# Symbolic Execution with Abstract Subsumption Checking

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

[1] College of Computing, Georgia Institute of Technology
saswat@cc.gatech.edu
[2] QSS and RIACS, NASA Ames Research Center, Moffett Field, CA 94035
{pcorina, wvisser}@email.arc.nasa.gov

**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 subsumption is in general not enough to ensure termination, as the number of symbolic states may be infinite, 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. 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.

## 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 [3, 11, 4], and static analysis [23, 7]. 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 [18]. On the other hand,

A. Valmari (Ed.): SPIN 2006, LNCS 3925, pp. 163–181, 2006.

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 [14, 19]. 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 subsumed 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 [18].

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: *(i)* 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. *(ii)* Abstractions for lists and arrays that encode the shape of the heap and the numeric constraints for the data stored in the summarized objects. *(iii)* Implementation in the Java PathFinder tool and examples illustrating the application of the framework on Java programs.

**Related Work.** Our work follows a recent trend in software model checking, which proposes under-approximation based abstractions for the purpose of *falsification* [1, 2, 12, 21]. 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 [1, 2, 12, 21]. The works presented in [12, 21] 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 [12,21] do not address abstractions for recursive data structures and arrays. The approach presented in [1, 2] 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 [1, 2] 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 [20] 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. [8, 15, 24] - however all these approaches don't address state matching. Symstra [26] uses symbolic execution over numeric data and subsumption checking for test generation; we generalize that work with subsumption for uninitialized complex data; in addition, we use abstraction to further reduce the explored symbolic state space.

The works in [18, 27] 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 [9, 10] addresses that in the context of arrays and recursive data structures. The work presented in [5] 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.

## 2 Background

**Java PathFinder.** JPF [13, 25] 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 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.

**Symbolic Execution in Java PathFinder.** Symbolic execution [16] allows one to analyze programs with un-initialized 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 [14,19], we extended JPF to perform symbolic execution for Java programs. The approach handles recursive data structures, arrays, numeric data and concurrency. Programs are instrumented to enable JPF to perform symbolic execution; concrete types are replaced with corresponding symbolic types and concrete operations are replaced with calls to methods that implement corresponding operations on symbolic expressions[1]. Whenever a path condition is updated, it is checked for satisfiability using an appropriate decision procedure. We use the Omega library [22] for linear integer constraints, but other decision procedures can be used. If the path condition is unsatisfiable, the model checker backtracks. Note that if the *satisfiability* of the path condition cannot be determined (i.e., as it may be undecidable), 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 [14], 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 indices), the numeric path condition (and a satisfying solution), and thread scheduling, which can be used to reproduce the error.

**Lazy Initialization.** Symbolic execution of a method is started with inputs that have *un-initialized* fields; *lazy initialization* is used to assign values to these fields, i.e., fields are initialized 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 number of input objects.

When the execution accesses an un-initialized reference field, the algorithm nondeterministically initializes the field to *null*, to a reference to a new object with uninitialized fields, or to a reference of an object created during a prior field initialization; this systematically treats aliasing.

Lazy initialization for arrays proceeds in a similar way. Input arrays are represented by a collection of initialized array cells and a symbolic value representing the array's length. Each cell has a symbolic *index* and a symbolic *elem* value. When symbolic execution accesses an un-initialized cell, it initializes it nondeterministically to a new cell or to a cell that was created during a prior initialization; the path condition is updated with constraints that ensure that

[1] The interested reader is referred to [14] for a detailed description of the code instrumentation.

the index is within the array bounds and index of the the cell equals to the index that was accessed.

Method preconditions are used during lazy initialization to ensure that the method is executed only on valid inputs.

