

Automatic verification of parameterized data structures ^{*}

Jyotirmoy V. Deshmukh, E.Allen Emerson, and Prateek Gupta

Department of Computer Sciences and Computer Engineering Research Center,
The University of Texas at Austin, Austin TX 78712, USA
{deshmukh,emerson,prateek}@cs.utexas.edu

Abstract. Verifying correctness of programs operating on data structures has become an integral part of software verification. A method is a program that acts on an input data structure (modeled as a graph) and produces an output data structure. The *parameterized correctness* problem for such methods can be defined as follows: Given a method and a property of the input graphs, we wish to verify that for all input graphs, parameterized by their size, the output graphs also satisfy the property. We present an automated approach to verify that a given method *preserves* a given property for a large class of methods. Examples include reversals of linked lists, insertion, deletion and iterative modification of nodes in directed graphs. Our approach draws on machinery from automata theory and temporal logic. For a useful class of data structures and properties, our solution is polynomial in the size of the method and size of the property specification.

Keywords: Parameterized correctness, Data structures.

1 Introduction

Data structures are the basic building blocks for all large software systems. Such systems typically manipulate arbitrarily large data structures using specialized programs known as *methods*. An incorrect implementation of a method can lead to failure of the entire software system. Consequently, reasoning about methods operating on data structures is a significant part of the software verification effort.

We investigate the problem of automatic verification of methods operating on data structures, *parameterized* by their size. Given a method \mathcal{M} , operating on an input data structure modeled as a graph G_i , and a property φ of the graph, we wish to verify that: if φ holds for the input graph G_i , then φ also holds for the graph G_o obtained by the action of \mathcal{M} on G_i , *i.e.*, \mathcal{M} preserves φ . For instance, given a method that adds a node to an acyclic singly linked list, we would like to verify that the output data structure is also a well formed acyclic singly linked list. In contrast to the standard testing approach for validation of

^{*} This research is supported in part by NSF grants CCR-009-8141 & ITR-CCR-020-5483, and SRC Contract No. 2002-TJ-1026

such methods, which ensures correctness for a few candidate data structures up to a bounded size, we would like to verify that methods exhibit correct behavior for arbitrarily large input data structures.

We provide an automatic procedure based on machinery from automata theory and temporal logic to establish parameterized correctness. Our approach is applicable to a broad spectrum of methods that perform updates on dynamically created data structures. For example, our technique can establish correctness for methods such as: reversal of singly linked lists; insertion or deletion of nodes in general graphs (such as linked lists, k -ary trees, directed acyclic graphs, etc.); swapping of nodes within a bounded distance in any general graph; and iterative modification of data values at nodes in any general graph.

In our technique the property to be verified is generally specified using a (tree) automaton running on graphs. Alternatively, we can use temporal logic as the specification language for properties. Thus, we can specify a rich class of properties, including, but not limited to:

1. Connectivity properties such as: reachability of a target node from a source node (where the nodes are specified by pointers); reachability of a given data value from a given node; existence of cycles; existence of sharing (two nodes point to a common node); treeness (each non-root node has a unique parent); list-ness; and checking whether two nodes are fully connected (either node is reachable from the other using the *next* pointer fields).
2. Data-dependent properties such as sortedness (*i.e.*, all nodes in a given graph obey a certain sorting discipline on data values).
3. Properties of dynamically allocated storage such as checking null pointer dereferences and absence of dangling pointers.

We refer to the automaton specifying the property as the *property automaton*, denoted by \mathcal{A}_φ . Similarly, the automaton specifying the negation of the property is specified as $\mathcal{A}_{\neg\varphi}$.

The method (\mathcal{M}) to be verified is algorithmically translated into an automaton. We refer to this automaton as the *method automaton*, denoted by $\mathcal{A}_\mathcal{M}$. $\mathcal{A}_\mathcal{M}$ operates on a pair of input-output graphs; it simulates the action of \mathcal{M} on the input graph checking whether the output matches the output graph. It accepts only those pairs of graphs which represent a valid operation of the method. The pair of input-output graphs is represented using a single *composite graph*.

The central step is to use the method automaton and the property automata to obtain a *composite automaton* (denoted by \mathcal{A}_c) that accepts counterexamples to the correct operation of the method. On a given composite graph, \mathcal{A}_c accepts iff: the property holds for the input, the output conforms to a valid action of the method on the input, and the property fails for the output. Thus if a graph is accepted by \mathcal{A}_c , it represents a witness to the failure of the method. Checking if such a graph exists is equivalent to checking the language accepted by \mathcal{A}_c for nonemptiness.

Formally, we obtain the composite product automaton \mathcal{A}_c , by the product of \mathcal{A}_φ , $\mathcal{A}_\mathcal{M}$ and $\mathcal{A}_{\neg\varphi}$. \mathcal{A}_c accepts a graph G_c representing an input-output pair (G_i, G_o) , iff $G_o = \mathcal{M}(G_i)$ and $\varphi(G_i)$ and $\neg\varphi(G_o)$ are true.

In the above approach, correctness properties are specified by the user, using automata as the specification language. In a variant, but closely related approach, we use a suitable temporal logic in lieu of automata to specify the properties of interest. Temporal logics such as *CTL* often allow an easier specification of properties. The property to be verified is specified as a formula f_φ in the given logic. The method automaton is translated to a formula $f_{\mathcal{M}}$. The parameterized correctness problem reduces to checking the satisfiability of the conjunction: $f_c = f_{\varphi(G_i)} \wedge f_{\mathcal{M}} \wedge f_{\neg\varphi(G_o)}$.

