), the framework non-deterministically chooses an array cell, which may be (1) a new cell that is created and added to the list, or (2) an existing cell from the list. The path condition is updated to encode the fact that index of the chosen cell equals to the index that was accessed. If the updated path condition becomes infeasible on any of the paths, that path is not explored. If the array access involves reading an element, the value of chosen cell is returned. Otherwise, if the access involves updating an element, the value of the chosen cell is updated appropriately. The fact that an array is represented as a list (linked list in particular) enables us to apply state matching algorithms (with and without abstractions) developed for linked lists to arrays with minor modifications.

Method preconditions are used during lazy initialization to ensure that the method is executed only on valid inputs – if the input structure violates the precondition, the model checker backtracks.

3 Example

We illustrate symbolic execution with abstract subsumption checking on the example from Figure 2. Class `Node` implements singly-linked lists of integers; fields `elem` and `next` represent, respectively, the node's value and a reference to the next node in the list. Method `find` returns the first node in the list whose `elem` field is greater than `v`. Let us assume for simplicity that the method has as precondition that the input list (pointed to by `this`) is non-empty and acyclic. We check if null pointer exceptions can be thrown in this program.

Figure 3 illustrates the paths that are generated during the symbolic execution of method `find` (we have omitted some intermediate states). Each symbolic state consists of a heap structure and the path condition (`PC`) accumulated along the execution path. A "cloud" in the figure indicates that the segment of the list pointed to by the `next` field is not yet initialized. The heap structures represent constraints on program variables and reference fields, e.g. the structure in s_1 represents all the lists that have at least one (non-null) element such that `n` points to the head of the list.

Branching corresponds to a nondeterministic choice that is made at branch points in the program that compare symbolic values, or to handle aliasing, during lazy initialization. For example, when the numeric condition at line `3` is executed symbolically execution splits into two paths leading to states s_2 and s_3 corresponding to each possible outcome of the condition's evaluation. As mentioned, branching is also introduced by lazy initialization. For example, at line `4`, the `next` field of the `Node` object pointed to by `n` in state s_2 is accessed for the first time. So the field is initialized to take into account all possible aliasing relationships in the input: on one branch (leading to state s_4), the "cloud" is replaced with a new node, whose `next` field points to a "cloud", while on the other branch (leading to s_5), the cloud is replaced with `null`. Note that if we did not impose the precondition that the input list is acyclic, there would have been a third branch corresponding to `next` pointing to the object pointed to by `n`.

For this example, the (symbolic) state space is infinite – the number of times the loop is executed is determined the length of the input linked list. And, the new states visited by the model checker in this example cannot be matched with previously visited sates (we define state matching on symbolic states, or *symbolic state subsumption* in section 4). So a model checker with symbolic execution and state matching will not terminate. However, if we use abstraction, the symbolic state space becomes finite. The list abstraction summarizes contiguous node segments that are not pointed to by local variables into a *summary* node. Since the number of local variables is finite, the number of abstract heap configurations is also finite. For the example, two nodes in state s_{12} are mapped to a summary node. As a result, the abstract state is subsumed by previously stored state s_8, at which point the model checker backtracks. The analysis terminates reporting that there are no null pointer exceptions. Note that due to abstract matching, the model checker might miss feasible behaviors. However, for this example, the abstraction is in fact *exact* – there is no loss of precision due to abstraction (all the successors of s_{12} are abstracted to states that are subsumed by the states depicted in Figure 3).

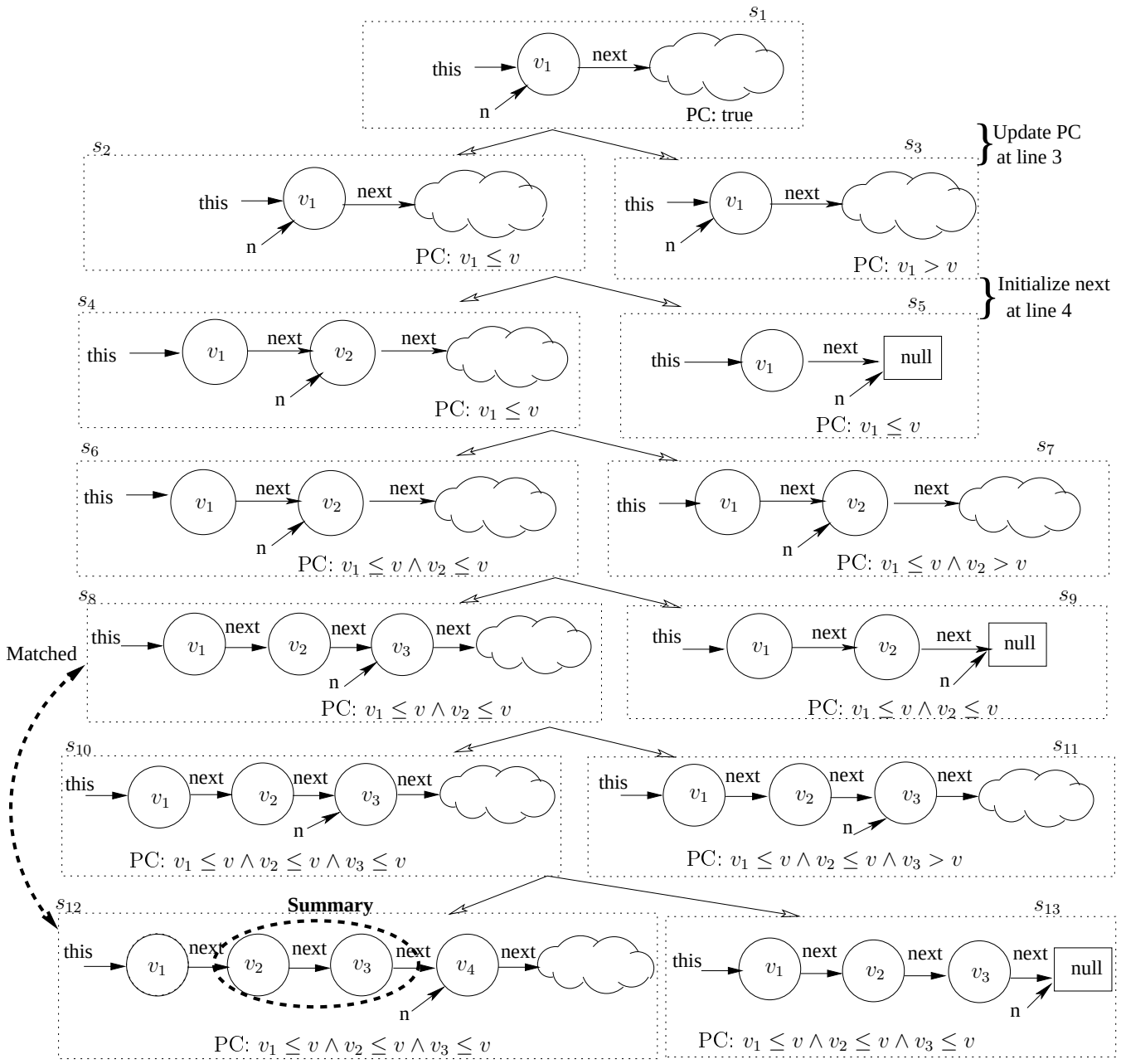

Fig. 3. State space generated during symbolic execution of `find` (excerpts)

4 Subsumption of Symbolic States

In this section we describe a method for comparing symbolic states. This method is used in our framework for state matching, during symbolic execution. The method is also used for comparing *abstracted* symbolic states as described in Section 6.

