Verification of Recursive Methods on Tree-like Data Structures

Jyotirmoy Deshmukh
Dept. of Electrical and Computer Engineering,
University of Texas at Austin.
jyotirmoy@cerc.utexas.edu

E. Allen Emerson
Dept. of Computer Science,
University of Texas at Austin.
emerson@cs.utexas.edu

***Abstract*—Programs that manipulate heap-allocated data structures present a formidable challenge for algorithmic verification. Recursive procedures (methods) in such software libraries are used for a large number of tasks ranging from simple traversals to complex structural transformations. Verification of such methods is undecidable in general. Hence, we present a programming language fragment with a syntax similar to that of `C` for which correctness can be algorithmically checked. For methods written in our fragment, and specifications in the form of tree automata, verification is efficient in most cases, as illustrated by our prototype tool. Our framework can be used to verify methods such as insertion and deletion of nodes in k-ary trees, binary search trees, linked lists, linked list reversal, and rotations in balanced trees, with respect to specifications such as acyclicity, sortedness, list-ness, tree-ness, and absence of null pointer dereferences.**

I. Introduction

Methods that manipulate heap-allocated, linked data structures such as linked lists, queues, binary search trees and balanced trees are important components of software. Exhaustive verification of such methods requires reasoning about a potentially infinite-state system, which is intractable with conventional model checking. Formally, given a method $\mathcal{M}$, and specifications in the form of pre/post-conditions φ and ψ, *correctness* of $\mathcal{M}$ implies that if each input data structure of $\mathcal{M}$ satisfies φ, then the corresponding output data structure satisfies ψ. Though push-button verification of such methods is highly desirable, there are theoretical limits as to what can be achieved. Methods that manipulate common data structures are often written as recursive programs, and checking correctness of arbitrary recursive methods operating on unbounded data structures is an undecidable problem. However, in this paper we show how certain practical recursive methods can be algorithmically verified.

The underlying framework for verification draws on previous work in [1], which presents an automata theoretic framework for verifying iterative methods that manipulate directed graphs. The main steps in such an approach involve: a) specification of data structure invariants and pre/post-conditions, i.e.*properties* of data structures, using tree automata (cf. [1–4]), b) construction of an *exact* abstraction for a method $\mathcal{M}$ that we term as the *method automaton* $\mathcal{A}_{\mathcal{M}}$, and c) emptiness checking for the product of $\mathcal{A}_{\mathcal{M}}$ with the input property automaton and a negation of the output property automaton. Essentially, correctness of $\mathcal{M}$ is reduced to checking emptiness of the product. In this paper, we use a similar automata-theoretic approach.

The key contributions in this paper are as follows: Compared to previous work that dealt with iterative, non-recursive, methods on general graphs, in this paper, we focus on *recursive methods* that manipulate trees. Secondly, we present a syntactic fragment for such recursive methods, and algorithms to compile methods in this fragment to method automata. The syntax of our fragment is similar to that of `C`, and a method in our fragment can be compiled using any standard `C` compiler, after a small amount of syntactic sugaring. Thirdly, we provide mathematical conditions that ensure that every method in our fragment can always be compiled into a meaningful method automaton, and can thus be verified. This is in stark contrast with prior work that assumes an oracle to provide auxiliary proofs about a method's behavior for verification to succeed. For instance, the technique in [1] relies upon a proof that a given method performs only a bounded number of destructive updates to the underlying data structure; in this paper, our syntactic fragment obviates this need. Finally, our techniques can enable verification of practical methods including depth-first insertion and deletion of nodes, search and replace of data in k-ary and binary search trees, linked-list reversal, and tree rotations. Early results with our prototype tool indicate good run-times in verifying useful methods on linked lists and binary trees. Optimizations such as symbolic representations for the automata constructed by our technique would make our approach highly scalable.

In a certain sense, our core methodology can be viewed as an extension of conventional model checking. The most attractive aspect of model checking, i.e., fully automatic verification, *given a set of specifications*, is preserved by our approach. One of the main criticisms of an automata-based approach is that its success hinges on the specification of sufficiently detailed pre/post-conditions. However, a large number of pre/post-conditions are *shape properties*, and are typically generic, i.e., independent of the specific data structure implementation, operating platform, or data values. While we do not focus on specifications in this paper, we argue that by providing such specifications pre-encoded as tree automata (or any suitable logic that can be translated to tree automata), the burden of writing good specifications is lessened. Hence, similar to [1–4], we use tree automata as a specification language. Examples include: acyclicity, sharing, list-ness, sortedness, absence of null pointer dereferences, absence of dangling pointers, and absence of double deletion. The key advantage of our technique is that it does not require annotations such as loop invariants or inductive invariants, and thereby minimizes human input.

The paper is laid out as follows: We introduce the required notation in Sec. II. We discuss the verification framework using automata in Sec. III. In Sec. IV, we provide the syntactic class of recursive methods for which verification is decidable. Experimental results are discussed in Sec. V, and we conclude with related and future work in Sec. VI.

II. Preliminaries

In the programs that we wish to verify, we assume that all program variables belong to only one of two types: a variable over some finite set $\mathcal{D}$ (referred to as a $\mathcal{D}$-variable), and a *pointer variable*. $\mathcal{D}$ is also termed as the data domain. Typical examples of $\mathcal{D}$ include: set of alphanumeric characters, set of integers up to a particular byte-width, set of strings up to a constant length over a finite alphabet, etc. To define a pointer variable, we first introduce the memory model.

Memory Model: A *heap* H is an array of memory cells, where each memory cell contains the value of some program variable. The index of a memory cell is a positive integer known as the *address* of that cell. A *pointer variable* (or simply pointer) p is a variable that evaluates to an address or to 0 (`null`). A *node* n consists of contiguous memory locations that contain the tuple $(d, l_1, \ldots, l_K)$. Here, $d \in \mathcal{D}$, and each l_i is some address. We use $addr(n)$ to denote the address of the first memory location within n. We use $n.d$ (resp. $n.l_i$) to denote the value d (resp. pointer l_i) contained in n. The value $n.d$ is also called the *data value* of n, and $n.l_i$ is also referred to as a *link* of n. We say that a pointer p *points to* n if $p = addr(n)$. If $p = addr(n)$, then the expression $deref(p)$, also called *dereference* of p evaluates to n. The expression $deref(p)$ is undefined and generates an error known as a *null pointer dereference* if the value of p is `null`. We assume that all pointers p are either `null`, or that $deref(p)$ evaluates to a valid node, i.e. we do not allow pointer variables to be explicitly assigned integers, or any form of pointer arithmetic. We assume that the number of links K, is fixed and is part of the definition for a node.