## 3 Example

We illustrate symbolic execution with abstract subsumption checking on the example from Figure 1. 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 2 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 introduced to build a path condition or to handle aliasing, during lazy initialization. For example, when the numeric condition at line `3` is executed symbolically there is a branch in execution for each possible outcome of the condition's evaluation (e.g. states $s_2$ and $s_3$). As mentioned, branching is also introduced by lazy initialization: for example, at line `4` the `next` field is accessed for the first time so it is initialized according to all the possible aliasing relationships in the inputs: on one branch, the "cloud" is replaced with a new node, whose `next` field points

```
   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. 1.** Example illustrating symbolic execution with abstract subsumption checking

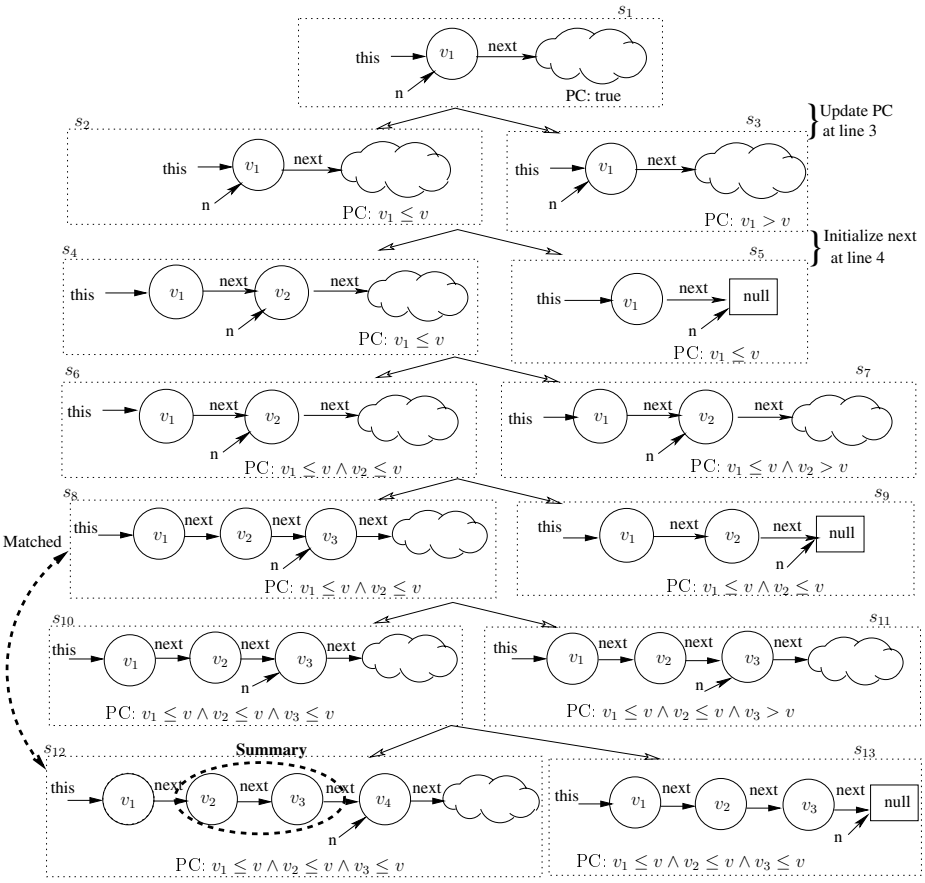


**Fig. 2.** State space generated during symbolic execution of `find` (excerpts)

to a "cloud", while on the other branch, the cloud is replaced with `null` (e.g. states $s_4$ and $s_5$). Note that if we wouldn't have imposed the precondition that the input list is acyclic, there had been a third branch corresponding to `next` pointing to itself.

The (symbolic) state space for this example is infinite and there is no subsumption between the generated symbolic states. 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 2).

## 4 Subsumption for 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 the next section).

Symbolic states represent multiple concrete states, therefore state matching involves checking *subsumption* between states. A symbolic state $s_1$ *subsumes* another symbolic state $s_2$, if the set of concrete states represented by $s_1$ contains the set of concrete states represented by $s_2$.

**Symbolic State Representation.** A symbolic state $s$ consists of a *symbolic heap configuration* $H$ and a path condition $PC$. The symbolic state also contains the program counter and thread scheduling information, which we ignore here for simplicity. Heap configurations may be partially initialized. Let $R$ and $F$ denote the set of all reference variables and object fields in the program respectively. We also assume that heap configurations are garbage free.

**Definition 1.** *A symbolic heap configuration $H$ 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 cell or to a reference program variable. $N = N_O \cup R \cup \{\text{null}, \text{uninit}\}$ where:*

- null *and* uninit *are distinguished nodes that represent respectively,* `null` *and objects not yet initialized.*
- *$N_O$ is the set of nodes representing dynamically allocated objects.*

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

- *$E_F \subseteq (N_O \times F \times (N \setminus R))$ represent selector* 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 $n_1$.*

A symbolic heap configuration represents a potentially infinite number of concrete heaps through the *uninit* node. Let $\gamma(H_S)$ denote all the concrete heaps $H$ represented by $H_S$. For symbolic heaps $H_2$, $H_1$: $H_2$ *subsumes* $H_1$ iff $\gamma(H_1) \subseteq \gamma(H_2)$.

A symbolic state also includes the *valuation* for the primitive typed fields (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 configurations (where we ignore the valuation of the primitive typed fields) and (2) *valid* implication between the numeric constraints encoded in the symbolic states.

**Data**: Heap Configurations $H_1 = (N^{H_1}, E^{H_1})$,$H_2 = (N^{H_2}, E^{H_2})$
**Result**: *true* if $H_2$ subsumes $H_1$, *false* otherwise; also builds labeling $l$ for matched nodes
$wl_1 := \{n$ such that $(r,n) \in E_R^{H_1}\}$, $wl_2 := \{n$ such that $(r,n) \in E_R^{H_2}\}$;