In practice, we provide a simple programming language for describing methods. Our programming language is a useful subset of most modern day high level languages. We can efficiently compile any program written in our language into a method automaton. The time complexity of our technique is polynomial in the size of the method and property automata.

The outline of the paper is as follows: In Section 2, we provide the preliminary background, the problem definition and the scope of our technique. Section 3 discusses the syntax and semantics of our programming language. The algorithm for translation of a method into a method automaton is given in Section 4. We briefly examine the specification of properties as automata in Section 5. We present an application in Section 6 and discuss a variant approach using temporal logic in Section 7. The complexity analysis is discussed in Section 8 and finally, a summary of our paper with related work is given in Section 9.

2 Preliminaries

A data structure can be readily modeled as a directed graph $G(V, E)$ by establishing a one to one correspondence between the nodes of the data structure and the vertices (V) of G , and similarly between links of the data structure and the edges (E) of G . Each node of the data structure is a vertex $v \in V$ in G and a pointer from a source node (\equiv vertex v_i) to a destination node (\equiv vertex v_j) represents a directed edge $(v_i, v_j) \in E$. The data content at each node of the data structure is modeled as a labelling function $L : V \rightarrow \mathcal{D}$, where \mathcal{D} is the domain of data values. For simplicity, we consider graphs with only a bounded *out-degree*, where the out-degree of a graph is defined as the maximum number of outgoing edges from a vertex in the graph.

A method \mathcal{M} is a program that has one or more data structures as input and produces an output data structure which is a mutation of the input. A property φ of a graph $G(V, E)$ is a predicate on the labeled set of vertices and edges of the graph. The property φ is often referred to as the *shape* of a graph. Conversely, a shape φ is identified with a family of graphs for which the property φ is true. For instance, graphs satisfying the property that every non-root vertex has a unique incident edge and the root vertex has no incident edges, are said to constitute the family of trees. Properties of graphs can be conveniently specified as tree automata (See Section 5). We now revisit some important definitions for tree automata operating on trees with out-degree k (*i.e.*, a k -ary tree).

2.1 Tree Automata

A *finite tree automaton* over an infinite k -ary tree is a tuple $\mathcal{A} = (\Sigma, Q, \delta, q_0, \Phi)$ where:

- Σ is the finite, nonempty input alphabet labeling the nodes of the tree,
- Q is the finite, nonempty set of states of the automaton,
- $\delta : Q \times \Sigma \rightarrow 2^{Q \times \dots \times Q(k \text{ times})}$ is the nondeterministic transition function,
- $q_0 \in Q$ is the start state of the automaton, and
- Φ is the acceptance condition.

In our technique, it is convenient to use the parity acceptance condition. The parity acceptance condition $\Phi = (\Phi_0, \Phi_1, \dots, \Phi_m)$ is expressed in terms of sequence of mutually disjoint subsets of Q . If $\pi = q_0, \dots, q_i, \dots$ is a finite or infinite sequence of automaton states q_i , then we say that π satisfies the acceptance condition if the following condition is satisfied: there exists an even number r , $0 < r < m$, such that some state in Φ_r appears infinitely often in π and each of the states in the set $\bigcup_{r < j \leq m} \Phi_j$ appears only finitely often in π . The parity condition is often alternately expressed as follows: A sequence of states π satisfies the parity acceptance condition, when the states of the automaton are colored with a set of colors $\{c_0, \dots, c_m\}$, and the maximal index of the color appearing infinitely often in π is even. For the rest of this paper, we implicitly assume the parity acceptance condition for all tree automata used.

A tree automaton can be meaningfully defined to run on graphs. Essentially a run ρ of a tree automaton on a Σ -labeled input graph is an annotation of the graph with the automaton states compatible with the transition relation of the automaton. Not every automaton has a run on every graph, but if an automaton accepts some tree, it accepts some “small”, finite graph, [EJ '88, Em '85]. Note that when $k = 1$, the tree automaton can be specialized to a string automaton.

2.2 Problem definition

We define a parameterized family of graphs as the set $\mathcal{G} = \{G \mid \varphi(G) \text{ is true}\}$, where the graphs are parameterized by their size. For all input graphs $G \in \mathcal{G}$ and a method \mathcal{M} operating on G , we wish to verify if the resultant graphs $\mathcal{M}(G)$ satisfy the property φ . Formally, we wish to verify the correctness assertion: $\langle \varphi(G_i) \rangle \mathcal{M} \langle \varphi(G_o) \rangle$.

2.3 Scope

Most methods that operate on data structures use a *cursor* or an iterator to traverse the data structure. Methods which have multiple cursors are analogous to multi-head automata. Unfortunately, the parameterized correctness problem for such methods is undecidable, since the nonemptiness problem of a k -head automaton with $k \geq 2$ is undecidable [Rose '65, NSV '04]. Thus, we focus on methods which can be simulated by a single head automaton. Such methods can have multiple cursors, which are constrained to remain within some bounded distance at all times.