Symbolic states represent multiple concrete states, therefore state matching involves checking *subsumption* between states. Intuitively, a symbolic state s_1 *subsumes* another symbolic state s_2, if the set of concrete states represented by s_1 is a superset of the set of concrete states represented by s_2.

4.1 Symbolic State Representation

A symbolic state s consists of a *symbolic heap* H, the *valuation* of the primitive typed fields, the path condition PC and the program counter. The symbolic state also contains the thread scheduling information, which we ignore here for simplicity. Heaps may be partially initialized, and are assumed to be free of garbage objects.

Definition 1. A symbolic heap H is a graph represented by a tuple (N, E).

N is the set of nodes in the graph, where each node corresponds to a heap object or to a reference variable in the program. $N = N_O \cup R \cup \{null, uninit\}$[2] where:

- *null* and *uninit* are distinguished nodes that represent respectively, `null` and uninitialized objects.
- N_O is the set of nodes representing non-null initialized heap objects.
- R is the set of of nodes, each of which represent a reference variable in the program, and is referred to as *root* of the heap.

E is the set of edges in H such that $E = E_F \cup E_R$ where:

- F denotes the set of fields of reference types in the program. And, $E_F \subseteq (N_O \times F \times (N \setminus R))$ represent *field edges*. An edge $(n_1, f, n_2) \in E_F$ denotes that field f of the object represented by n_1 points to the object represented by n_2.
- $E_R \subseteq (R \times (N_O \cup \{null\}))$ represent *points-to edges*. An edge $(r, n_1) \in E_R$ represents the fact that reference variable `r` points to the object represented by n_1.

Note that there can be at most one outgoing edge per field from a node. By symbolic heap, we mean a heap that has *uninit* nodes. A symbolic heap represents a potentially infinite number of concrete heaps through the *uninit* node. We treat a concrete heap as a special symbolic heap that does not contain *uninit* nodes. In rest of the paper, symbolic heaps (e.g., H_1, H_2, etc.) represent the heap of the analyzed program, and thus have the same F and R sets that represent the set of fields and reference variables in the program respectively.

Definition 2. We define a partial order $\leq$ over symbolic heaps, such that $H_1 \leq H_2$ iff there exists a injective (total) function $\mu : N_O^{H_2} \rightarrow N_O^{H_1}$ such that for nodes $x, y \in N_O^{H_2}$, $f \in F$, and $v \in R$, following conditions hold:

1. $(x, f, y) \in E_F^{H_2} \Leftrightarrow (\mu(x), f, \mu(y)) \in E_F^{H_1}$
2. $(x, f, null) \in E_F^{H_2} \Leftrightarrow (\mu(x), f, null) \in E_F^{H_1}$
3. $(v, x) \in E_R^{H_2} \Leftrightarrow (v, \mu(x)) \in E_R^{H_1}$

Intuitively, $H_2 \leq H_1$ if there is a subgraph of H_1 containing all nodes pointed to by the nodes in R, which is isomorphic to H_2 modulo all *uninit* nodes and all edges incident on them. We say a concrete heap H_c is *represented by* a symbolic heap H_s iff $H_c \leq H_s$.

Definition 3. A symbolic heap H_2 subsumes a symbolic heap H_1, denoted by $H_1 \sqsubseteq H_2$, iff the set of concrete heaps represented by a H_2 contains all concrete heaps represented by H_1. Formally, $H_1 \sqsubseteq H_2 \equiv \forall H_c.\ H_c \leq H1 \Rightarrow H_c \leq H2$.

Corollary 1. $H_1 \sqsubseteq H_2 \equiv H_1 \leq H_2$.

[2] without loss of generality, we assume that the sets N_0, R, and {*null*,*uninit*} are mutually disjoint.

As mentioned, a symbolic state also includes the *valuation* for the primitive typed fields, e.g, `elem` field in Figure 2, (described later in this section) and the program counter. We check subsumption only for states that have the same program counter. Checking subsumption involves checking (1) subsumption for heap shape and (2) *valid* implication between the *state constraints* on the the symbolic states. While checking for shape subsumption, only the structure of the heap is considered (the symbolic values of primitive type fields are ignored). The symbolic values of the primitive typed fields are taken into account while checking implication between state constraints.

4.2 Subsumption of Heap Shapes

Data: Heaps $H_1 = (N^{H_1}, E^{H_1})$, $H_2 = (N^{H_2}, E^{H_2})$
Result: true if H_2 subsumes H_1, false otherwise. Also builds labeling l for matched nodes.
$l : (N_O^{H_1} \cup N_O^{H_2}) \rightarrow \mathbb{L} \cup \{nolbl\}$, where $\mathbb{L}$ is a set of labels $\{l_1, l_2, l_3...\}$.

```
1  begin
2      for n ∈ N_O^{H1} ∪ N_O^{H2} do l(n) := nolbl;
3      wl1 := mklist({n such that (r,n) ∈ E_R^{H1}});
4      wl2 := mklist({n such that (r,n) ∈ E_R^{H2}});
5      while wl2 is not empty do
6          if wl1 is empty then return false;
7          n1 := remove(wl1), n2 := remove(wl2);
8          if n2 = uninit then continue;
9          if n1 = uninit then return false;
10         if n1 = null ∧ n2 = null then continue;
11         if n1 = null ∨ n2 = null then return false;
12         if (l(n2) ≠ nolbl ∨ l(n1) ≠ nolbl) then
13             if l(n2) ≠ l(n1) then return false;
14             continue;
15         end
16         l(n2) := l(n1) := new_unique_label();
17         add (wl1, succs (n1));
18         add (wl2, succs (n2));
19     end
20     if wl1 is not empty then return false;
21     return true;
22 end
```

Algorithm 1: Subsumption for Heap Shape

In order to check if a program state s_2 subsumes another program state s_1, we first check if the heap shape H_2 of s_2 subsumes the heap shape H_1 of s_1. Intuitively, H_2 subsumes H_1 if H_2 is "more general" (i.e., represents more concrete heap shapes) than H_1. Subsumption for heap shape is checked by Algorithm 1.

The algorithm traverses the two heap graphs at the same time, in the same order, starting from the *roots* (nodes in R) and trying to *match* the nodes in the two structures. For simplicity, we consider here a depth first

search traversal. We impose an ordering on the reference variables and the heap graph is traversed from each of the roots in that order. The algorithm maintains two work lists wl_1 and wl_2 to record the visited nodes; The lists are initialized (through call to `mklist`) to an ordered list of heap objects pointed to by the variables in R. `remove` and `add` are list operations that remove the first element and add an element to the end of the list, respectively. We also impose an ordering on the fields of reference type from F. So the successors of a node n, $n \in N_O$, can be ordered by the order on their respective fields. `succs` in the algorithm returns the successors of a node using this ordering.