**while** $wl_2$ *is not empty* **do**
  **if** $wl_1$ *is empty* **then return** false;
  $n_1 :=$ `get`$(wl_1)$, $n_2 :=$ `get`$(wl_2)$;
  **if** $n_2 = uninit$ **then continue**;
  **if** $n_1 = uninit$ **then return** false;
  **if** $(l(n_2) \neq null \vee l(n_1) \neq null) \wedge l(n_2) \neq l(n_1)$ **then return** false;
  /* $n_1$, $n_2$ matched before: */
  **if** $(l(n_2) \neq null \wedge l(n_1) = l(n_2))$ **then continue**;
  **if** $n_1 = null \wedge n_2 = null$ **then continue**;
  **if** $n_1 \neq null \wedge n_2 \neq null$ **then**
    $l(n_2) := l(n_1) := new\_label()$;
    add successors of $n_1$, $n_2$ to $wl_1$, $wl_2$ respectively in the same order;
  **end**
  **else return** false;
**end**
**if** $wl_1$ *is not empty* **then return** false;
**return** *true;*

**Algorithm 1.** Subsumption for Heap Configurations

**Subsumption for Heap Configurations.** In order to check if a program state $s_2 = (H_2, PC_2)$ subsumes another program state $s_1 = (H_1, PC_1)$, we first check if heap configuration $H_2$ subsumes heap configuration $H_1$. Intuitively, $H_2$ subsumes $H_1$ if $H_2$ is "more general" (i.e., represents more concrete heap configurations) than $H_1$. Subsumption for heap configurations is described in Algorithm 1. The algorithm traverses the two heap graphs at the same time, in the same order, starting from the *roots* and trying to *match* the nodes in the two structures. Each of the reference variables from $R$ represents a *root* of the heap. We impose an order 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 with the heap objects pointed to by the variables in $R$; `get` and `add` are list operations that remove the first element and add an element to the end of the list, respectively.

The algorithm also *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). Let $l : (N_O^{H_1} \cup N_O^{H_2}) \rightarrow \mathbb{L} \cup \{null\}$, where $\mathbb{L}$ is a set of labels $\{l_1, l_2, l_3...\}$. If $H_2$ subsumes $H_1$ with a labelling $l$, we write $H_2 \sqsupseteq^l H_1$. 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; 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

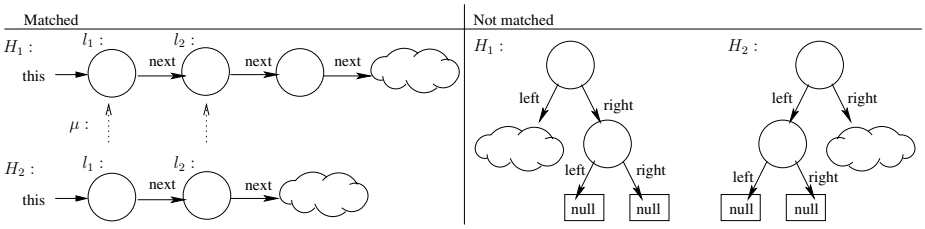


**Fig. 3.** Matched and unmatched heap configurations; $l_1$ and $l_2$ label matched nodes

in $H_2$. Note that nodes in $H_1$ which were not visited due to matching with uninitialized nodes are not labeled (i.e. they have *null* labels). As an example, Figure 3 shows the heap configurations for two matched lists and two unmatched binary trees. Figure 3 (left) illustrates the labeling for the two matched lists.

**Theorem 1.** *If Algorithm 1 returns true and labeling $l$ for inputs $H_1$ and $H_2$ then $H_2 \sqsupseteq^l H_1$.*

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 selector field. 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 next section (since our abstractions preserve the deterministic nature of the heap).

**Checking Validity 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. 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; $val^S(n, f)$ denotes the (symbolic) value stored in the integer field $f$ of node $n$ in state $S$.

**Definition 2.** *The "valuation" of a symbolic state $s$ parameterized by labeling $l : N_O \rightarrow \mathbb{N}$ is defined as:*

$$val(s, l) = \bigwedge_{\substack{n \in N_O s.t. l(n) \neq null \\ f \in primfld(n)}} fn(l(n), f) = val^s(n, f).$$

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

Let $v_s$ denote 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. In order to check validity for the numeric constraints, we use existential quantifier elimination for these symbolic variables to obtain the numeric constraints for a symbolic state.

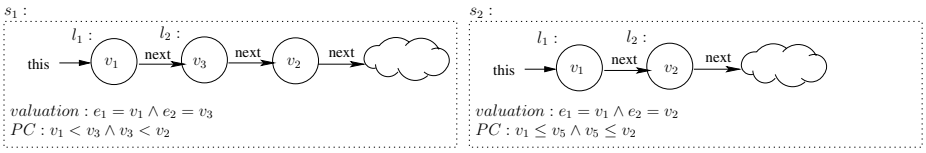


**Fig. 4.** State Subsumption

We are now ready to describe subsumption checking for symbolic states. A state $s_1 = (H_1, PC_1)$ is subsumed by another state $s_2 = (H_2, PC_2)$ (or $s_2$ subsumes $s_1$) if