Methods can also be characterized by the way they access and mutate the data structure. Some methods perform only a bounded number of *destructive passes* over the data structure. We define a destructive pass as a single traversal of the data structure involving at least one update to some node of the data structure. It is difficult to reason about the parameterized correctness of methods which perform an unbounded number of destructive passes over the data structure, since their operation simulates a linear bounded automaton (LBA). The nonemptiness problem of an LBA is undecidable [HU '79]. Thus, our work focusses on methods which can only perform a bounded number of destructive passes over the data structure.

It is stipulated that the method terminates¹ and performs only a bounded number of destructive passes over the data structure. We also assume that the domain \mathcal{D} of data values is finite.

2.4 Solution framework

In our approach, we use automata to check if a given method \mathcal{M} preserves a property φ . Our technique involves determining the existence of a pair of input-output graphs (G_i, G_o) such that:

1. the input graph G_i satisfies φ or $\varphi(G_i)$ is true (*i.e.*, the input is well formed),
2. the output graph G_o does not satisfy φ or $\neg\varphi(G_o)$ is true and
3. G_o represents a valid action of \mathcal{M} on G_i or $G_o = \mathcal{M}(G_i)$ is true.

Formally, a property φ to be verified is specified as a tree automaton \mathcal{A}_φ , which accepts the set of all graphs which satisfy φ . We are given a similar automaton $\mathcal{A}_{\neg\varphi}$, to accept all graphs that satisfy $\neg\varphi$. The method \mathcal{M} is algorithmically translated into a *method automaton* $\mathcal{A}_\mathcal{M}$, which checks whether $G_o = \mathcal{M}(G_i)$. The input-output graph pair is represented using a *composite graph*, denoted by G_c .

The composite graph $G_c(V, E)$ has each vertex $v \in V$ and each edge $e \in E$ annotated with one of three colors *black*, *green* or *red*. The color *black* represents part of the input graph that remains the same, color *red* represents deleted nodes or edges, and the color *green* represents new nodes or edges. Each vertex of the composite graph is labeled with an ordered pair of labels (d_i, d_o) ($d_i, d_o \in \mathcal{D}$) to model the old and the new data values at the corresponding node in the data structure. The input graph G_i , can be extracted from G_c by considering the subgraph composed of vertices and edges colored *red* or *black* and the labels d_i . Similarly, the output graph G_o can be extracted by considering the set of nodes and edges labeled *black* or *green* and the labels d_o . We define projection operators Γ_i and Γ_o to obtain the graphs G_i and G_o respectively, from the composite graph G_c . The method automaton $\mathcal{A}_\mathcal{M}$ runs on such composite graphs and accepts a composite graph G_c iff $G_i = \Gamma_i(G_c)$ $G_o = \Gamma_o(G_c)$ and $G_o = \mathcal{M}(G_i)$. Similarly,

¹ A similar assumption on program termination can be found in techniques such as shape analysis [Lev-Ami *et al.*], PALE [MS '01], and separation logic [ORY '01], which implicitly assume the termination of the program being analyzed.

the property automata \mathcal{A}_φ and $\mathcal{A}_{\neg\varphi}$ run on composite graphs, and look at the input or output parts of the composite graph.

Remark: Such an annotated graph can be obtained only if the method performs a bounded number of destructive passes over the data structure. For our current discussion, we assume that the method performs a single destructive pass of the data structure, *i.e.*, the method automaton traverses each node of the data structure exactly once. We generalize the assumption to handle multiple, but a bounded number of passes over the data structure, later in Section 4.

Finally, we construct a composite automaton \mathcal{A}_c , which is the synchronous product of $\mathcal{A}_{\neg\varphi}$, \mathcal{A}_M and \mathcal{A}_φ . The product construction for the composite automaton is defined in standard fashion, [Car '94]. The number of states of the composite automaton is proportional to the product of the number of states of the constituent automata. The composite automaton is empty iff the method preserves the property. If the automaton is non-empty, then there exists a graph G which satisfies the property φ , but $\mathcal{M}(G)$ does not satisfy φ . Thus we also obtain a counterexample which illustrates erroneous behavior of the method.

3 Programming language description

In this section, we define the syntax and semantics of our programming language. An atomic unit of a data structure is termed as a *node*. Each node has a data field and a set of k pointer fields $\mathbf{next}_1, \dots, \mathbf{next}_k$. We define **cursor** as a reserved word for an iterator through a given data structure. Since we focus on methods which can be mimicked using a single headed automaton, our programming language supports a single **cursor**². If the node being pointed to by the cursor is n , a bounded window w is defined as a set of all nodes within a fixed distance from n . The size of the window w , denoted by $|w|$, is the cardinality of w . We define **head** or **root** as reserved words to indicate start nodes of the data structure.

We use a C-like syntax for describing methods, and accordingly use the abbreviation **cursor->field** to indicate the corresponding *field* of a node pointed to by the **cursor**. A statement in our programming language can have one of the forms as show in Table 1.

The sequential composition of two more statements with **;** as the composition operator is called a *block statement*. For memory related operations, our language allows deletion of nodes being pointed to by any pointer $ptr(\neq \mathbf{cursor})$, using the **delete** statement.

Every update to a **cursor** is preceded by storing the current value of the **cursor** in a special variable called **prev**, which cannot be used on the left hand side of an assignment statement. The addition of **prev** enhances the expressive power of our language by allowing methods to perform operations based on past value of the **cursor**. Destructive updates are allowed only within the bounded

² We can easily extend our approach to handle a fixed number of *virtual* cursors within a bounded window.

window defined by the `cursor`. When a new node is created, the pointer fields of the new node are initialized to any value within the current window. The initialization of the fields of a new node is made before `cursor` or `cursor->nexti` is updated.