Data Structure: A *data structure* is defined as an arbitrary set of nodes. In this paper, we wish to focus on *rooted* and *connected* tree-like data structures. Such a data structure (say D) has a one-to-one correspondence with a rooted, labeled, directed tree[1] $T(V, E, \mathcal{L}_V, \mathcal{L}_E)$. Here, V is the set of vertices, E is the set of edges, $\mathcal{L}_V : V \to \mathcal{D}$ is a function that maps each vertex to some data value in $\mathcal{D}$, and $\mathcal{L}_E : E \to \{1, \ldots, K\}$ is a function that maps each out-going edge of a given node to a unique positive integer. Each node n in D corresponds to a vertex v_n, such that $\mathcal{L}_V(v_n) = n.d$, and for every n', s.t. $n.l_i = addr(n')$, the edge $e : (v_n, v_{n'})$ belongs to E, and $\mathcal{L}_E(e) = i$. We call every v' s.t. $(v, v') \in E$, a *successor* of v, and analogously call v a predecessor of v'. We assume that there is a designated unique *root* vertex $r \in V$ (corresponding to the root node of the data structure) with the property that r has no predecessor.

[1]A tree is defined in standard fashion as a rooted directed graph in which each node except the root has exactly one predecessor, and the root node has no predecessor.

Methods: A *method* $\mathcal{M}$ is a procedure with an associated signature $sig(\mathcal{M})$ that defines its inputs and return values. For simplicity, we assume that the methods have a *void* return value, i.e. methods do not return anything. Thus, $sig(\mathcal{M})$ effectively defines inputs to $\mathcal{M}$, that are: a) a finite list of pointers $p_1, \ldots, p_n$, and b) a finite list of values $x_1, \ldots, x_m \in \mathcal{D}$. The body of $\mathcal{M}$ may define a finite number of *local* pointer variables and $\mathcal{D}$-variables. $\mathcal{M}$ may additionally access any number of *global* pointer variables and $\mathcal{D}$-variables. A *recursive method* is a method that calls itself. In this paper, we focus on recursive methods that use depth-first traversal to manipulate tree-like data structures. Such a $\mathcal{M}$ satisfies the property that: it is invoked with a pointer to the root r of the tree T as an argument, and for every node n in T, before returning back to the predecessor of n, $\mathcal{M}$ is invoked on all nodes within the sub-tree rooted at n. $\mathcal{M}$ terminates once it returns from recursive invocations on all successors of r.

Def. 1 (Correctness of $\mathcal{M}$)**.** For every input tree T_i that satisfies the pre-condition φ for $\mathcal{M}$, the tree T_o resulting from the action of $\mathcal{M}$ satisfies ψ.

A procedure that can decide the truth or falsehood of the above statement can decide the correctness of $\mathcal{M}$. Unfortunately, this is an undecidable problem. However, in this paper, we argue that by restricting the programming language we can obtain efficiently decidable fragments.

Tree Automata Basics: A K-ary tree has the property that the maximum out-degree of any node in the tree is K. A tree automaton $\mathcal{A}$ running over an infinite K-ary tree T (for some fixed K) is a tuple $\mathcal{A} = (\Sigma, Q, \delta, q_0, \Phi)$ where Σ is a finite, nonempty input alphabet labeling the nodes of T, Q is a finite, nonempty set of states, $\delta : Q \times \Sigma \to 2^{Q^K}$ is the nondeterministic transition function, $q_0 \in Q$ is the initial state, and Φ describes a specific acceptance condition [5]. The run ρ of $\mathcal{A}$ on a Σ-labeled T is an annotation of T with the states Q compatible with δ. The acceptance of T by $\mathcal{A}$ is defined as a specific property of ρ, as described by Φ. For instance, the Büchi condition specifies a set of states (say Q_a) and requires that some state in Q_a appear along every path in ρ infinitely often. The language of $\mathcal{A}$ (denoted $\mathcal{L}(\mathcal{A})$ is the set of all trees accepted by $\mathcal{A}$. Given two automata $\mathcal{A}_1 = (\Sigma_1, Q_1, \delta_2, q_{01}, \Phi_1)$ and $\mathcal{A}_2 = (\Sigma_2, Q_2, \delta_2, q_{02}, \Phi_2)$, the synchronous product $A_p = \mathcal{A}_1 \otimes \mathcal{A}_2$ is defined if $\Sigma_1 = \Sigma_2$. The components of A_p are defined in in terms of $\mathcal{A}_1$ and $\mathcal{A}_2$ as follows: $\Sigma_p = \Sigma_1 = \Sigma_2$, $Q_p = Q_1 \times Q_2$,

$q_{0p} = (q_{01}, q_{02})$, $\Phi_p = \Phi_1 \times \Phi_2$. Further, let $q_p = (q_1, q_2)$ be a state of $\mathcal{A}_p$. We define $\delta_p(q_p, \sigma)$ as the Cartesian product of each next-state tuple in $\delta_1(q_1, \sigma)$ and $\delta_2(q_2, \sigma)$.

III. Verification using Automata

In this section, we formally define a *method automaton* $\mathcal{A}_{\mathcal{M}}$ to serve as an exact abstraction for a given method $\mathcal{M}$. We denote the maximum out-degree of any node in a tree T by K (also termed branching-arity). We recursively define a ranking function rk that maps each vertex of a tree T to a unique string from $\{1, \ldots, K\}^*$, as follows: $rk(root)$ = 1 and $rk(n.l_j) = rk(n) \bullet j$, where $\bullet$ denotes string concatenation. Thus, $rk(root.l_2)$ = 12, $rk(root.l_2.l_1)$ = 121, and so on. We define a composite tree T_c as the *superposition* of trees T_i and T_o, and denote the superposition operation as $T_c = T_i \circ T_o$. Essentially, a vertex v_c in T_c is a pair of vertices (v, v'), s.t. $v \in T_i$, $v' \in T_o$, and $rk(v_c) = rk(v) = rk(v')$. For any v in T_i if there is no v' such that $rk(v)$ = $rk(v')$, we let $v' = \bot$ (and similarly for a $v' \in T_o$ corresponding to a missing $v \in T_i$). We define the method automaton $\mathcal{A}_{\mathcal{M}}$ to run on T_c and accept it if the components of T_c, i.e., T_i and T_o are valid input/output trees for $\mathcal{M}$.

Def. 2 (Mimics)**.** We say that $\mathcal{A}_{\mathcal{M}}$ *mimics* a method $\mathcal{M}$ (denoted $\mathcal{A}_{\mathcal{M}} \bowtie \mathcal{M}$), if for all input trees T_i, $\mathcal{A}_{\mathcal{M}}$ accepts (T_i, T_o) if and only if $T_o = \mathcal{M}(T_i)$.