1. $H_2 \sqsupseteq^l H_1$ and
2. $\exists v_{s_1}.val(s_1, l) \wedge PC_1 \Rightarrow \exists v_{s_2}.val(s_2, l) \wedge PC_2$.

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.

As an example, consider two symbolic states in Figure 4, where $s_1$ is subsumed by $s_2$. The corresponding heap configurations were matched and labeled as described before ($H_2$ subsumes $H_1$). The valuation encodes the constraints for the numeric fields, e.g. for the first list $e_1 = v_1 \wedge e_2 = v_3$ encodes that the `elem` field of the node labeled by $l_1$ (denoted by $e_1$) has symbolic value $v_1$ while the `elem` field of the node labeled by $l_2$ (denoted by $e_2$) has symbolic value $v_3$. The path condition puts further constraints on the symbolic values $v_1$ and $v_3$. 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.

For state comparison, we "normalize" the numeric constraints, i.e., we use the Omega library for existential quantifier elimination – intuitively, for this example, we are only interested in the relative order of the data stored in the matched heap nodes. For $s_1$ we compute $\exists v_1, v_3, v_2 : e_1 = v_1 \wedge e_2 = v_3 \wedge v_1 < v_3 \wedge v_3 < v_2$ which simplifies to $e_1 < e_2$. Note that since the third node in the list in $s_1$ was not matched it is not represented in the constraint. Similarly, for $s_2$, $\exists v_1, v_2, v_5 : e_1 = v_1 \wedge e_2 = v_2 \wedge v_1 \leq v_5 \wedge v_5 \leq v_2$ simplifies to $e_1 \leq e_2$. And as $e_1 < e_2 \Rightarrow e_1 \leq e_2$ is *valid* and $H_2$ subsumes $H_1$, $s_2$ subsumes $s_1$.

## 5 Abstractions

### 5.1 Abstraction for Singly Linked Lists

The abstraction that we have implemented is inspired by [18, 27] 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 [18, 27] and the abstraction presented here is that we also keep track of the numeric data stored in

the summary 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 symbolic nodes. Intuitively, a summary node stores the *union* of the values stored in the summarized nodes. Subsumption can then be used as before to perform state matching for abstract states (see Algorithm 1 where summary nodes are treated in the same way as the heap object nodes).

**Definition 3.** *A node $n$ is defined as an interrupting node, or simply an interruption if $n$ satisfies at least one of following conditions:*

1. $n = \text{null}$
2. $n = \text{uninit}$
3. $n \in \{m \text{ such that } (r, m) \in E_R\}$*, ie. $n$ is pointed to by at least one reference variable.*
4. $\exists n_1, n_2$ *such that* $(n_1, \texttt{next}, n), (n_2, \texttt{next}, n) \in E_F$*, ie. $n$ must be 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 mapping $\alpha$ between symbolic heap configurations replaces all maximally uninterrupted list segments in heap $H$ with a summary node in $\alpha(H)$. If $[u, v]$ is a maximally uninterrupted list segment in $H$, its abstraction $\alpha(H)$ is computed from $H$ as follows:

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

Note that the edges between the nodes in the list segment, which is replaced by a summary node, are not represented in the abstraction $\alpha(H)$. With this abstraction, Algorithm 1 is used to check subsumption for abstracted heaps.

In order to check validity of numeric constraints, the definition of *valuation* is modified as follows:

**Definition 4.** *Valuation for an abstract state $s$, parameterized by labeling $l$ is defined as,*

$$val^{abs}(s, l) = \bigwedge_{\substack{n \in (N_O \setminus N_S) s.t. l(n) \neq null \\ f \in \text{primflds}(n)}} fn(l(n), f) = val^s(n, f)$$

$$\bigwedge_{n_{sum} \in N_S s.t. l(n) \neq null} \bigvee_{\substack{t \in \text{sumnodes}(n_{sum}) \\ f \in \text{primflds}(t)}} fn(l(n_{sum}), f) = val^s(t, f)$$

*where, $N_S \subseteq N_O$ represents the set of summary nodes in $N_O$, and* $\text{sumnodes}(n_{sum})$ *denotes the set of nodes that are summarized by $n_{sum}$.*