The algorithm *labels* the heap nodes during traversal, such that two matched nodes have the same *unique* label. These labels are used for checking state subsumption (as discussed below). If the algorithm finds two nodes that cannot be matched, it returns false. Moreover, whenever an uninitialized H_2 node is visited during traversal, the algorithm backtracks, i.e., successors of the node in H_1 that matches this uninitialized node are not added to the worklist (line 8); the intuition is that an uninitialized node *uninit* in H_2 can be matched with an arbitrary subgraph in H_1. However, an uninitialized node in H_1 can only match an uninitialized node in H_2 (line 9), and a `null` node in one heap can only match a `null` node in the other (lines 10,11). Lines 12-15 ensures that the matching is one-to-one. In other words, if either of n_1 or n_2 is already labeled because it is being revisited, then both of them must have been visited before and have same labels.

Note that, if the algorithm returns true, nodes in H_1 that are not visited due to matching with uninitialized nodes have *nolbl* labels; on the other hand, every node n, $n \in N_O^{H_2}$ is visited and thus has a label other than *nolbl*.

As an example, Figure 4 illustrates the shapes of several tree data structures. The double-headed dotted arrows connect the matched nodes in the structures. In the left example, the right tree subsumes the left tree. Whereas, in the right example, there is no subsumption relation between the two trees.

Theorem 1. *If Algorithm 1 returns true and labeling l for inputs H_1 and H_2 then H_2 subsumes H_1.*

Proof. (Sketch) Let $\mu : N_O^{H_2} \rightarrow N_O^{H_1}$ be such that $\mu(n_2) = n_1$ iff $l(n_2) = l(n_1)$. We show that μ satisfies the conditions in Definition 2, therefore $H_1 \leq H_2$ and, according to Corollary 1, $H_1 \sqsubseteq H_2$.

Note that μ is injective because two nodes in $N_O^{H_2}$ can not have the same label (line 16 in Algorithm 1). Moreover, μ is total: all nodes in $N_O^{H_2}$ are labeled (since Algorithm 1 performs a depth first search traversal of H_2 and it returns *true* only when wl_2 is empty).

We consider two cases:

- Let $(r, n_2) \in E_R^{H_2}$ and $(r, n_1) \in E_R^{H_1}$. The successors of r are added in the same order to wl_2 and wl_1 respectively. Therefore n_2 and n_1 are removed at the same time from wl_2 and wl_1 respectively at line 7. Therefore, either $n_2 = n_1 = null$ (line 10) or $l(n_2) = l(n_1)$ (line 14 or 16) or $n_2 = uninit$ (line 8) (in any other case, Algorithm 1 returns false).
- Let $(n_2, f, n_2') \in E_F^{H_2}$ and $(n_1, f, n_1') \in E_F^{H_1}$ such that $l(n_2) = l(n_1)$. Since n_1 and n_2 have the same label, it follows that their label was assigned in line 16 of the algorithm, their successors are added in the same order to wl_1 and wl_2, therefore n_2' and n_1' are removed from wl_2 and wl_1 respectively at line 7. Therefore, similar to the above, either $n_2' = n_1' = null$ (line 10) or $l(n_2') = l(n_1')$ or $n_2' = uninit$ (line 8).

If H_2 subsumes H_1 with a labeling l, we write $H_2 \sqsupseteq^l H_1$. Note that Algorithm 1 works on shapes represented as graphs that are *deterministic*, i.e. for each node, there is at most one outgoing edge for each field f, $f \in F$. Therefore, the algorithm applies to concrete heap shapes as well as partially initialized symbolic heap shapes (representing, linked lists, trees, etc.). The same algorithm also works on the abstractions for singly linked lists and arrays that we present in the Section 6 (since our abstractions preserve the deterministic nature of the heap).

4.3 Checking Subsumption of Numeric Constraints

Shape subsumption is only a pre-requisite of state subsumption: we also need to compare the numeric data stored in the symbolic states. In symbolic execution, the state contains symbolic values instead of concrete values for numeric variables and fields. The path condition of the state encodes constraints on these symbolic values. Due to the symbolic values, each symbolic state may *represent* a potentially infinite number of concrete states.

Let $primfld(n)$ denote all the fields of node n that have primitive types. For the purpose of this paper, we consider only integer types, but other primitive types can be handled similarly, provided that we have appropriate decision procedures. And let $v^s(n, f)$ denote the (symbolic) value stored in the integer field f of node n in state s.

Definition 4. The *valuation* of a node $n \in N_O$ in state s with respect to labeling $l : N_O \rightarrow \mathbb{L}$ is a constraint, defined as:

$$val^s(n, l) := \bigwedge_{f \in primfld(n)} fn(l(n), f) = v^s(n, f)$$

where, $fn(label, field)$ returns a fresh name that is unique to $(label, field)$ pair.

Definition 5. The *state constraint* SC_s^l of a state s with heap shape H and path condition PC is defined as:

$$SC^l := \exists v_s. \bigwedge_{\substack{n \in N_O^H \, s.t. \\ l(n) \neq nolbl}} val^s(n, l) \wedge PC$$

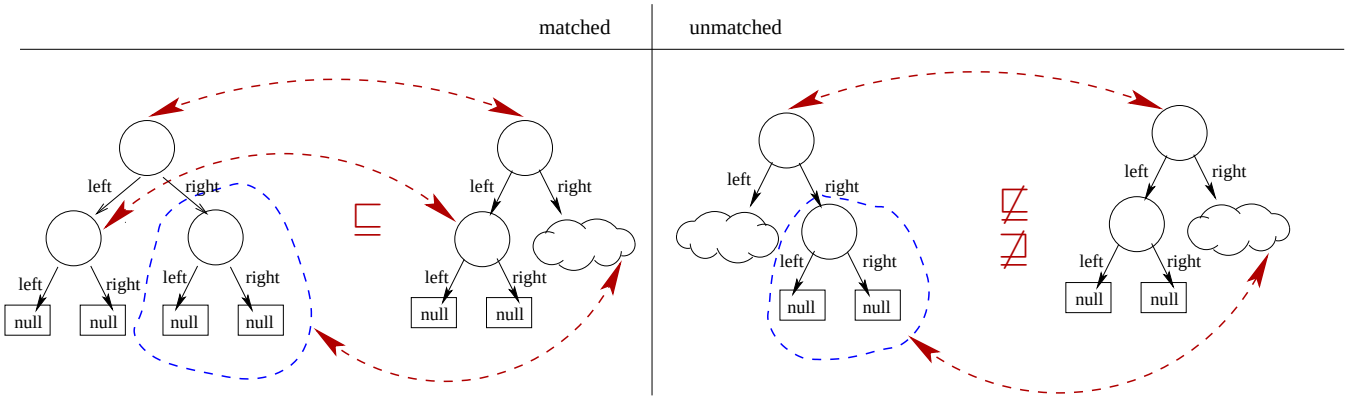

Fig. 4. Matched and unmatched heap shapes

where l is a labeling $l : N_O^H \rightarrow \mathbb{L} \cup \{nolbl\}$ and v_s denotes all the symbolic names that are used in symbolic state s; this includes both the values stored in the heap and the values that appear in the path condition.