Recall from Sec. I that our verification technique combines such $\mathcal{A}_{\mathcal{M}}$ with the (finite-state) automata for the pre/post-conditions ($\mathcal{A}_{\varphi}$ and $\mathcal{A}_{\neg\psi}$) and checks the product $\mathcal{A}_p$ for non-emptiness. For verification to be solvable, we require that: (a) each of $\mathcal{A}_{\mathcal{M}}$, $\mathcal{A}_{\varphi}$, and $\mathcal{A}_{\neg\psi}$ have a decidable emptiness problem, (b) the product operator $\otimes$ for combining these automata is well-defined, and (c) the resulting product $\mathcal{A}_p$ has decidable non-emptiness. In this paper, we restrict our attention to properties specifiable as finite state tree automata; examples can be found in [1]. $\mathcal{A}_{\varphi}$ and $\mathcal{A}_{\neg\psi}$ thus trivially satisfy the requirement in (a) and (b). However, an arbitrary recursive method $\mathcal{M}$ using the underlying data structure like a tape, is able to read, write, and move on this tape in either direction, and is able to test if a tape location was previously visited. Machines that have these capabilities are, in general, as powerful as Turing machines. However, Turing machines are not useful abstractions for recursive methods, as most interesting problems for them are undecidable. To overcome this problem, in Lem. 1, we show that if a method $\mathcal{M}$ performs only a "bounded amount of work" on its input data structure, then there exists a finite state tree automaton $\mathcal{A}_{\mathcal{M}}$, s.t. $\mathcal{A}_{\mathcal{M}} \bowtie \mathcal{M}$. Such an $\mathcal{A}_{\mathcal{M}}$ is of value, as it clearly satisfies the requirements in (a), (b) and (c). To precisely explain the notion of bounded work, we first define a *destructive update*.

Def. 3. A *destructive update* (du) is a modification to a data structure node n (directly by accessing n or indirectly through a pointer p to n). Let x be some element of $\mathcal{D}$, and let p_i's be pointers, then du has one of the following forms:

- $n.d = x$ or $p\text{->}d = x$ (changing the data value of n),
- $n.l_i = p_j$ or $p\text{->}l_i = p_j$ (changing a link of n),
- $delete(p)$ (marking the node pointed to by p as free),
- $p\text{->}l_j = new(x, p_1, \ldots, p_K)$ (inserting a new node as a successor of $deref(p)$).

Lemma 1. *Let the maximum number of destructive updates performed by $\mathcal{M}$ on any node of the input data structure D_i, before it terminates, be r. If $r \leq c$ for some constant c, then there exists a finite state automaton $\mathcal{A}_{\mathcal{M}}$, such that $\mathcal{A}_{\mathcal{M}} \bowtie \mathcal{M}$.*

Proof: For simplicity, consider the case where $r = 1$. Let T_c = $T_i \circ T_o$. Each vertex n_c in T_c is the pair (n_i, n_o). We can now define $\mathcal{A}_{\mathcal{M}}$ to run on T_c, starting at the root of T_c in state q_0. We can view the single destructive update performed by $\mathcal{M}$ on n_i as a function f_{du} that maps each n_i to some output node n_o. $\mathcal{A}_{\mathcal{M}}$ transitions to a reject state rej if $n_o \neq f_{du}(n_i)$, otherwise it remains in the state q_0 along all successors of n_i. Finally, if $\mathcal{A}_{\mathcal{M}}$ reaches a terminal vertex (i.e., no successors) in T_c, it transitions to acc, and stays in that state. If $\mathcal{A}_{\mathcal{M}}$ accepts T_c (i.e., reaches a terminal node along every path in T_c) then it must be true that each (n_i, n_o) encodes a valid action of $\mathcal{M}$. In other words, if $T_c = T_i \circ T_o$ is accepted by $\mathcal{A}_{\mathcal{M}}$, then $T_o = \mathcal{M}(T_i)$. If $r > 1$, then we can define T_c = $T^0 \circ T^1 \ldots \circ T^r$, where $T^0 = T_i$ and $T^r = T_o$, and $\mathcal{A}_{\mathcal{M}}$ accepts T_c if for each pair (T^j, T^{j+1}), $T^{j+1} = f_{du_j}(T^j)$. By construction, if r is a constant, we can define $\mathcal{A}_{\mathcal{M}}$ that can mimic $\mathcal{M}$ using only a finite input alphabet that is proportional to the arity of the tree K and r, and uses only a finite number of states. Thus, if $\mathcal{M}$ performs only r ($\leq$ some c) destructive updates to each node in T_i, there always exists a finite $\mathcal{A}_{\mathcal{M}}$ such that $\mathcal{A}_{\mathcal{M}} \bowtie \mathcal{M}$. ■

We refer to Lem. 1 as the *bounded updates property*. Unfortunately, we can show that it is impossible to determine whether an arbitrary recursive method satisfies the bounded updates property. The proof is based on a reduction from Rice's theorem [6], and we skip it for brevity. While Lem. 1 shows that for any $\mathcal{M}$ that has the bounded updates property, there is *some* $\mathcal{A}_{\mathcal{M}}$ s.t. $\mathcal{A}_{\mathcal{M}} \bowtie \mathcal{M}$, it does not provide a recipe for extracting $\mathcal{A}_{\mathcal{M}}$ from $\mathcal{M}$. Also, trying to compile an $\mathcal{A}_{\mathcal{M}}$ from an arbitrary $\mathcal{M}$ is futile, due to the undecidability barrier. In this paper, we identify a syntactic class of methods obtained by restricting the recursive calling patterns and the pointer usage by these methods, such that for any method $\mathcal{M}$ belonging to this fragment, we can automatically obtain a finite state tree automaton $\mathcal{A}_{\mathcal{M}}$. Before we inspect this class, we introduce window-based abstraction, which helps in defining the input alphabet for method automata.

Window-based Abstraction: A window w is a finite encoding of a node n and a bounded number of nodes that succeed n, similar to the finite encodings developed in [1, 2]. A window is obtained by mapping arbitrary integer addresses to a small, finite subset of integers. For the window-based abstraction to be applicable, we need the following assumptions: 1) $\mathcal{M}$ does not access global pointer variables, and, 2) $sig(\mathcal{M})$ has exactly one pointer argument denoted by $\mathtt{I}$ (short for iterator), and a finite number of $\mathcal{D}$-valued arguments. Let $v_{\mathtt{I}}$ denote the node pointed to by I, i.e., $v_{\mathtt{I}} = deref(\mathtt{I})$. In a given invocation of $\mathcal{M}$, since I is the only pointer passed as an input to $\mathcal{M}$, other nodes have to be accessed by following the links of $v_{\mathtt{I}}$. In any reasonable syntax, this limits $\mathcal{M}$ to accessing a small set of nodes. Borrowing notation from C, we use $\mathtt{I}\text{->}l_i$ to denote $v_{\mathtt{I}}.l_i$. Similarly, $\mathtt{I}\text{->}l_i\text{->}l_j$ denotes $deref(v_{\mathtt{I}}.l_i).l_j$. Each expression of the form $\mathtt{I}\text{->}l_i\text{->}l_j\ldots$ is called a *pointer expression*. Any method body contains a finite number of such pointer expressions, which can be statically identified. Let $PE_{\mathcal{M}}$ denote this set.