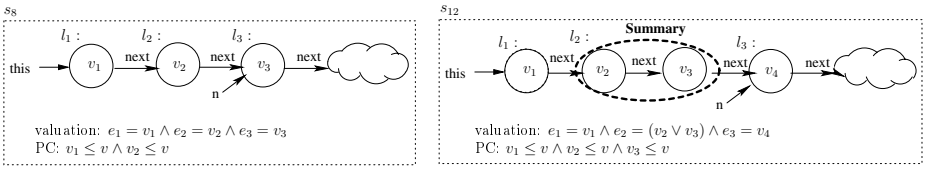


**Fig. 5.** Abstract subsumption between $s_8$ and $s_{12}$

**Example.** To illustrate the approach, let us go back to the example presented in Section 3. Figure 5 depicts the abstraction for state $s_{12}$ and the valuation for the abstracted heap configuration. The abstracted state is subsumed by state $s_8$. Note that we don't explicitly summarize list segments of size one (e.g. the second list element in $s_8$) - in this case, the abstracted and the un-abstracted versions of a symbolic state are in fact the same.

**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 different constraints over the numeric data. Also note that the focus here is on abstracting heap structures, more specifically lists, and the numeric data stored in these structures. Therefore we ignored here the numeric values of local program variables, which may also be unbounded (they are currently discarded in the abstracted state). To address these issues, we plan to use predicate abstraction in conjunction with the abstractions presented here. This is the subject of future work.

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 is not true in general for other abstractions (e.g. in a tree abstraction, a summary node may have multiple outgoing edges for the same selector field). In the future we plan to study the decidability of subsumption checking for more general heap abstractions – see e.g. [17] – and we plan to extend our approach to these cases (e.g. through a conservative approximation of the algorithm for subsumption checking).

### 5.2 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, which are *sorted* according to the relative order of the array indices. 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 un-initialized array elements that may possibly exist in the (concrete) array. We must note that this is only one particular abstraction, and there may be others – we adopt this one because in this way we can leverage on our abstraction techniques for lists.

With this list representation we apply subsumption as before. However, the "roots" are now integer program variables that are used to index the array and the special summary nodes are treated as "normal" heap objects that contain unconstrained values. Abstraction is applied in a way similar to abstraction for linked lists. The interruptions are extended to contain the special summary nodes that were introduced to model un-initialized array segments. Note that subsumption becomes "approximate", i.e., we might miss the fact that a state subsumes another, but it is never the case that we say that a state subsumes another state incorrectly.

**Array Representation.** A symbolic array $A$ is represented by a a collection of array cells and a symbolic value $len$ representing the array length. Each array cell $c$ is a tuple $(index, elem)$: $index$ 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 $c$; also let $next(c)$ denote the cell that is next to $c$ in the list.

The following invariants hold for the list.

1. $index$ of first node is greater than or equal to 0.
2. $index$ of last node is less than $len$.
3. For each array cell $c$, other than the last cell, $index(c) < index(next(c))$.

Note that our implementation maintains these invariants during lazy initialization, i.e., whenever symbolic execution accesses an un-initialized array cell, it initializes it non-deterministically to a previously created cell or to a new cell to be placed either between two existing cells that may not be consecutive, or at the end or at the beginning of the list. The path condition is also updated to encode this information. As discussed, in order to check subsumption, we further introduce additional summary nodes between nodes that represent non-consecutive array elements.