As an example, consider two symbolic states in Figure 5, where s_1 is subsumed by s_2. The matched nodes from the two heaps have matching labels l_1 and l_2. The valuations for the two nodes labeled l_1 and l_2 in the left-hand side list are $e_1 = v_1$ and $e_2 = v_3$ respectively; e_1 and e_2 are the names computed by the function fn. Similarly, valuations for the two nodes with labels l_1 and l_2 in the right-hand side list are $e_1 = v_1$ and $e_2 = v_2$ respectively. The state constraint for s_1 and s_2 are respectively $\exists v_1, v_3, v_2 : e_1 = v_1 \wedge e_2 = v_3 \wedge v_1 < v_3 \wedge v_3 < v_2$ and $\exists v_1, v_2, v_5 : e_1 = v_1 \wedge e_2 = v_2 \wedge v_1 \leq v_5 \wedge v_5 \leq v_2$. Note that the path conditions may contain symbolic values that are not stored in the heap (e.g. v_5 in s_2) according to the program path that led to the symbolic state.

Definition 6. Let $Sol(SC_s^l)$ denote the set of satisfying solutions for the state constraint SC_s^l of state s for a labeling l. $SC_{s_2}^l$ subsumes $SC_{s_1}^l$ iff $Sol(SC_{s_1}^l) \subseteq Sol(SC_{s_2}^l)$ for *same* labeling $l : N_O^{H_1} \cup N_O^{H_2} \rightarrow \mathbb{L} \cup \{nolbl\}$.

Since in general, it may be computationally expensive/impossible to enumerate all the solutions of SC_1 and SC_2 and check for set inclusion, we rather check $SC_1 \Rightarrow SC_2$, which if *valid* ensures that $Sol(SC_1) \subseteq Sol(SC_2)$.

Now we combine the definitions of heap shape subsumption and state constraint subsumption to define state subsumption as follows:

Definition 7. A state s_1 is subsumed by another state s_2 (or s_2 subsumes s_1) iff $H_2 \sqsupseteq^l H_1$ and $SC_{s_1}^l \Rightarrow SC_{s_2}^l$.

In the example from Figure 5, as described before, Algorithm 1 returns true indicating that the heap shape of s_2 subsumes that of s_1. Matching nodes from the two states are labeled with l_1 and l_2. Notice that the third node in s_1 is not labeled due to the *uninit* node in s_2. To check for subsumption of state constraints we check if the implication between the state constraint of s_2 and that of s_1 is valid. State constraint of s_1 and s_2 simplifies to $e_1 < e_2$ and $e_1 <= e_2$ respectively. Since $e_1 < e_2 \Rightarrow e_1 <= e_2$ is valid, s_2 subsumes s_1.

The complexity for one subsumption step includes the complexity of heap traversal ($O(n)$ where n is the size of the heap) and the complexity for checking numeric constraints. While the cost of checking numerical constraints cannot be avoided, we believe that the cost of heap traversal can be somewhat alleviated if it is performed during garbage collection. However we need to experiment further with this idea.

5 Symbolic Execution with (Abstract) Subsumption Checking

Algorithm 2 illustrates the procedure for performing symbolic execution with (abstract) subsumption checking. The procedure checks if the input program P can reach an error state ϕ from initial state s_0. The procedure uses a depth first search order state exploration and it maintains a set of *VisitedStates* for the states visited so far and a *Stack* for storing the states to be explored. The procedure is similar to "classical" model checking state exploration, except that the explored states are symbolic, rather than concrete. The path condition on numeric data is checked for satisfiability to ensure exploration of feasible paths.

As discussed, we use state subsumption to determine if a state was visited before. Performing symbolic execution and subsumption checking during model checking may yield an unbounded number of symbolic states space. Therefore, we use abstractions to limit the model checker's search space. For each explored symbolic state s', the model checker computes an abstract state $\alpha(s')$, which is then stored for state comparison. Subsumption checking is used to compare the abstracted states, to determine if an abstract state is being re-visited. This effectively explores an under-approximation of the feasible paths through the program. Therefore, all the reported

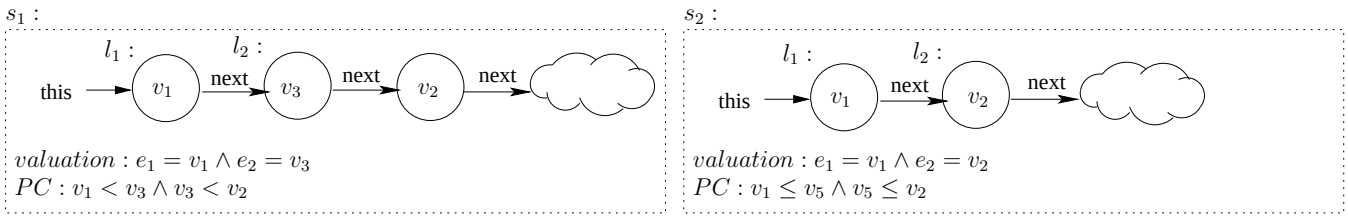

Fig. 5. State Subsumption

Data: Program P and error state ϕ
Result: Counterexample if ϕ is reachable

```
1  begin
2      add (α(s0), VisitedStates);
3      push (s0, Stack);
4      while Stack is not empty do
5          s := pop (Stack);
6          if s = φ then return counterexample;
7          foreach transition t enabled in s do
8              s':= successor (s, t);
9              if PathCondition(s') is not satisfiable
               then continue;
10             if there exists s'' ∈ VisitedStates s.t. α(s')
               subsumed by s'' then continue;
11             // s' not subsumed by any of the visited
               states
12             add (α(s'), VisitedStates);
13             push (s', Stack);
14         end
15     end
16 end
```

Algorithm 2: Symbolic Execution with (Abstract) Subsumption Checking

errors correspond to real errors in the analyzed program. Note however that the analysis might miss some errors, due to the imprecision of the abstraction.

6 Abstractions

6.1 Abstraction for Singly Linked Lists

The abstraction that we have implemented is inspired by [22,31] and it is based on the idea of summarizing all the nodes in a *maximally uninterrupted* list segment with a *summary* node. The main difference between [22, 31] and the abstraction presented here is that we also summarize the numeric data stored in the summarized nodes and we give special treatment to un-initialized nodes. The numeric data stored in the abstracted list is summarized by setting the valuation for the summary node to be a *disjunction* of the valuations of the summarized nodes. Intuitively, the numeric data stored in a summary node can be equal to that of any of the summarized nodes.

Shape subsumption between abstract states is done by Algorithm 1 as before, which treats summary node as any other node in the heap. For checking subsumption between numeric constraints, we introduce a new valuation function for the summary nodes as described before.