Table 1. Programming language syntax	
Assignment statement	
	<code>cursor->data := data-constant;</code>
	<code>cursor->next_i := ptr^a</code>
	<code>cursor := ptr;</code>
	<code>cursor := new node { data := data-constant;</code>
	<code>next₁ := ptr;...; next_k := ptr;};</code>
	<code>cursor->next_i := new node {...};</code>
Conditional statement	
	<code>if (test-expr) {</code>
	<code>block statement;</code>
	<code>} else { block statement; }</code>
Loop statement	
	<code>while (loop-cond) {</code>
	<code>loop-body;</code>
	<code>update statement; }</code>
	Here, the <i>update statement</i> is of the form:
	<code>cursor := ncursor₁ when cond₁</code>
	<code>ncursor₂ when cond₂</code>
	<code>⋮</code>
	<code>ncursor_k when cond_k;</code>
Break statement	
	<code>break;</code>
Null statement	
	<code>null;</code>
<hr style="width: 20%; margin-left: 0;"/> ^a <i>ptr</i> represents an allowable pointer expression, which can take one of the following forms: <code>cursor->next_{i₁}->next_{i₂}->... next_{i_m}}}</code> or <code>prev</code> or <code>cursor</code> , where <i>m</i> is bounded by the size of the window.	

In a conditional statement, a *test-expr* is a boolean expression which either involves a comparison of the data value of the current node with another data value or a comparison of two *ptr* expressions.

A loop statement consists of three parts, a *loop condition*, the *loop body* and an *update statement*. A loop condition *loop-cond*, is a boolean expression involving the comparison of a *ptr* expression with `null`³. A method continues executing the loop as long as the loop condition is true. The loop body is a sequence of two or more non-loop statements. We do not allow nesting of loops statements, since this can in general mimic a *k*-head automaton. At the beginning of each iteration

³ Note that any special termination condition required can always be modeled with the help of a `break` statement coupled with a conditional statement inside the loop body.

of the loop the values `cursor->nexti` are cached in special variables `ncursori` which cannot be used on the left hand side of an assignment statement.

The `cursor` can be updated inside a loop statement only using an update statement. The value of `cursor` is assigned to `ncursori` if `condi` evaluates to true and `condj $\forall j < i$` evaluates to false, where `cond1 . . . condk` are any boolean valued expressions. A `break` statement breaks out from the `while` loop enclosing the break statement. If the break statement is not inside a loop body, no action is taken.

4 Translation into automata

We can mechanically compile any given method in our language into corresponding parts of the *method automaton*, \mathcal{A}_M . \mathcal{A}_M is a k -ary tree automaton running on graphs. For ease of exposition, we presently assume that each node has a single successor. We assume that all the statements in the method are labeled with a unique line number $\{1, \dots, |\mathcal{M}|\}$, where $|\mathcal{M}|$ is the length of \mathcal{M} .

\mathcal{A}_M is of the form $(\Sigma, Q_M, \delta_M, q_{0_M}, \Phi_M)$, where the notation used is similar to the one described in Section 2.1. The parity acceptance condition Φ_M is specified using two colors $\{(red = c_1), (green = c_2)\}$. States colored *green* are accepting states and those colored *red* are rejecting.

The action of a statement of \mathcal{M} is mimicked by a transition of \mathcal{A}_M . On a given input graph G_i , the moves of the automaton are completely deterministic and a run of \mathcal{A}_M on G_i is unique and well defined. For two different input graphs, the state of \mathcal{A}_M after executing the same statement of \mathcal{M} may be different. We use Q_j to denote the set of all possible states of \mathcal{A}_M (for all input graphs) after executing the statement s_j .

A state q_j of \mathcal{A}_M is modeled as a tuple $(j, cur_d, cur_p, new_d, new_p)$, where j corresponds to the line number of the statement s_j , cur_d is the current data value of the node being pointed to by the `cursor`, cur_p is the value of `cursor->next`, new_p is 0 if no new node is added at the current `cursor` position, else it is a non-zero value indicating the location of `new_node->next`, and new_d contains the data value of the new node that is added. The initial state of the automaton is denoted by $q_{0_M} = (0, 0, 0, 0, 0)$.

Let θ be a boolean valued expression over the set of program variables. We say that a state q satisfies θ , denoted by $q \models \theta$, if the valuation of θ over the components of q is true. (Note that a state q completely encodes the values of all program variables.). We denote by Q^θ the set $\{q | q \models \theta\}$ and $Q^{-\theta}$, the set $\{q | q \not\models \theta\}$.

4.1 Algorithm for translation

We now give the algorithm used to populate the transition relation of \mathcal{A}_M :

1. Let Q be the set of possible states prior to an assignment statement s_j . For every state $q \in Q$, s_j is modeled by adding a transition of the form (q, ϵ, q') , where q' encodes the new data value, the pointer field value or a new node inserted at the

- cursor position by s_j . For instance given a state $q = (k, cur_d, cur_p, new_d, new_p)$, the assignment statement `cursor->data:=val` is modeled by adding the transition (q, ϵ, q') , where $q' = (j, val, cur_p, new_d, new_p)$.
2. Let Q be the set of possible states prior to a conditional statement s_j . Let ϕ be the test expression of s_j . Let s_k (s_m , resp.) be the first statement within the `if` (`else`, resp.) block of the conditional statement. For a conditional statement, we add transitions of the form: $\forall q \in Q^\phi, \forall t \in Q_k : \{(q, \epsilon, t)\}$, and $\forall q' \in Q^{-\phi}, \forall t' \in Q_m : \{(q', \epsilon, t')\}$.
 3. (a) Let Q denote the set of possible states prior to a loop statement s_j . For the loop statement shown in the left hand column of the table below, we add transitions shown in the right hand column.