Def. 4 (Window)**.** A *window* $w(v_{\mathtt{I}})$ is a tuple = $\{v_{\mathtt{I}}, u_1, \ldots, u_n\}$, where each u_i is obtained by dereferencing an element of $PE_{\mathcal{M}}$. I.e., each u_i has the property that if $u_i = deref(p)$ for some $p \in PE_{\mathcal{M}}$, then $(v_{\mathtt{I}}, u)$ is an element of one of : E, E^2, ..., E^z, for a finite z.

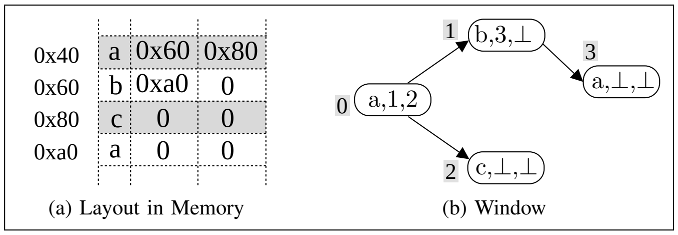


Fig. 1: Window-based Abstraction

Informally, each u_i is a node that is some descendant of $v_{\mathtt{I}}$. Given a set of concrete nodes, we define a function $laddr$ that maps each u_i to i, and $v_{\mathtt{I}}$ to 0. The local address function $laddr$ thus replaces each actual address with an "abstract address". For a node in the concrete data structure that has a successor that lies outside the window, the corresponding link is marked with a '$*$', and null pointers are marked with '$\perp$'. Fig. 1a shows the layout of four nodes of a binary tree in the memory. A window of size 4 representing this heap structure is shown in Fig. 1b, where the numbers above each node indicate the $laddr$ of that node.

By encoding an input tree T using windows, we can obtain an equivalent tree T_w, s.t., every node $v_{\mathtt{I}}$ in T maps to the vertex $w(v_{\mathtt{I}})$ in T_w. In other words, every vertex of T_w is a window; T is embedded within T_w, and can be extracted from T_w by eliding all nodes, except the first, from each vertex in T_w. We note that an arbitrary tree T'_w in which each vertex is a window may not correspond to a valid tree. Since windows encode sets of nodes in T, adjacent windows in T'_w may contain different values for the same nodes in T. We say that T_w is a *consistent tree*, if the overlapping portions of adjacent windows in T_w are identical (except for the values marked with $*$), and for the window corresponding to a terminal node $v_{\mathtt{I}}$, all other nodes in the window are marked as null ($\perp$).

Example 1. Consider a node n of the form: $n = (d, l_1)$. Fig. 2 illustrates a list T_w in which every vertex is a window of size 2, and T_w is consistent with T.

Modeling destructive updates: A destructive update du to $v_{\mathtt{I}}$,

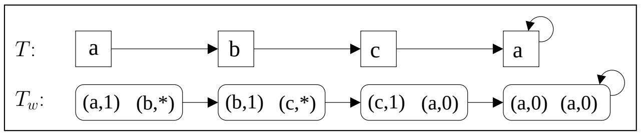


Fig. 2: Consistent Abstraction

either marks $v_{\mathtt{I}}$ as deleted, or changes $\mathtt{I}\text{->}d$ or $\mathtt{I}\text{->}l_i$ (for some i). We can view du as a function mapping an "input" window w_i to an "output" window w_o, where w_o is obtained by performing the actions of du on w_i. Thus for statement: $\mathtt{I}\text{->}d = x$; w_o is identical to w_i except for $\mathtt{I}\text{->}d$, which has the value x. When du modifies $\mathtt{I}\text{->}l_i$, the expression on the RHS of du is a pointer expression, or a new node. The effect of statement $\mathtt{I}\text{->}l_i := p$, is to set $\mathtt{I}\text{->}l_i$ in w_o to $laddr(p)$. Insertion of a new node is modeled by adding a new node to w_o and setting $\mathtt{I}\text{->}l_i$ to the $laddr$ of this new node. Deleting a node is modeled by over-writing each field in the corresponding node in w_o with some special character (say '$-$'). Thus, for any destructive update du, we can compute a function f_{du} that maps a given w_i to some w_o.

Example 2. Consider a node with the same layout as in Ex. 1. Let $w(v_{\mathtt{I}}) = \{(a, 1), (b, *)\}$, where $v_{\mathtt{I}} = (a, 1)$. Now consider the action of the destructive update du: $\mathtt{I}\text{->}d := \mathtt{I}\text{->}l_1\text{->}d$. The resulting window $w'(v_{\mathtt{I}})$ is $\{(b, 1), (b, *)\}$. In this case, f_{du} is a function mapping any window w to a window w' in which both nodes in w' have the same data value as the data value of the second node in w.

IV. Syntactic Class for Recursive Methods

In this section, we present a syntactic class for depth-first recursive methods on tree-like data structures for which verification is efficiently decidable. We split this class into the class of tail-recursive methods, and the more general class allowing non-tail recursion. Tail recursive methods can be converted into iterative methods, and one could argue that these can then be verified using techniques developed in [1]. However, the purpose of discussing tail-recursive methods is twofold: a) it displays how the syntax that we present ensures the bounded updates property, which contrasts with [1], where the bounded updates property had to be ensured by an oracle, and b) it makes the exposition on the more general class easier by introducing necessary terminology in the context of an easier problem.

We say that a method $\mathcal{M}$ visits a node $v_{\mathtt{I}}$ when it is invoked with I as its pointer argument. The control-flow structure of a general recursive method $\mathcal{M}$ is as follows: $\mathcal{M}$ first visits the *root* node. Any other node $v_{\mathtt{I}}$ is visited following a recursive invocation from $v_{\mathtt{I}}$'s parent. $\mathcal{M}$ re-visits $v_{\mathtt{I}}$ interleaved with recursive invocations with successors of $v_{\mathtt{I}}$ as arguments, before it finally *returns* back to the parent of $v_{\mathtt{I}}$ from which it was invoked. In a *tail recursive* method all recursive calls are the final operations in the method body, before it returns.