Definition 8. A node n is defined as an interrupting node, or simply an *interruption* if n satisfies at least *one* of following conditions:

1. $n = null$
2. $n = uninit$
3. $n \in \{m$ such that $(r, m) \in E_R\}$, i.e., n is pointed to by at least one reference variable.
4. $\exists n_1, n_2$ such that $(n_1, \texttt{next}, n), (n_2, \texttt{next}, n) \in E_F$. In other words, n is pointed-to by at least two nodes (cyclic list).

An uninterrupted list segment is a segment of the list that does not contain an interruption. An uninterrupted list segment $[u, v]$ is maximal if, $(a, \texttt{next}, u) \in E_F \Rightarrow a$ is an interruption and $(v, \texttt{next}, b) \in E_F \Rightarrow b$ is an interruption.

The abstraction for linked list replaces all maximally uninterrupted list segments in heap H with a summary node in the abstract state. If $[u, v]$ is a maximally uninterrupted list segment in H, the following transformations on H produces its abstract mapping.

1. A new *summary* node n_{sum} is added to the set of nodes N_O^H.
2. If there is an edge $(a, \texttt{next}, u) \in E_F^H$, a new edge $(a, \texttt{next}, n_{sum})$ is added to E_F^H.
3. If there is an edge $(v, \texttt{next}, b) \in E_F^H$, a new edge $(n_{sum}, \texttt{next}, b)$ is added to E_F^H.
4. All nodes m in the list segment $[u, v]$, and all edges incident on or going out of each m are removed from H.

Note that the edges between the nodes in the list segment, which are summarized by a summary node, are not represented in the abstraction state. With this abstraction, Algorithm 1 is used to check subsumption of shapes for abstracted heaps.

In order to check subsumption of numeric constraints, we define a *valuation* function for the summary nodes as follows. Let N_S, $N_S \subset N_O$, denote the set of summary nodes introduced in the heap during abstraction.

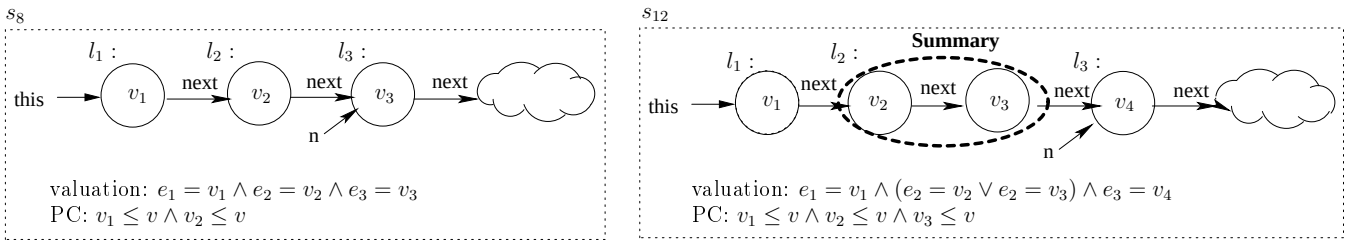

Fig. 6. Abstract subsumption between s_8 and s_{12}

Definition 9. The "valuation" of a summary node $n_{sum} \in N_S$ in state s, with respect to labeling $l : N_O \rightarrow \mathbb{L}$ is defined as:

$$val^s(n_{sum}, l) := \bigvee_{\substack{t \in sumnodes(n_{sum}) \\ f \in primflds(t)}} fn(l(n_{sum}), f) = v^s(t, f)$$

where, $sumnodes(n_{sum})$ denotes the set of nodes that are summarized by n_{sum}.

6.1.1 Example

To illustrate the approach, let us go back to the example presented in Section 3. Figure 6 depicts the abstract heap shape and the valuations of matched nodes for state s_{12}. The abstracted state is subsumed by state s_8 as there is a subsumption of heap shape, as represented by labelings of respective matching nodes and a valid implication between the normalized numeric constraints of the two states. Note that we don't explicitly summarize list segments of size one (e.g. the second list element in s_8); the abstracted and the un-abstracted states for s_8 are in fact the same.

6.1.2 Discussion

Note that the list abstraction ensures that the number of possible abstract heap configurations is finite; however, it is still possible to have an infinite number of states due to the numeric constraints. To address this issue, we plan to use predicate abstraction in conjunction with the abstractions presented here, to further abstract the numeric constraints. This is the subject of future work. Also note that the focus here is on abstracting heap structures. Therefore we ignored the numeric values of local program variables, which may also be unbounded (they are currently discarded in the abstracted state). Predicate abstraction can also be used for the local numeric variables.

6.2 Abstraction for Arrays

We extended our framework with subsumption checking and an abstraction for arrays of integers. The basic idea is to represent symbolic arrays as singly linked lists and to apply the (abstract) subsumption checking methods developed for lists. Specifically, we maintain the arrays as singly linked lists; nodes in the list represent individual array cells and their ordering in the list correspond to the order of indices of array cells they represent. Consecutive (initialized) array elements are represented as linked nodes. Summary nodes are introduced between array elements that are not consecutive. These summary nodes model zero or more uninitialized array elements that may possibly exist in the (concrete) array.

With the list representation of arrays we determine subsumption of program states with arrays as before. However, the *roots* are now integer program variables that are used to index the array, and the special summary nodes representing uninitialized array segments are treated as any other node in the heap N_O^H while checking for shape subsumption in Algorithm 1. Abstraction is applied in a way similar to abstraction for linked lists. The definition of interruption is extended to contain the special summary nodes.

We must note that this is only one particular abstraction, and there may be others – for example, abstractions based on array representations as ordered sequences of updates. We adopt this particular representation because in this way we can leverage on our abstraction techniques for lists. Note that subsumption becomes "approximate", i.e., we might miss the fact that a state subsumes another.

6.2.1 Array representation

A symbolic array A is represented by a symbolic value *len* representing the array length and an association list of array cells. Each array cell c is a pair $(index, elem)$: *index* is a symbolic value representing the index in the array and *elem* is a symbolic value representing the value stored in the array at position *index*.

The array cells are stored in a singly linked list which is sorted according to the relative order of the indices of the cells. Each list element corresponds to an array cell in A. Given array cell c, let $index(c)$ and $elem(c)$ denote the index and the value of c respectively; also let $next(c)$ denote the cell that is next to c in the list.

The following invariants hold for the list in a program state with path condition PC.

1. $PC \Rightarrow index(f) >= 0$ is valid, where f is the first cell in the list.

2. $PC \Rightarrow index(l) < len$ is valid, where l is the last cell in the list.
3. $PC \Rightarrow index(c) < index(next(c))$ is valid, where c is an cell other than the last cell in the list.

Note that our framework maintains these invariants through lazy initialization when array elements are accessed during symbolic execution. When an array is accessed with a symbolic index, the framework nondeterministically chooses (1) an existing cell from the list, or (2) a new cell, which is placed either at the beginning or end of the list, or between two cells that *may not* correspond to two consecutive elements of the array. In all cases the path condition is updated so that the above invariants hold for the new list. And as usual, if for any of the cases the updated path condition becomes unsatisfiable, the path is not explored.