$j: \text{while } (\psi) \{$ $k: \quad s_k;$ $\quad \vdots$ $l: \quad \text{update statement};$ $m: s_m;$	$\forall q \in Q^\psi, \forall q' \in Q_k : \{(q, \epsilon, q')\}$ $\forall q \in Q^{-\psi}, \forall q' \in Q_m : \{(q, \epsilon, q')\}$ $\forall q \in Q_l^\psi, \forall q' \in Q_k : \{(q, \epsilon, q')\}$ $\forall q \in Q_l^{-\psi}, \forall q' \in Q_m : \{(q, \epsilon, q')\}$
---	--
 - (b) Suppose a loop body contains the `break` statement s_b . Let the set of possible states before the `break` statement be Q . We add transitions of the form: $\forall q \in Q, \forall q' \in Q_m : \{(q, \epsilon, q')\}$.
 4. A statement s_j that alters the current cursor position initializes the window w to a new cursor position. The state q of \mathcal{AM} before the execution of s_j encodes the action of \mathcal{M} on the current input node n_i . Let τ be a boolean valued expression, which is *true* iff the output node in the composite graph n_o , conforms to $\mathcal{M}(n_i)$ and *false* otherwise. (For details on how the check is performed please refer to the Appendix.). Let the next node that the automaton reads be n' , with $n'_i = \Gamma_i(n')$. Let $n'_i \rightarrow \text{data} = d'$ and $n'_i \rightarrow \text{next} = p'$. The next state q' after execution of s_j is $q' = (k, d', p', 0, 0)$, where k is the line number of the statement following s_j in the control flow graph. The state q_{rej} represents a *reject* state. Let the set of possible states prior to s_j be Q . We add a set of transitions of the form: $\forall q \in Q^\tau : \{(q, n', q')\}$, $\forall q \in Q^{-\tau} : \{(q, \epsilon, q_{rej})\}$, and $\forall \sigma \in \Sigma \cup \{\epsilon\} : \{(q_{rej}, \sigma, q_{rej})\}$. Intuitively, if a node is found in a composite graph such that the input and output parts of the node do not conform to the action of the method, the automaton rejects that composite graph.
 5. Let Q_{last} be the set of possible states after executing the last statement of \mathcal{M} . We add transitions of the form: $\forall q \in Q_{last} : \{(q, \epsilon, q_{acc})\}$, and $\forall \sigma \in \Sigma \cup \{\epsilon\} : \{(q_{acc}, \sigma, q_{acc})\}$.

The transition relation computed by the above algorithm is *partial* and in order to make it complete, we add transitions (q, ϵ, q_{rej}) for all states q which do not have a successor. The number of states of the method automaton is bounded above by $O(|\mathcal{M}|)$, since cur_p, new_p range over $|w|$ values; cur_d, new_d range over \mathcal{D} ; and $|w|$ and $|\mathcal{D}|$ are fixed constants. In practice, the size of the data domain $|\mathcal{D}|$ can be significantly reduced by techniques such as data abstraction. For instance, for a method that searches for a node with a particular data value d , we can easily abstract the data domain to have just two values, $\mathcal{D}' = \{0, 1\}$, where $\forall x \in \mathcal{D} : x \mapsto 0$ (if $x \neq d$) and $x \mapsto 1$ (if $x = d$). Similarly, we can apply techniques such as reachable state space analysis to further reduce the size of the automaton. Note that, for a method operating on a tree, the automaton deterministically chooses a path in the tree, and trivially accepts along all other

branches.

Remark: Our approach can be extended to handle methods that perform a bounded number of passes over the input graph. The basic idea is to encode the changes for each pass in the composite graph. Assuming that we make at most k destructive passes, the composite graph is represented as a k -tuple, $G_c = (G_0, G_1, \dots, G_k)$ with $G_0 = G_i$ and $G_k = G_o$. Intuitively, the result of the j^{th} traversal is encoded as G_j and the automaton can verify that the graph $G_j = \mathcal{M}(G_{j-1})$. We use colors $\{red_1, \dots, red_k\}$, $\{green_1, \dots, green_k\}$ and $\{black\}$ to define the annotation encoding the k^{th} traversal in the composite graph. The color red_i , ($green_i$, resp.) represents nodes or edges deleted (added, resp.) in the i th traversal of the method. Note that these colors are annotations in the composite graph and not related to the coloring of the automaton states.

5 Property specification

We use automata as the specification language for properties. A property automaton \mathcal{A}_φ is a finite tree automaton specified as a tuple $(\Sigma, Q, \delta, q_0, \Phi)$, where all symbols have the usual meanings as described in Section 2.1. We assume that the states of the automaton are colored using a coloring function $c : Q \rightarrow \{c_0, \dots, c_k\}$.