Algorithm 2 ensures that if two array cells $c_1$ and $c_2$ may represent non-adjacent array elements, then they are represented as list nodes separated by special summary nodes ($n_*$). On the other hand, if $c_1$ and $c_2$ represent two consecutive elements, they are connected directly by a `next` link. Similarly, if the first (last) cell of the array may not represent the first(last) element of the array, a special summary node is added before (after) the node.

With this transformation, we can apply subsumption checking as before. However, the "roots" of the heap representing the array now include a variable pointing to the head of the list (that represents the array), and all integer program variables that index array elements. These variables are the analog of the *reference variables* $r \in R$, and are denoted by $I.\ val^s(i), i \in I$ denotes the (symbolic) value of $i$.

Abstraction over arrays is very similar to the one used for lists. It summarizes *maximally uninterrupted segments* corresponding to consecutive array elements.

**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
**foreach** *c in* $N_O$ **do**
  add $c$ to $N'_O$;
  **if** *c is the first element in* $H^A \wedge PC \Rightarrow index(c) = 0$ *is invalid* **then**
    add a special summary node $n_*$ before $c$ in $H'^A$;
  **end**
  **if** *c is the last element in* $A \wedge PC \Rightarrow index(c) = len - 1$ *is invalid* **then**
    add a special summary node $n_*$ after $c$ in $H'^A$;
  **else**
    $next(c)$ := cell following $c$ in A;
    **if** $PC \Rightarrow index(c) = index(next(c)) - 1$ *is invalid* **then**
      add a special summary node $n_*$ after $c$ in $A'$;
    **end**
  **end**
**end**

**Algorithm 2.** Building sorted linked lists representing symbolic arrays

However, the definition for an interruption is slightly different, as it considers the special summary nodes introduced by Algorithm 2 as interruptions.

**Definition 5.** *A node c in is an interruption if* $c = n_*$, *or* c=*null, or c represents an array cell such that* $\exists i \in I.val^s(i) = index(c)$.

Abstraction involves replacing all uninterrupted segments with a summary node (similar to list abstraction). Note that this abstraction can be improved further, by mapping all contiguous segments of (summary and non-summary) nodes that are not pointed to by local variables to a (new) summary node.

**Example.** Consider the symbolic array in Figure 6 (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 $n$ 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 6 (b) shows the list representation for the symbolic array. The list is sorted according to the relative order of indices.

In the example the first four array elements are represented by nodes that are directly connected, as they have consecutive indices. However, the 5th array element is separated from the other nodes by two summary nodes (marked with a "*"). Note that unlike *uninit* nodes, these summary nodes are not completely unconstrained (we know their relative order in the array). We use the Omega library to decide if two elements have consecutive indices (in which case they are directly connected in the list). Figure 6 (c) shows the abstracted list, obtained with the method described before, where we consider the two variables `lo` and `hi` as interruptions; the "*" nodes are also considered interruptions.

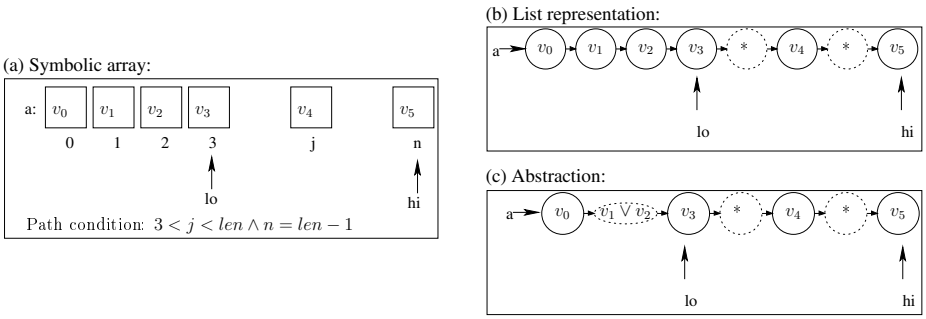


**Fig. 6.** A symbolic array (a), its list representation (b) and its further abstraction (c)

## 6 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. 7(a), is a list partition taken from [6]. 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. This post-condition is not satisfied for the buggy program.

In order to apply symbolic execution, we first instrumented the code, as shown in Fig. 7(b). Concrete types 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. 8(a), is an array partition taken from [1]. 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