To be able to check for subsumption between states containing arrays, we first apply a transformation (Algorithm 3) to the heaps. The transformation introduces special summary nodes, denoted by n_*, in the lists of array cells to represent uninitialized array segments. An array segment may be uninitialized if none of the array elements in that segment have been accessed so far. Algorithm 3 takes in the heap H^A that represents the heap of the program state containing a list of array cells representing an array A, and returns an transformed heap H'^A that contains additional special summary nodes. The transformation ensures that if two adjacent array cell c and $next(c)$ in H^A *may* represent non-consecutive array elements, then they are separated by a special summary node in H'^A. On the other hand, if c and $next(c)$ *must* represent two consecutive array elements, they are connected directly by a `next` link in H'^A (as in H^A). Whether two adjacent array cells c and $next(c)$ in H^A may represent non-consecutive array elements is determined by checking whether $PC \Rightarrow index(c) = index(next(c)) - 1$ is *invalid*; The formula is invalid if there exists some solution of PC that does not satisfy the constraint $index(c) = index(next(c)) - 1$, indicating that for that particular solution, indices of array cells c and $next(c)$ are not consecutive. The transformation also ensures that if the first(last) cell in H^A *may not* represent the first(last) element of the array A, a special summary node is added before(after) the cell in H'^A. If $PC \Rightarrow index(f) = 0$ is *invalid*, then it indicates that the first array cell f may not represent the first element of the array. Similarly, if $PC \Rightarrow index(l) = len - 1$ is *invalid*, then it indicates that the last cell l may not represent the last element of the array.

While checking for heap shape subsumption (Algorithm 1), the heap is traversed from the elements of R; each of which represents an reference-type variable in the program. However, with arrays elements of R may also represent integer-type program variables that index into an array. Formally, let I denote the set of integer-type program variables that index into an array, and $v^s(i), i \in I$ denote the (symbolic) value of i in state s.

Data: Sorted linked list $H^A = (N, E)$ representing array A
Result: Sorted linked list $H'^A = (N', E')$ that contains additional summary nodes representing uninitialized consecutive array elements

```
1  begin
2      foreach c in N_O do
3          add c to N'_O;
4          if c is the first element in H^A ∧
           PC ⇒ index(c) = 0 is invalid then
5              add a special summary node n_* before c in
               H'^A;
6          end
7          if c is the last element in A ∧
           PC ⇒ index(c) = len − 1 is invalid then
8              add a special summary node n_* after c in
               H'^A;
9          else
10             next(c) := cell following c in A;
11             if PC ⇒ index(c) = index(next(c)) − 1 is
               invalid then
12                 add a special summary node n_* after c
                   in A';
13             end
14         end
15     end
16 end
```

Algorithm 3: Building sorted linked lists representing symbolic arrays

Along with nodes representing reference-type (including array-type) variables, now the heap also contains one node for each variable i in I such it points to array cell c if $v^s(i) = index(c)$; R contains all such nodes, and as before, an arbitrary ordering on the elements of R is imposed for the traversal in Algorithm 1.

Abstraction over arrays is very similar to the one used for lists. It summarizes *maximally uninterrupted segments* corresponding to consecutive array elements. However, the definition for an interruption is slightly different, as it considers the special summary nodes introduced by Algorithm 3 as interruptions. Furthermore, we do not have to take into account heap shared nodes as in case of linked lists because arrays are represented by acyclic linked lists.

Definition 10. A node c in is an interruption if $c = n_*$, or $c = null$, or c is pointed to by a root $r \in R$.

Abstraction involves replacing all uninterrupted segments with a summary node (similar to list abstraction).

6.2.2 Example

Consider the symbolic array in Figure 7 (a); $v_0..v_5$ are symbolic values stored in the initialized array elements. The concrete values 0..3 and the symbolic values j and

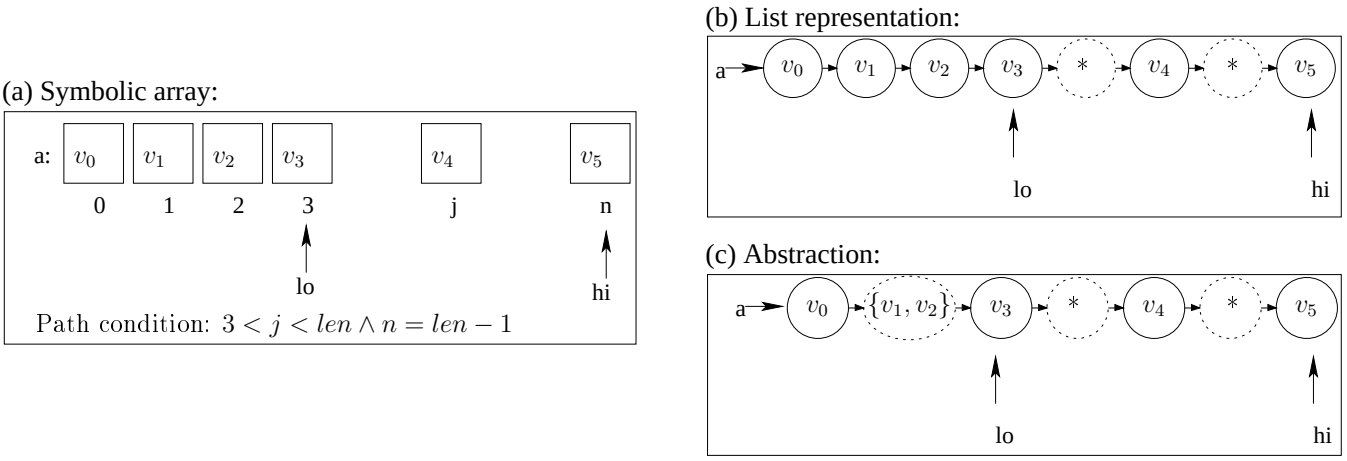

Fig. 7. A symbolic array (a), its list representation (b) and its abstraction (c)

n are array indices. Note that j and n are constrained by the path condition; *len* is a symbolic value representing the array length. Local program variables `lo` and `hi` are used to index the array. Figure 7 (b) shows the list representation for the symbolic array. The list is sorted according to the relative order of indices. The first four array elements are represented by nodes that are directly connected, because they represent consecutive array elements from indices 0 to 3. However, the 5th array element (containing value v_4) is separated from the other nodes a summary node (marked with a "*") on each side; it represents the fact that there may exist array elements before and after this element (i.e., between this element and the element with value v_3 and similarly, between this element and the element with value v_5), but have not been accessed so far during execution. Figure 7 (c) shows the abstracted list. The special summary nodes n_* and the nodes pointed to by nodes representing the program variables `a`, `lo`, `hi` are considered as interruptions. And thus, the second and third nodes of the list in (b) form a maximally uninterrupted segment and hence is summarized into a new node that is constrained to store a value that may be equal to the contents of either of the summarized nodes (i.e., v_1 or v_2).