Existence of a cycle: A rooted directed graph is said to have a cycle if there exists some path in the graph which visits a node infinitely often. The property automaton for checking existence of a cycle in a binary graph (maximum out-degree 2) has the form: $\mathcal{A}_\varphi = (\Sigma, \{q, q_f\}, \delta, q, \{c(q_f) = c_1, c(q) = c_2\})$, where the transition relation is given as: $\delta(q, n) = (q_f, q_f)$ when $n = \text{null}$, $\delta(q, n) = (q, q)$ when $n \neq \text{null}$ and $\delta(q_f, n) = (q_f, q_f)$ for all n including null . Intuitively, the automaton labels every node of the input graph with the state q . The automaton transitions to a final state iff the path is terminating. Thus the automaton accepts a graph iff there exists a non-terminating path along which q is visited infinitely often. Note that the automaton for the complement property, *i.e.* acyclicity is obtained by simply reversing the coloring of the states.

Reachability of a given data value: Suppose, given a binary tree we wish to determine if there exists a node with a given data value (*key*) reachable from the unique root node of the graph. Intuitively, the automaton non-deterministically guesses a node with the desired value and then checks it. If the desired node is found, then the automaton transits to a final state for each child node. Formally the automaton is given as: $(\Sigma, \{q, q_f\}, \delta, q, \{c(q) = c_1, c(q_f) = c_2\})$. The transition relation is defined as: $\delta(q, n) = \{(q, q_f), (q_f, q)\}$ when $n \rightarrow \text{data} \neq \text{key}$ and $n \neq \text{null}$; $\delta(q, n) = (q_f, q_f)$ when $n \rightarrow \text{data} = \text{key}$ and $\forall n : \delta(q_f, n) = (q_f, q_f)$.

Sortedness: A linked structure satisfies the sortedness property if within each bounded window of size two, the value of the current node is smaller (or greater) than the successor node. An automaton that checks if a list is sorted in ascending

order rejects the list iff there exists a window such that the data value of the current node is greater than the data value of the successor node.

6 Application: Insertion in a singly linked list

We wish to make sure that the method `InsertNode` that inserts a node in an acyclic singly linked list, preserves *acyclicity*. Since the underlying data structure is a linear list, the method automaton and the property automata are string automata. A representation of the method automaton obtained by the algorithm in Section 4.1 is shown in Figure 1. In the figure, ψ is the loop condition, ϕ is the test expression of the `if` statement $\phi = \text{cursor} \rightarrow \text{data} == \text{value}$, and τ is the boolean expression which is true iff $n_o = \mathcal{M}(n_i)$. Q_i s represent sets of states of the automaton. A dotted arrow represents an ϵ -transition and a solid line indicates a normal transition. The states q_{rej} and q_{acc} represent the reject and accept states respectively.

```

method InsertNode (value, newValue){
  1: cursor := head;
  2: while (cursor != null) {
      [ncursor := cursor->next]
  3:   if (cursor->data == value) {
  4:     cursor->next := new node {
        data := newValue;
        next := ncursor;};
  5:     break; }
  6:   cursor := ncursor when true; } }

```

The property automaton for checking acyclicity is given as: $\mathcal{A}_\varphi = (\Sigma, \{q, q_f\}, \delta, q, \{c(q) = c_1, c(q_f) = c_2\})$. The complement automaton $\mathcal{A}_{\neg\varphi}$ is given by reversing the coloring of q and q_f . The transition relation for the automaton is given as: $\delta(q, n) = q$ for $n \neq \text{null}$, $\delta(q, n) = q_f$ for $n = \text{null}$ and $\delta(q_f, n) = q_f$ for all n , including `null`. The composite automaton can be constructed in the standard fashion, and calculations show that the resultant automaton is empty, *i.e.*, the method `InsertNode` preserves acyclicity.

7 Extensions

In a variant approach, we use a suitable temporal logic in lieu of automata to specify the properties of interest. In this approach, method automaton \mathcal{A}_M is translated to formula f_M in the given logic. The property φ is specified as a formula $f_{\varphi(G_i)}$. The parameterized correctness problem reduces to checking the satisfiability of the conjunction: $f_c = f_{\varphi(G_i)} \wedge f_M \wedge f_{\neg\varphi(G_o)}$. The method M does not preserve property φ , iff f_c is satisfiable. Most temporal logics also have the nice property that the logic is closed under complementation. Thus given a property φ specified as a formula f_φ , the negation of the property is simply the

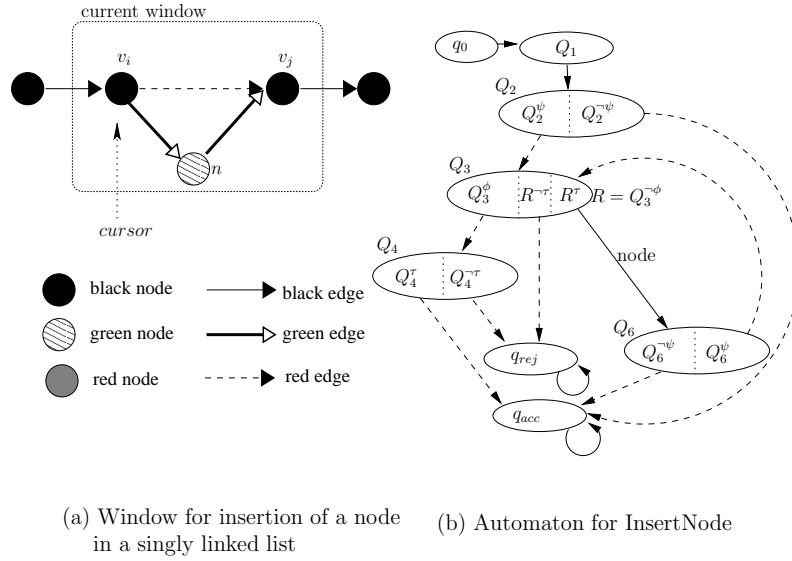


Fig. 1. Insertion of a node in a linked list

formula $\neg f_\varphi$. We take a look at two example properties that can be specified using temporal logic.