Tail-Recursive Methods: We formally present the syntax for tail-recursive methods in Fig. 3. Each $\mathcal{M}$ has a signature

sig and a body. $\mathcal{M}$'s body is sub-divided into base case blocks ($baseblks$), and a recursive block. The semantics are as follows: $\mathcal{M}$ first evaluates $bcond_j$ over the nodes in $w(v_{\mathtt{I}})$, and if *true*, performs the actions within $bblock_j$ followed by a `return`. If none of the $bcond_j$ evaluate to *true*, then $\mathcal{M}$ performs the destructive updates specified by $rblock_0$ over the nodes within $w(v_{\mathtt{I}})$, followed by recursive visits to successors of $v_{\mathtt{I}}$ for which $rcond_j$ evaluates to *true* over the possibly updated $w(v_{\mathtt{I}})$. Block statements ($block$) are recursively defined to consist of conditional statements ($ifstmt$), local assignments ($local$), destructive update statements (du), or empty statements ($skip$). We omit the syntax for assignments to local variables ($local$), and the syntax for du statements is specified in Def. 3. The expressions exp, $bcond_j$ and $rcond_j$ are Boolean-valued equality or disequality expressions comparing: a) two pointers, b) a pointer with the null value ($\perp$), or c) data variables. We assume that any calls to methods other than $\mathcal{M}$ are inlined within the body of $\mathcal{M}$. In contrast to standard programming languages, we specifically disallow loops and pointer arithmetic.

$\mathcal{M}$	::=	sig { $baseblks$ $rblock_0$ $calls$ `return` }
sig	::=	(`I`) | (`I`, $x_1, \ldots, x_n$)
$baseblks$	::=	$baseblk$ | $baseblks$ $baseblk$
$baseblk$	::=	`if` ($bcond_j$) { $bblock_j$ `return`}
$\forall j : bblock_j$	::=	$block$
$rblock_0$	::=	$block$
$calls$	::=	$call$ | $calls$ $call$
$call$	::=	`if` ($rcond_j$) $\mathcal{M}$(`I->`l_j)[a]
$block$	::=	$stmt$ | $block$ $stmt$
$stmt$	::=	$ifstmt$ | du | `skip` | $local$
$ifstmt$	::=	`if` (exp) { $block$ } `else` { $block$ }

[a]Note: We statically enforce that for any two invocations of the form $\mathcal{M}(\mathtt{I->}l_j)$ and $\mathcal{M}(\mathtt{I->}l_k)$, $\mathtt{I->}l_j \neq \mathtt{I->}l_k$, failing which is a syntax error.

Fig. 3: Syntax for Tail-Recursive methods

Lemma 2. *Methods that respect the syntax specified in Fig. 3 satisfy the bounded updates property.*

The correctness of Lem. 2 follows from the observation that our syntax ensures that $\mathcal{M}$ recursively visits each successor of any node $v_{\mathtt{I}}$ at most once[2] - when it is recursively invoked from the parent of $v_{\mathtt{I}}$. Moreover, as no destructive update is performed when $\mathcal{M}$ returns, effectively, $\mathcal{M}$ destructively updates any $v_{\mathtt{I}}$ at most once. From Lem. 1, we know that there exists some finite state automaton $\mathcal{A}_{\mathcal{M}}$ such that $\mathcal{A}_{\mathcal{M}} \bowtie \mathcal{M}$. We show how we can derive the desired $\mathcal{A}_{\mathcal{M}}$ in Algo. 1.

Recall that $\mathcal{A}_{\mathcal{M}}$ is defined to run on a composite tree T_c, where the set of nodes of T_c (i.e. the input alphabet for $\mathcal{A}_{\mathcal{M}}$) is a set of pairs of windows, i.e., $\Sigma = \{\sigma | \sigma = (w_{in}, w_{out})\}$. The required size and form of each of the windows can be statically computed in a single pass over $\mathcal{M}$ by inspecting the set $PE_{\mathcal{M}}$ (pointer expressions within $\mathcal{M}$). Note that our syntax

Algorithm 1: CompileTailRecursive

```
1  begin
2      $W$:= all windows (computed by inspecting $PE_{\mathcal{M}}$),
       $\Sigma$:=$W \times W$, $q_0$:=$init$, $Q$:=$\{init, acc, rej\} \cup Q_\Sigma$,
       $\Phi$:=$\{acc\}$, $H$ :={}
       /* Note that $\sigma$ = $(w_{in}, w_{out})$ */
3      foreach $\sigma$ in $\Sigma$, $q$ in $Q_\Sigma$ do
           /* Reject inconsistent $(q,\sigma)$ */
4          $H$ :=$H \cup \{(\sigma, init)\}$
5          if (consistent($\sigma$,$q$)) then $H$ :=$H \cup (\sigma, q)$
6          else reject($q$,$\sigma$)
7      $RS$:=$\Sigma$
8      foreach $baseblk$ in $baseblks$, $\sigma$ in $\Sigma$ do
9          if ($\sigma \models bcond_j$) then
10             $RS$ :=$RS \setminus \{\sigma\}$
11             $f_{bblock_j}$ := computeBlock($bblock_j$)
12             if ($w_{out}$ == $f_{bblock_j}(w_{in})$) then
                   /* $\sigma$ mimics $bblock_j$ */
13                 foreach $q$ in $H(\sigma)$ do accept($q$,$\sigma$)
14             else
                   /* $\sigma$ does not mimic $bblock_j$ */
15                 foreach $q$ in $H(\sigma)$ do reject($q$,$\sigma$)
16     foreach $\sigma$ in $RS$ do
17         $f_{rblock_0}$ :=computeBlock($rblock_0$)
18         if ($w_{out}$ == $f_{rblock_0}(w_{in})$) then
               /* $\sigma$ mimics $rblock_0$ */
19             foreach $q$ in $H(\sigma)$ do
20                 foreach $j$ in $\{1, \ldots, K\}$ do
21                     if ($\sigma \models rcond_j$) then
22                         $next[j]$:= computeState($\sigma$)
23                     else
24                         $next[j]$:=$acc$
25                 $\delta$:= $\delta \cup \{(q, \sigma, next)\}$
26         else
               /* $\sigma$ does not mimic $rblock_0$ */
27             foreach $q$ in $H(\sigma)$ do reject($q$,$\sigma$)
28 end
```

guarantees that the size and number of possible windows is finite, and bounded by the largest expression in $PE_{\mathcal{M}}$; thus, the size of Σ (which is simply the set of all pairs of windows) is finite. The set of states $Q = \{acc, rej, init\}$ have their usual meanings as an *accept* state, a *reject* state, and an *initial* state respectively. The states in Q_Σ are used to check if $w(v_{\mathtt{I}})$ is consistent with the window(s) read at predecessors of $v_{\mathtt{I}}$. If $|w(v_{\mathtt{I}})| = 1$, then Q_Σ is empty, and then Lines 2-6 are skipped. Otherwise, we reject those state/symbol pairs that correspond to an inconsistent annotation at neighboring nodes in T_c. As Σ is finite, the set of states Q is also finite. If a state/symbol pair is consistent, we add it to the map H in Line 5.