6.3 Abstraction for General Data Structures

Our approach can be extended to more general data structures, e.g. by using an abstraction that is similar to the list abstraction. The idea again is to summarize all uninterrupted heap segments into summary nodes, where an interruption is one of the following:

- a node pointed to by a reference variable
- a node that is *heap shared* (i.e., a node that is pointed to by at least two other nodes)
- a node that represents *null* or *uninit*

As mentioned, the list abstraction that we use preserves the deterministic nature of the heap; therefore we can use Algorithm 1 for checking subsumption for abstract heap structures. However, this may not hold in general for other abstractions. For example, consider the abstraction of a tree structure. Each selector field of a summary node can now have a *set* of values (instead of only one) representing multiple outgoing heap edges. To address this issue, Algorithm 1 can be extended to support set of values of each selector field of summary nodes (i.e., by comparing the set sizes and by fixing an order for each set). This will yield a conservative approximation of subsumption checking: Algorithm 1 may fail to determine that a structure is in fact subsumed by another. We leave this extension for our future work. In future, we also plan to study the decidability of subsumption checking for more general heap abstractions (e.g., [21]) and extend our approach to these cases.

7 Experiments

We have implemented (abstract) subsumption checking on top of the symbolic execution framework implemented in JPF; the implementation uses the Omega library as a decision procedure. We applied our framework for error detection in two Java programs, that manipulate lists and arrays respectively.

The first program, shown in Fig. 8(a), is a list partition taken from [9]. The method takes as input an acyclic linked list `l` and an integer `v` and it removes all the nodes whose `elem` fields are greater than `v`; the removed elements are stored in a new list, which is pointed to by `newl`. A post-condition of the method is that each element in the list pointed to by `l` after method's execution must be less than or equal to v. We introduced a bug in the program so that the post-condition is not satisfied for the buggy program. The bug is activated by uncommenting the line `L1`.

In order to apply symbolic execution, we first instrumented the code, as shown in Fig. 8(b). Concrete types

```
class Node{
  Node next;
  int elem;
  ...
}

Node partition(Node l, int v){
  Node curr, prev, newl, nextCurr;
  prev = newl = null;
  curr = l;
  while(curr != null){
    nextCurr = curr.next;
    if(curr.elem > v){
      if(prev != null)
L1:       //if(nextCurr != null) //bug
            prev.next=nextCurr;
      if(curr == l) l = nextCurr;
      curr.next=newl;
      newl = curr;
    }
    else prev = curr;
    curr = nextCurr;
  }
  check();
  return l;
}
```

(a) Original code

```
class SymNode{
  SymNode next;
  Expression elem;
  ...
}

SymNode partition(SymNode l, Expression v){
  SymNode curr, prev, newl, nextCurr;
  prev = newl = null;
  curr = l;
  while(curr != null){
    Verify.ignoreIf(ifSubsumed(1));
    nextCurr = curr.get_next();
    if(curr.get_elem()._GT(v)){
      if(prev != null)
L1:       //if(nextCurr != null) //bug
            prev.set_next(nextCurr);
      if(curr == l) l = nextCurr;
      curr.set_next(newl);
      newl = curr;
    }
    else  prev = curr;
    curr = nextCurr;
  }
  symCheck();
  return l;
}
```

(b) Instrumented code

Fig. 8. List Partition Example.

are replaced with symbolic types (library classes that we provide), and concrete operations are replaced with method calls that implement equivalent symbolic operations. For example, classes `SymList` and `SymNode` implement symbolic `List`s and `Node`s respectively, while class `Expression` supports manipulation of symbolic integers.

Method `ifSubsumed` checks for state subsumption. It takes an integer argument that denotes the program counter, and it returns true only if the current program state is subsumed by a state which was observed before at that program point. If `ifSubsumed` returns true, then the model checker backtracks (as instructed by the `Verify.ignoreIf` method); otherwise, the current state is stored for further matching and the search continues. `check()` and its symbolic version `symCheck()` checks if the method's post-condition is satisfied.

Symbolic execution with abstract subsumption checking discovers the bug and it reports a counterexample of 10 steps, for an input list that has two elements, such that the first element is $\leq v$, and the second element is $> v$.

The second program, shown in Fig. 9(a), is an array partition taken from [4]. It is a buggy version of the `partition` function used in the QuickSort algorithm, a classic example used to study test generation. The function permutes the elements of the input array so that the resulting array has two parts: the first part contains values that are less than or equal to the chosen pivot value `a[0]`; while the second part has elements that are greater than the pivot value. There is an array bound check missing in the code at line `L2` that can lead to an array bounds error. The corresponding instrumented code is shown in Fig. 9(b) – class `SymbolicIntArray` implements symbolic arrays of integer, while `ArrayIndex` implements symbolic integers that are array indexes.

Symbolic execution with abstract subsumption checking reports a counterexample of 30 steps, for an input array that has four elements.

We also analyzed the corrected versions of the two partition programs to see whether symbolic execution with abstract subsumption checking terminates when the state-space is infinite, which is the case for the two programs. The state-exploration indeed terminates without reporting any error. For the list partition the analysis checked subsumption 23 times of which 11 states were found to be subsumed (12 unique states were stored). For the array partition the respective numbers were: 30 checks, with 17 subsumed and 13 states stored. This demonstrates the effectiveness of the abstractions in limiting the state space. We should note that subsumption checking without abstraction is not sufficient to limit the state space. This is in general the case for looping pro-

```
void parition(int[] a, int len){
  int tmp, pivot;
  int lo;
  int hi;
  //assume (n > 2);
  pivot = a[0];
  lo = 1;
  hi = n-1;
  while(lo <= hi){
L2: //while(a[lo] <= pivot) //bug
    while(lo <= hi &&
          a[lo] <= pivot){    //fix
      lo++;
    }
    while(a[hi] > pivot){
      hi--;
    }
    if(lo < hi){
      tmp = a[hi];
      a[hi] = a[lo];
      a[lo] = tmp;
    }
  }
}
```

(a) Original code

```
class ArrayCell{
  Expression index;
  Expression elem;
  ...
}
class ArrayIndex{
  ArrayCell cell;
  Expression index;
  ...
}
class SymbolicArray{
  LinkedList _v; //list of array cells
  Expression length;
}
void partition(SymbolicIntArray a, Expression len){
  Expression tmp, pivot;
  ArrayIndex lo = new ArrayIndex("lo");
  ArrayIndex hi = new ArrayIndex("hi");
  Verify.ignoreIf(n._LE(2));
  pivot = a.get(0);
  lo.assign(new IntegerConstant(1));
  hi.assign(n._minus(1));
  while(lo.index()._LE(hi.index())){
    Verify.ignoreIf(ifSubsumed(1));
L2:  //while(a.get(lo)._LE(pivot)){  //bug
    while(lo.index()._LE(hi.index()) &&
          a.get(lo)._LE(pivot)){    //fix
      Verify.ignoreIf(ifSubsumed(2));
      lo.assign(lo.index()._plus(1));
    }
    while(a.get(hi)._GT(pivot)){
      Verify.ignoreIf(ifSubsumed(3));
      hi.assign(hi.index()._minus(1));
    }
    if(lo.index()._LT(hi.index())){
      Expression tmp = a.get(hi);
      a.set(hi, a.get(lo));
      a.set(lo, tmp);
    } } } }
```

(b) Instrumented code

Fig. 9. Array Partition Example

grams. Although in theory, we should check for subsumption at every program point to get maximum savings, it may be very expensive.