Reachability: A node n_y is said to be reachable from a node n_x if n_y can be reached from n_x by using only the **next** pointer links. Since particular nodes in the data structure are usually specified as pointers, we are interested in checking reachability of pointer expressions, where x and y are pointers to nodes n_x and n_y respectively. We introduce virtual nodes labeled with v_x and v_y , such that their **next** pointers point to n_x and n_y respectively and then check whether $\mathbf{AG}(\mathbf{EX}v_x \Rightarrow \mathbf{EF}\mathbf{EY}v_y)$. Intuitively, this formula checks that for all nodes being pointed to by $v_x \rightarrow \mathbf{next}$ (alias for x), there exists some node n_y being pointed to by $v_y \rightarrow \mathbf{next}$ (alias for y) which is reachable from x . (\mathbf{EY} (there exists some past) is a temporal operator in *CTL* with branching past).

Sharing: A node n in a data structure is called *shared* if there exist two distinct nodes, x and y in the graph such that they have n as the common immediate successor. We say that *sharing exists* in a graph if there exists a node in the graph which is *shared*. The above property (*sharing exists*) can be specified in *CTL* with branching past, as follows:

$$\exists(x, y, n) : (x \equiv \mathbf{EY}(n)) \wedge (y \equiv \mathbf{EY}(n)) \wedge \neg(x \equiv y).$$

8 Complexity analysis

The complexity of testing nonemptiness of the composite automaton \mathcal{A}_c , depends on the sizes of the property automata \mathcal{A}_φ and $\mathcal{A}_{\neg\varphi}$, and the method automaton \mathcal{A}_M . A method M having $|\mathcal{M}|$ lines of code gives rise to an automaton of size $O(|\mathcal{M}|)$ states. The number of states of the composite automaton is proportional to the product of the number of states of its constituent automata. Hence the number of states of \mathcal{A}_c is linear in the number of states of the property automata and the size of the method. Since the number of colors used for the parity condition by the property and method automata is fixed (and typically small), the number of colors used by the composite automaton is also fixed.

The complexity of checking nonemptiness of a parity tree automaton is polynomial in the number of states [EJ '91] (for a fixed number of colors in the parity acceptance condition). Thus, our solution is polynomial in the size of the method as well as the sizes of the property automata. Note that for linear graphs the method automaton and property automata can be specialized to string automata and thus the complexity of our technique is *linear*.

If we use temporal logic to specify properties, satisfiability of a formula in the *CTL* with branching past can be done in time exponential in the size of the formula [Sch '02]. We argue that the exponential cost is incurred in the construction of the tableaux from the formula. If the size of the formula is small, we can easily bear this penalty. The cost of checking emptiness of the tableau is still polynomial in the size of the tableau.

9 Conclusions and Related Work

We present an efficient solution to the parameterized correctness problem for methods operating on linked data structures. In our technique, a method is algorithmically compiled into a method automaton and properties are specified as tree automata. We construct a composite automaton, from the method automaton and the property automata, for checking if the given method preserves the given property. The property is not preserved iff the language accepted by the composite automaton is nonempty. Our technique is polynomial in the size of the method and the sizes of the property automata. In a variant approach an appropriate temporal logic can be used for specifying properties.

A key advantage of our approach is that for a broad, useful class of programs and data structures we provide an efficient algorithmic solution for verifying safety properties. Since reasoning about parameterized data structures is undecidable in general, we present a solution for methods which are known to terminate for all well-formed inputs. Techniques such as shape analysis [SRW '99], pointer assertion logic engine [MS '01] and separation logic [ORY '01] make interesting comparison with our approach, since they address a similar genre of problems.

Shape analysis is a technique for computing shape invariants for programs by providing over-approximations of structure descriptors at each program point

using 3-valued logic. In contrast to our technique which provides exact solutions, shape analysis provides imprecise (albeit conservative) results in double exponential time. In [BRS '99] the authors discuss a decidable logic L_r for describing linked data structures. However, their work does not provide a practical algorithm for checking the validity of formulas in this logic and the complexity of the given decision procedure is high.

Pointer Assertion Logic Engine tool [MS '01] encodes programs and partial specifications as formulas of monadic second order logic. Though their approach can handle a large number of data structures and methods, the complexity of the decision procedure is non-elementary. Moreover, the technique works only for loop-free code and loops need to be broken using user specified loop invariants.

Separation logic [ORY '01], which is an extension of Hoare Logic for giving proofs of partial correctness of methods, does not easily lend itself to automation. Furthermore, classical separation logic without arithmetic is not recursively enumerable [Rey '02].