We then identify those symbols σ that correspond to $\mathcal{M}$ entering any of the base cases. For the $bblock_j$ block of statements within the j^{th} base case, we compute f_{bblock_j} that is the composition of the functions for the statements within $bblock_j$. If the update encoded by σ is faithful to f_{bblock_j}, i.e. $w_{out} = f_{bblock_j}(w_{in})$, we accept all consistent states for

[2]Of special note is the case where $\mathcal{M}$ visits the l_1-successor of $v_{\mathtt{I}}$, followed by updating $v_{\mathtt{I}}$ so that $v_{\mathtt{I}}.l_2 = v_{\mathtt{I}}.l_1$, followed by visiting $v_{\mathtt{I}}.l_2$ (which is now the same as $v_{\mathtt{I}}.l_1$). Such a scenario is detected at compile-time and disallowed, as specified by the footnote in Fig. 3.

this symbol (Line 13), else we reject them (Line 15). Once all $baseblk$ statements are processed, the remaining symbols (in the set RS) correspond to symbols for which $\mathcal{M}$ enters the recursive case. For each symbol $\sigma \in RS$, we check if the update encoded by σ is faithful to f_{rblock_0}. If not, we reject all consistent states for that σ (Line 27). If yes, we identify the successors of $v_{\mathtt{I}}$ within $w_{out}(v_{\mathtt{I}})$ that $\mathcal{M}$ would visit (by virtue of $rcond_j$ evaluating to $true$), and transition to the appropriate state (in Q_Σ) for those successors (Line 22). The remaining successors are not visited by $\mathcal{M}$, and hence, we simply transition to acc for these (Line 24). We use `reject` (q,σ) as a macro that adds the transition: $(q, \sigma, (rej, \ldots, rej))$ to δ (similarly `accept`). The function `computeBlock` composes the functions f_s for individual statements s within a block statement. `getState` returns the state in Q_Σ that encodes the value of the current pair of windows (σ). `consistent` checks if a given $q \in Q_\Sigma$ that encodes the values in some preceding window, is consistent with the window being currently processed. As a final step (not shown in the algorithm), we add self-loops to the acc and rej states, making them "trap" states. Note that by construction, our algorithm guarantees that Lem. 3 is true. The description presented above gives a proof sketch; we omit the details for brevity.

Lemma 3. *$\mathcal{A}_{\mathcal{M}}$ derived using Algo. 1 has the following properties: a) $\mathcal{A}_{\mathcal{M}}$ is a finite tree automaton, i.e., a finite Q and Σ, b) if $\mathcal{A}_{\mathcal{M}}$ accepts a composite tree $T_c = T_i \circ T_o$, then $T_o = \mathcal{M}(T_i)$, i.e., $\mathcal{A}_{\mathcal{M}} \bowtie \mathcal{M}$.*

General Recursive Methods: In contrast to tail-recursive methods, a non-tail-recursive method can re-visit a node between recursive calls to its successors, and perform destructive updates. We make the same assumptions as for tail-recursive methods that: a) methods do not use global pointers, b) methods use a single pointer argument during recursive invocation, and c) at any node, for any given successor s, $\mathcal{M}$ is invoked with s as an argument at most once. The syntax for the general class of recursive methods is presented in Fig. 4. Instead of a single destructive update block $rblock_0$ as in Fig. 3, we allow up to K $cblock_j$ statements, where each $cblock_j$ statement consists of a recursive call to the j^{th} successor of $v_{\mathtt{I}}$, followed by a destructive update block $rblock_j$. We only show the differing parts in this description, as all other definitions remain the same (except *calls*/*call*, which is no longer relevant).

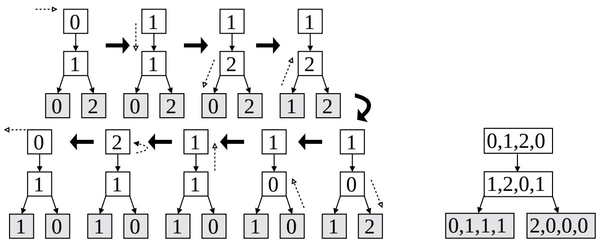
$\mathcal{M}$	::=	*sig* { *baseblks* $rblock_0$ *cblocks* `return` }
cblocks	::=	$cblock_j$ | *cblocks* $cblock_j$
$cblock_j$	::=	$call_j$ $rblock_j$
$rblock_j$	::=	*block*

Fig. 4: Syntactic Class for Recursive Methods

Lemma 4. *Methods with syntax as specified by Fig. 4 satisfy the bounded updates property.*

Proof: A recursive method $\mathcal{M}$ satisfying the above assumptions visits a node with K out-going edges and performs destructive updates: a) the first time when called from the parent node of n ($rblock_0$), and b) K times after each recursive call returns ($rblock_{1..K}$), and c) never after the return. Thus, the total number of destructive updates to any node is at most $K+1$. However, for a given tree, K is a constant, and thus the total number of destructive updates is bounded by a constant. Thus, such an $\mathcal{M}$ satisfies the bounded updates property. ■

The overall scheme for compilation into $\mathcal{A}_{\mathcal{M}}$ for the class in Fig. 3 is similar to the one used for compiling tail-recursive methods. In Algo. 1, all destructive updates by a tail-recursive method on a given window can be composed into a single destructive update described by the function f_{rblock_0} using `computeBlock`. This is not possible for a method belonging to Fig. 4, since it performs $K+1$ distinct blocks of updates. However, by altering the way we define the composite tree, we can use an algorithm very similar to Algo. 1 to compile $\mathcal{M}$ into $\mathcal{A}_{\mathcal{M}}$. We first observe that if we record the actions of such a $\mathcal{M}$ during a depth-first traversal at each node in the underlying data structure D, we obtain an annotated D', where every node of D' is a $K+2$-tuple of values. We illustrate this with an example in Ex. 3.