```
class ListPartition{
  List list = new List();
  Node l = list.root;
  Node curr, prev, newl, nextCurr;
  int v;
  public void partition(){
    prev = newl = null;
    curr = l;
    while(curr != null){
      nextCurr = curr.next;
      if( curr.elem > v ){
        //if(prev != null &&
        //  nextCurr != null) //bug
        if(prev != null)      //fix
          prev.next = nextCurr;
        if(curr == l) l = nextCurr;
        curr.next = newl;
        newl = curr;
      }
      else prev = curr;
      curr = nextCurr;
    } check();} }
```

(a) Original code

```
class ListPartition{
  SymList list = new SymList();
  SymNode l = list.root();
  SymNode curr, prev, newl, nextCurr;
  Expression v = new SymbolicInteger();
  public void partition(){
    prev = newl = null;
    curr = l;
    while(curr != null){
      Verify.ignoreIf(ifSubsumed(1));
      nextCurr = curr.get_next();
      if(curr.elem()._GT(v)){
        //if(prev != null &&
        //   nextCurr != null)     //bug
        if(prev != null)           //fix
          prev.set_next(nextCurr);
        if(curr == l) l = nextCurr;
        curr.set_next(newl);
        newl = curr;
      }
      else  prev = curr;
      curr = nextCurr;
    } symCheck(); } }
```

(b) Instrumented code

**Fig. 7.** List Partition Example

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. 8(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 programs. 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.