In [Bou *et al.*] the authors describe a technique to verify safety properties of programs that modify data structures. Initial configurations of a program are encoded as automata and the program is translated into a transducer. The main idea is to check whether action of the transducer on the initial configurations leads to a bad configuration of the program. This problem is undecidable since a transducer could, in general, encode a Turing machine computation. The authors use abstraction-refinement to verify properties. Their technique is restricted to data structures with a single successor, and also limited by the efficiency of abstractions and the refinement process.

References

- [BRS '99] Michael Benedikt, Thomas W. Reps, Shmuel Sagiv, *A Decidable Logic for Describing Linked Data Structures*, In Proceedings of 8th European Symposium on Programming, 1999, (ESOP '99), pp. 2-19
- [Bou *et al.*] A. Bouajjani, P. Habermehl, P. Moro, T. Vojnar, *Verifying Programs with Dynamic 1-Selector-Linked Structures in Regular Model Checking*, In Proceedings of 12th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, 2005, (TACAS'05), LNCS 3440, April 2005.
- [Car '94] Olivier Carton, *Chain Automata*, In IFIP World Computer Congress 1994, Hamburg, pp. 451-458, Elsevier (North-Holland).
- [EJ '88] E. Allen Emerson, Charanjit S. Jutla, *The Complexity of Tree Automata and Logics of Programs*, In Proceedings of 29th IEEE Foundations of Computer Science, 1988, (FOCS '88), pp. 328-337.
- [EK '02] E. Allen Emerson, Vineet Kahlon, *Model Checking Large-Scale and Parameterized Resource Allocation Systems*, In Proceedings of Tools and Algorithms for the Construction and Analysis of Systems, 8th International Conference, 2002, (TACAS '02), pp. 251-265
- [EJ '91] E. A. Emerson and C. S. Jutla, *Tree Automata, Mu-Calculus and Determinacy*, (Extended Abstract), In Proceedings of Foundations of Computer Science 1991, (FOCS '91), pp. 368-377.

- [Em '85] E. Allen Emerson. *Automata, Tableaux, and Temporal Logics*, Conference on Logics of Programs, New York, NY. LNCS 193, pp. 79-88
- [HU '79] John E. Hopcroft and Jeffrey D. Ullman, *Introduction to Automata Theory, Languages and Computation*, Addison Wesley, (1979).
- [Lev-Ami *et al.*] Tal Lev-Ami, Thomas W. Reps, Shmuel Sagiv, Reinhard Wilhelm, *Putting static analysis to work for verification: A case study* In International Symposium on Software Testing and Analysis, 2000, (ISSTA'00), pp. 26-38
- [MS '01] Andres Møller, Michael I. Schwartzbach, *The Pointer Assertion Logic Engine*, In Proceedings of SIGPLAN Conference on Programming Languages Design and Implementation, 2001, (PLDI '01), pp. 221-231.
- [NSV '04] Frank Neven, Thomas Schwentick, Victor Vianu, *Finite state machines for strings over infinite alphabets*, In ACM Transactions on Computational Logic, (TOCL), Volume 15 Number 3, pp. 403-435, July 2004.
- [ORY '01] Peter O'Hearn, John Reynolds, Hongseok Yang, *Local Reasoning about Programs that Alter Data Structures*, Invited Paper, In Proceedings of 15th Annual Conference of the European Association for Computer Science Logic, 2001, (CSL '01), pp. 1-19.
- [Rey '02] John C. Reynolds, *Separation Logic: A Logic for Shared Mutable Data Structures*, In Proceedings of the 17th IEEE Symposium on Logic in Computer Science, 2002, (LICS 2002), pp. 55-74.
- [Rose '65] Arnold L. Rosenberg, *On multi-head finite automata*, FOCS 1965, pp.221-228
- [Sch '02] Ph. Schnoebelen, *The complexity of temporal logic model checking*, In Advances in Modal Logic, papers from 4th International Workshop on Advances in Modal Logic 2002, (AiML'02), Sep.-Oct. 2002, Toulouse, France.
- [SRW '99] M. Sagiv, T. Reps, and R. Wilhelm, *Parametric shape analysis via 3-valued logic*, In Symposium on Principles of Programming Languages, 1999, (POPL '99).

Appendix: Checking whether $n_o = \mathcal{M}(n_i)$

For simplicity, we discuss only methods operating on linear graphs. Let the state of the automaton before checking the condition $n_o = \mathcal{M}(n_i)$ be $q = (j, cur_d, cur_p, new_d, new_p)$. For a given node, the method can modify either the outgoing edge from the current node, add a new node m at the current position, or delete the current node. Let the old and new successor nodes be n_1 and n'_1 respectively. Let l be the coloring function for the nodes and edges. Let the (**data**, **next**) fields for the nodes n_i , n_o and m be (d, n_1) , (d', n'_1) and (d_m, m') respectively. We need to check for the following conditions:

1. $d' = cur_d$
2. If $n_1 \neq n'_1$, *i.e.* the **next** pointer of the **cursor** has changed, $(l(n_i, n_1) = red) \wedge (l(n_o, n'_1) = green)$,
3. If m exists, $(l(m) = green) \wedge (d_m = new_d) \wedge (l(m, m') = green) \wedge (l(n_o, m) = green) \wedge (l(n_i, n_1) = red)$.

Additionally, the action of a **delete** statement is checked by checking whether the color of a node is *red*.