(a) Depth-first Traversal (b) Annotated Tree

```
method changeData (I) {
    if ((I->$l_1$ == $\bot$)&&(I->$l_2$ == $\bot$)) {
        incMod3(I->$d$);
        return;
    }
    incMod3 (I->$d$);
    if (I->$l_1$ $\neq$ $\bot$) { changeData (I->$l_1$); }
    incMod3 (I->$d$);
    if (I->$l_2$ $\neq$ $\bot$) { changeData (I->$l_2$); }
    incMod3 (I->$d$);
    return;
}
```

(c) Method changeData

Fig. 5: Depth-first Traversal by `ChangeData`

Example 3. Fig. 5c shows the recursive method `changeData`, that changes the data value of each node in the input tree. We assume that $\mathcal{D}$ is the set $\{0, 1, 2\}$, and use `incMod3` as a macro to replace the three conditional assignments that specify $modulo$-3 increment. Fig. 5a shows the actions of `changeData` on an input tree, while Fig. 5b shows the input tree annotated with the actions of `changeData`.

The intuition for Algo. 2 is that depth-first traversal by a method $\mathcal{M}$ generates an annotation similar to that in Ex. 3 on the underlying tree. We define the composite tree T_c s.t., each node σ in T_c is a $(K+2)$-tuple of the form

Algorithm 2: CompileGeneralRecursive

```
1  begin
       /* Note that
           $\sigma = (w_{in} = w_0, w_1, \ldots, w_K, w_{out} = w_{K+1})$ */
2      Lines 2-15 of Algo. 1
3      foreach $\sigma$ in $RS$ do
4          $rejected{:=}false$
5          foreach $j$ in $\{0, \ldots, K\}$ do
6              $f_{rblock_j}$:= computeBlock($rblock_j$)
7              if $(w_{j+1} \neq f_{rblock_j}(w_j))$ then
8                  foreach $q$ in $H(\sigma)$ do reject($q, \sigma$)
9                  $rejected := true$
10                 break
11         if $(\neg rejected)$ then Lines 19-25 of Algo. 1
12 end
```

$(w_0, w_1, \ldots, w_K, w_{K+1})$. For a composite tree T_c being inspected by $\mathcal{A}_\mathcal{M}$, $w_{in} = w_0$, $w_{out} = w_{K+1}$, and each w_{j+1} is the conjectured value for the effect of $rblock_j$. Our syntax ensures that a recursive call to the children of w_j does not affect w_j. Hence, we can check if for the function f_{rblock_j} corresponding to each $rblock_j$ statement, whether $w_{j+1} = f_{rblock_j}(w_j)$, for all j s.t. $0 \leq j \leq K$ in a single swoop. If any of these conditions is false, we reject the entire symbol σ (Line 8), else we proceed in the same way as in Algo. 1. Due to space limitations, Algo. 2 only shows the parts that differ from Algo. 1. The definitions of `consistent`, `getState` and `computeBlock` are changed to account for the changes in σ.

To make a correctness argument, we introduce some notation. Let T_c be a tree in which every node is a $K + 2$-tuple of windows. Let T_{i_w} be the tree obtained by eliding all but the first window in every node of T_c, and T_{o_w} be obtained by eliding all but the last window in every node of T_c. Suppose T_{i_w} and T_{o_w} are consistent trees (recall the definition from Sec. III), then let T_i (resp. T_o) be the corresponding tree for which T_{i_w} (resp. T_{o_w}) is the window-based abstraction. By construction, Algo. 2 guarantees Theorem 1; we skip the proof for brevity.

Theorem 1. *$\mathcal{A}_\mathcal{M}$ derived using Algo. 2 has the following properties: a) $\mathcal{A}_\mathcal{M}$ is a finite tree automaton, i.e., a finite Q and Σ, b) $\mathcal{A}_\mathcal{M}$ rejects T_c if T_{i_w} or T_{o_w} are inconsistent, and c) if $\mathcal{A}_\mathcal{M}$ accepts T_c, then $T_o = \mathcal{M}(T_i)$, i.e. $\mathcal{A}_\mathcal{M} \bowtie \mathcal{M}$.*

V. Experimental Results

The complexity of our technique is equal to the complexity of checking emptiness of the product automaton $\mathcal{A}_p$. For acceptance conditions such as the Büchi condition, this is polynomial in the number of states of $\mathcal{A}_p$, which is itself linear in the size of $\mathcal{A}_\mathcal{M}$ (denoted $|\mathcal{A}_\mathcal{M}|$), $|\mathcal{A}_\varphi|$ and $|\mathcal{A}_{\neg\psi}|$. $|\mathcal{A}_\mathcal{M}|$ is proportional to the size of $\mathcal{M}$, but is dominated by $|\Sigma|$, which in turn is polynomial in $|\mathcal{D}|$ and exponential in the branching-arity (K) of the tree. We note that for purely structural properties, $\mathcal{D}$ can often be abstracted to a single symbol. We have implemented a prototype tool in Java that can verify methods with the syntax specified by Fig. 4. Some of the early results are shown in Table I. The methods we used for testing are commonly found implementations for linked lists and binary trees written in C, adapted to our syntax. As seen in Table I, while the time required to construct $\mathcal{A}_\mathcal{M}$ is a fraction of the total time taken, the memory consumption is a sore spot. This is so because, in our current implementation, we explicitly construct Σ. The size of Σ can be reduced by several orders of magnitude by an abstraction that involves forming clusters of similar symbols in Σ. Furthermore, we can employ symbolic techniques like BDDs or SAT to compactly represent Σ, which would ameliorate the memory consumption. Lastly, as predicted by the complexity analysis, we observe that the run-time and memory consumption increases sharply with K.

Method	Spec.	Time[a] (secs) $\mathcal{A}_\mathcal{M}$	Time[a] (secs) Total	Mem. (MB)
On Linked Lists:				
`DeleteNode`	Acyclic	0.3	1.3	20
`InsertAtTail`	Acyclic	0.01	0.8	<1
`InsertNode`	Acyclic	0.4	1.6	48
On Binary Trees:				
`InsertNode`	Acyclic	15	329	2512
`ReplaceAll(a,b)`	Acyclic	5	26	324
	$\nexists$`I` : `I->`$d = a$	5	27	432
`DeleteLeaf`	Acyclic	12	48	630

[a]Experiments were performed on an Athlon 64X2 4200$^+$ system with 6GB RAM.

TABLE I: Experimental Results

Counterexample Generation: If $\mathcal{A}_p$ is found to be non-empty, the tree T_c witnessing its non-emptiness can be extracted from the transition diagram δ_p of $\mathcal{A}_p$ using standard techniques. By projecting T_c onto its components, we can obtain trees T_i and T_o respectively. T_i represents a valid input tree to $\mathcal{M}$ for which the "bad" output T_o is generated, i.e., a concrete counterexample to the correctness of $\mathcal{M}$.