In all our experiments, we checked for subsumption inside every loop only once, before the body of the loop is executed. The idea is similar to the use of *cutpoints* in deductive verification of programs [10].

We should note that these simple preliminary experiments show only the feasibility of the approach. A lot more experimentation and engineering is needed to be able to assess the merits of the approach on realistic programs. We should note however that even for such small examples, traditional testing methods would not discover the errors easily (e.g. a test-suite which gives 100% statement, or branch coverage might not be able to detect the errors).

8 Conclusion

We described a state space exploration approach that uses symbolic execution and subsumption checking for the analysis of programs that manipulate heap structures and arrays. The approach explores only *feasible* program behaviors. We also defined abstractions for lists and arrays, to further reduce the explored symbolic state space. We implemented the approach in the Java PathFinder tool and we applied it for error detection in Java programs.

The approach presented here is complementary to over-approximation abstraction methods and it can be used in conjunction with such methods, as an efficient way of discovering counter-examples that are guaranteed to be feasible. We view the integration of the two approaches as an interesting topic for future research. For the future, we plan to investigate how/if our approach extends to other shape abstractions and to use predicate abstraction for the numeric program data. We also plan to use our technique for systematic generation of complex test inputs (similar to [18]) and to characterize when there is loss of precision introduced by abstraction, for automatic abstraction refinement (similar to [25]). Moreover we plan to investigate the use of subsumption checking for *compositional analysis* of large programs. The presented abstractions were used in the context of *falsification*; however, we believe that they have merit in the context of verification - this could be achieved by storing the abstracted state and starting the symbolic execution from this abstracted state.

9 Acknowledgements

The first author was partially funded by NSF awards CCF-0429117 and CCF-0306372 to Georgia Institute of Technology during this work.

References

1. S. Anand, A. Orso, and M. J. Harrold. Type-dependence analysis and program transformation for symbolic execution. In *Proc. TACAS*, pages 117–133, 2007.
2. S. Anand, C. S. Pasareanu, and W. Visser. Symbolic execution with abstract subsumption checking. In *Proc. SPIN*, pages 163–181, 2006.
3. S. Anand, C. S. Pasareanu, and W. Visser. JPF-SE: A symbolic execution extension to Java Pathfinder. In *Proc. TACAS*, pages 134–138, 2007.
4. T. Ball. A theory of predicate-complete test coverage and generation. *MSR-TR-2004-28*, 2004.
5. T. Ball, O. Kupferman, and G. Yorsh. Abstraction for falsification. In *Proc. CAV*, pages 67–81, 2005.
6. T. Ball and S. K. Rajamani. The SLAM toolkit. In *Proc. CAV*, pages 260–264, 2001.
7. S. Chaki, E. Clarke, A. Groce, S. Jha, and H. Veith. Modular verification of software components in C. *ACM Trans. Computer Systems*, 30(6):388–402, 2004.
8. D. Dams and K. S. Namjoshi. Shape analysis through predicate abstraction and model checking. In *Proc. VMCAI*, pages 310–324, 2003.
9. C. Flanagan and S. Qadeer. Predicate abstraction for software verification. In *Proc. POPL*, pages 191–202, 2002.
10. R. W. Flyod. Assigning meanings to programs. In *Proc. Symposia in Applied Mathematics 19*, pages 19–32. American Mathematical Society, 1967.
11. R. Ghiya and L. J. Hendren. Is it a tree, a DAG, or a cyclic graph? a shape analysis for heap-directed pointers in c. In *Proc. POPL*, pages 1–15, 1996.
12. P. Godefroid, N. Klarlund, and K. Sen. DART: Directed automated random testing. In *Proc. PLDI*, pages 213–223, 2005.
13. D. Gopan, F. DiMaio, N. Dor, T. Reps, and M. Sagiv. Numeric domains with summarized dimensions. In *Proc. TACAS*, pages 512–529, 2004.
14. D. Gopan, T. Reps, and M. Sagiv. A framework for numeric analysis of array operations. In *Proc. POPL*, pages 338–350, 2005.
15. T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Software verification with BLAST. In *Proc. SPIN*, pages 235–239, 2003.
16. G. J. Holzmann and R. Joshi. Model-driven software verification. In *Proc. SPIN*, pages 76–91, 2004.
17. Java PathFinder. http://javapathfinder.sourceforge.net.
18. S. Khurshid, C. Păsăreanu, and W. Visser. Generalized symbolic execution for model checking and testing. In *Proc. TACAS*, pages 553–568, 2003.
19. S. Khurshid and Y. Suen. Generalizing symbolic execution to library classes. In *Proc. PASTE*, pages 103–110, 2005.
20. J. C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, 1976.
21. V. Kuncak and M. Rinard. Existential heap abstraction entailment is undecidable. In *Proc. SAS*, pages 418–438, 2003.
22. R. Manevich, E. Yahav, G. Ramalingam, and M. Sagiv. Predicate abstraction and canonical abstraction for singly-linked lists. In *Proc. VMCAI*, pages 181–198, 2005.
23. C. Păsăreanu and W. Visser. Verification of Java programs using symbolic execution and invariant generation. In *Proc. SPIN*, pages 164–181, 2004.
24. C. S. Păsăreanu, M. B. Dwyer, and W. Visser. Finding feasible abstract counter-examples. *STTT*, 5(1):34–48, 2003.
25. C. S. Păsăreanu, R. Pelánek, and W. Visser. Concrete model checking with abstract matching and refinement. In *Proc. CAV*, pages 52–66, 2005.
26. W. Pugh. The Omega test: A fast and practical integer programming algorithm for dependence analysis. *Commun. ACM*, 31(8), Aug. 1992.
27. S. Sagiv, T. W. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. *ACM Trans. Program. Lang. Syst.*, 24(3):217–298, 2002.
28. K. Sen, D. Marinov, and G. Agha. CUTE: A concolic unit testing engine for C. In *Proc. ESEC/SIGSOFT FSE*, pages 263–272, 2005.
29. W. Visser, K. Havelund, G. Brat, S. J. Park, and F. Lerda. Model checking programs. *Automated Software Engineering Journal*, 10(2):203–232, April 2003.
30. T. Xie, D. Marinov, W. Schulte, and D. Notkin. Symstra: A framework for generating object-oriented unit tests using symbolic execution. In *Proc. TACAS*, pages 365–381, 2005.
31. T. Yavuz-Kahveci and T. Bultan. Automated verification of concurrent linked lists with counters. In *Proc. SAS*, pages 69–84, 2002.