```
class ArrayPartition{
  int[] a;
  int n, tmp, pivot;
  int lo;
  int hi;
  public void parition(){
    //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 ArrayPartition{
  SymbolicIntArray a;
  Expression pivot, n, tmp;
  ArrayIndex lo = new ArrayIndex("lo");
  ArrayIndex hi = new ArrayIndex("hi");

  public void partition(){
    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. 8.** Array Partition Example

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 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).

## 7 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 [14]) and to characterize when there is loss of precision introduced by abstraction, for automatic abstraction refinement (similar to [21]). 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.

## References

1. T. Ball. A theory of predicate-complete test coverage and generation. *MSR-TR-2004-28*, 2004.
2. T. Ball, O. Kupferman, and G. Yorsh. Abstraction for falsification. In *Proc. of CAV'05*, 2005.
3. T. Ball and S. K. Rajamani. The slam toolkit. In *Proc of CAV '01*, 2001.
4. 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.
5. D. Dams and K. S. Namjoshi. Shape analysis through predicate abstraction and model checking. In *Proc. VMCAI*, 2003.
6. C. Flanagan and S. Qadeer. Predicate abstraction for software verification. In *Proc. POPL*, 2002.
7. 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 of POPL*, pages 1–15, 1996.
8. P. Godefroid, N. Klarlund, and K. Sen. Dart: Directed automated random testing. In *Proc. PLDI*, 2005.
9. D. Gopan, F. DiMaio, N. Dor, T. Reps, and M. Sagiv. Numeric domains with summarized dimensions. In *Proc. TACAS*, 2004.
10. D. Gopan, T. Reps, and M. Sagiv. Numeric analysis of arrays operations. In *Proc. 32nd POPL*, 2005.
11. T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Software verification with blast. In *Proc. of SPIN'03*, volume 2648 of *LNCS*, 2003.
12. G. J. Holzmann and R. Joshi. Model-driven software verification. In *Proc. 11th SPIN Workshop*, volume 2989 of *LNCS*, Barcelona, Spain, 2004.
13. Java PathFinder. +http://javapathfinder.sourceforge.net+.
14. S. Khurshid, C. Pasareanu, and W. Visser. Generalized symbolic execution for model checking and testing. In *Proc. TACAS'03*, Warsaw, Poland, April 2003.
15. S. Khurshid and Y. Suen. Generalizing symbolic execution to library classes. In *Proc. 6th PASTE*, 2005.
16. J. C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, 1976.
17. V. Kuncak and M. Rinard. Existential heap abstraction entailment is undecidable. In *Proc. of SAS*, 2003.
18. R. Manevich, E. Yahav, G. Ramalingam, and M. Sagiv. Predicate abstraction and canonical abstraction for singly-linked lists. In *Proc. VMCAI*, Paris, 2005.

19. C. Pasareanu and W. Visser. Verification of java programs using symbolic execution and invariant generation. In *Proc of SPIN'04*, volume 2989 of *LNCS*, 2004.
20. C. S. Păsăreanu, M. B. Dwyer, and W. Visser. Finding feasible abstract counter-examples. *STTT*, 5(1):34–48, November 2003.
21. C. S. Păsăreanu, R. Pelánek, and W. Visser. Concrete model checking with abstract matching and refinement. In *Proc. CAV'05*, 2005.
22. W. Pugh. The Omega test: A fast and practical integer programming algorithm for dependence analysis. *Commun. ACM*, 31(8), Aug. 1992.
23. S. Sagiv, T. W. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. *TOPLAS*, 2002.
24. K. Sen, D. Marinov, and G. Agha. Cute: A concolic unit testing engine for c. In *Proc. 5th ACM Sigsoft ESEC/FSE*, 2005.
25. W. Visser, K. Havelund, G. Brat, S. J. Park, and F. Lerda. Model checking programs. *Automated Software Engineering Journal*, 10(2), April 2003.
26. T. Xie, D. Marinov, W. Schulte, and D. Notkin. Symstra: A framework for generating object-oriented unit tests using symbolic execution. In *Proc. TACAS 2005*, 2005.
27. T. Yavuz-Kahveci and T. Bultan. Automated verification of concurrent linked lists with counters. In *Proc of SAS*, volume 2477 of *LNCS*, 2002.