VI. Related Work and Conclusions

Shape Analysis: Shape analysis [7–9], focuses on computing (3-valued) structure descriptors at each program point, typically using static analysis. Shape analysis can be used to analyze a broad class of methods, but to our best knowledge provides approximate results in double exponential time. Predicate abstraction has been used for shape analysis in [10–12]. [10, 12] focus on singly linked lists, and [13] extends the authors' previous work to programs with single-parent heaps. While [14] provides a way to combine predicate abstraction and model checking, it may require hints to converge to a solution.

Separation Logic: Separation logic, typically allows deductive verification for heap modifying programs [15, 16], and has been traditionally used for manual proofs or in conjunction with a theorem prover. Recent work has focussed on automation, by deriving decidable fragments for programs operating on structures with single successors [17].

Automata-based approaches: This paper significantly extends prior work in [1], which uses tree automata for verifying iterative methods. In [2, 3], system configurations are trees, succinctly encoded as tree automata. The transition relation is a bottom-up tree transducer τ, and the technique checks if τ^* applied to the initial configuration automaton reaches a bad state. Though this is undecidable, the authors use abstraction-refinement to obtain a conservative solution. [4] uses tree automata with size constraints to verify balanced trees implementations. PALE [18] encodes programs and partial specifications in MSO logic, which has a non-elementary decision procedure.

Logic-based Approaches: [19] describes a logic of reachable patterns that is undecidable, which when restricted to certain reachability patterns, yields a decidable fragment that can be checked in double exponential time. The restrictions imposed to obtain decidability are incomparable to the work in this paper. While the work in [20] inspires some of the later work on decidable fragments (cf. [19]), the paper itself does not yield a practical algorithm. Bottom-up shape analysis [21] for heap-manipulating programs computes Hoare triples as summaries for a given method. It may be possible to combine our technique with bottom-up analysis by substituting method fragments that do not respect our imposed syntax rules with equivalent summaries, thereby allowing us to model a larger class of methods.

Due to space restrictions, we omit a few simple extensions that our technique can handle such as: allowing methods to return values in a restricted form (both $\mathcal{D}$-values and pointer values), allowing methods to have a limited access to predecessor nodes up to a bounded distance, and allowing more than one pointer argument in the method signature, with the restriction that all arguments are contained within a window. A more significant extension is verification of recursive methods on directed acyclic graphs (*dags*). Since *dags* can contain sharing between nodes, the restriction of "one visit per successor" is not enough to ensure the bounded updates property. However, if we can enforce (or guarantee) that a method visits each node in a *dag* (and thus every sub-*dag*) at most once, then such methods would satisfy the bounded updates property, and could be verified using a modified form of the algorithms presented here.

Acknowledgements: We thank Prateek Gupta for interesting discussions in the initial phase of this work, and the anonymous reviewers for their helpful comments and suggestions.

References

[1] J. Deshmukh, E. Emerson, and P. Gupta, "Automatic verification of parameterized data structures," in *Proc. of TACAS*, 2006, pp. 27–41.

[2] A. Bouajjani, P. Habermehl, P. Moro, and T. Vojnar, "Verifying programs with dynamic 1-selector-linked structures in regular model checking," in *Proc. of TACAS*, 2005, pp. 13–29.

[3] A. Bouajjani, P. Habermehl, and A. Rogalewicz, "Abstract regular tree model checking of complex dynamic data structures," in *Proc. of SAS*, 2006, pp. 52–70.

[4] P. Habermehl, R. Iosif, and T. Vojnar, "Automata-based verification of programs with tree updates," in *Proc. of TACAS*, 2006, pp. 350–364.

[5] E. A. Emerson and C. S. Jutla, "The complexity of tree automata and logics of programs," *SIAM J. Comput.*, vol. Vol. 29, pp. 132–158, 1999.

[6] M. Sipser, *Introduction to the Theory of Computation*, 1st ed. Course Technology, Dec. 1996.

[7] N. Rinetzky and M. Sagiv, "Interprocedural shape analysis for recursive programs," in *Proc. of CC*, 2001, pp. 133–149.

[8] M. Sagiv, T. Reps, and R. Wilhelm, "Parametric shape analysis via 3-valued logic," *ACM Trans. Program. Lang. Syst.*, vol. Vol. 24, pp. 217–298, 2002.

[9] T. Lev-Ami, N. Immerman, and M. Sagiv, "Abstraction for shape analysis with fast and precise transformers," in *Proc. of CAV*, 2006, pp. 547–561.

[10] I. Balaban, A. Pnueli, and L. D. Zuck, "Shape analysis by predicate abstraction," in *Proc. of VMCAI*, 2005, pp. 164–180.

[11] R. Manevich, E. Yahav, G. Ramalingam, and M. Sagiv, "Predicate abstraction and canonical abstraction for singly-linked lists," in *Proc. of VMCAI*, 2005, pp. 181–198.

[12] J. Bingham and Z. Rakamaric, "A logic and decision procedure for predicate abstraction of heap-manipulating programs," in *Proc. of VMCAI*, 2006, pp. 207–221.

[13] I. Balaban, A. Pnueli, and L. Zuck, "Shape analysis of single-parent heaps," in *Proc. of VMCAI*, 2007, pp. 91–105.

[14] D. Dams and K. Namjoshi, "Shape analysis through predicate abstraction and model checking," in *Proc. of VMCAI*, 2003, pp. 310–323.

[15] J. C. Reynolds, "Separation logic: A logic for shared mutable data structures," in *Proc. of LICS*, 2002, pp. 55–74.

[16] D. Distefano, P. OHearn, and H. Yang, "A local shape analysis based on separation logic," in *Proc. of TACAS*, 2006, pp. 287–302.

[17] J. Berdine, C. Calcagno, and P. O'Hearn, "A decidable fragment of separation logic," in *Proc. of FSTTCS*, 2004, pp. 97–109.

[18] A. Møller and M. I. Schwartzbach, "The pointer assertion logic engine," in *Proc. of Programming Language Design and Implementation*, 2001, pp. 221–231.

[19] G. Yorsh, A. Rabinovich, M. Sagiv, A. Meyer, and A. Bouajjani, "A logic of reachable patterns in linked data-structures," in *Proc. of FOSSACS*, 2006, pp. 94–110.

[20] M. Benedikt, T. Reps, and M. Sagiv, "A decidable logic for describing linked data structures," in *Proc. of ESOP*, 1999, pp. 2–19.

[21] B. Gulavani, S. Chakraborty, G. Ramalingam, and A. Nori, "Bottom-up shape analysis," in *Proc. of SAS*, 2009, pp. 188–204.