Proving Termination of Tree Manipulating Programs

Peter Habermehl[1], Radu Iosif[2], Adam Rogalewicz[2,3], and Tomáš Vojnar[3]

[1] LSV, ENS Cachan, CNRS, INRIA; 61 av. du Président Wilson, F-94230 Cachan, France
e-mail: haberm@liafa.jussieu.fr
[2] VERIMAG, CNRS, 2 av. de Vignate, F-38610 Gières, e-mail:{iosif,rogalewi}@imag.fr
[3] FIT BUT, Božetěchova 2, CZ-61266, Brno, e-mail: vojnar@fit.vutbr.cz

Abstract. We consider the termination problem of programs manipulating tree-like dynamic data structures. Our approach is based on a counter-example guided abstraction refinement loop. We use abstract regular tree model-checking to infer invariants of the program. Then, we translate the program to a counter automaton (CA) which simulates it. If the CA can be shown to terminate using existing techniques, the program terminates. If not, we analyse the possible counterexample given by a CA termination checker and either conclude that the program does not terminate, or else refine the abstraction and repeat. We show that the spuriousness problem for lasso-shaped counterexamples is decidable in some non-trivial cases. We applied the method successfully on several interesting case studies.

1 Introduction

Verification of programs with dynamic linked data structures is a difficult task, among other reasons, due to the use of unbounded memory, and the intricate nature of pointer manipulations. Most of the approaches existing in this area concentrate on checking safety properties such as, e.g., absence of null pointer dereferences, preservation of shape invariants, etc. In this paper, we go further and tackle the universal termination problem of programs manipulating tree data structures. Namely, we are interested in proving that such a program terminates for any input tree out of a given set described as an infinite regular tree language over a finite alphabet.

We handle sequential, non-recursive programs working on trees with parent pointers and data values from a finite domain. The basic statements we consider are data assignments, non-destructive pointer assignments, and tree rotations. This is sufficient for verifying termination of many practical programs over tree-shaped data structures (e.g., AVL trees or red-black trees) used, in general, for storage and a fast retrieval of data. Moreover, many programs working on singly- and doubly-linked lists fit into our framework as well. We do not consider dynamic allocation in this version of the paper, but insertion/removal of leaf nodes, common in many practical tree manipulating programs, can be easily added, if not used in a loop.

We build on *Abstract Regular Tree Model Checking* (ARTMC) [5], a generic framework for symbolic verification of infinite-state systems, based on representing regular sets of configurations by finite tree automata, and program statements as operations on tree automata. We represent a given program as a control flow graph whose nodes are annotated with (overapproximations of) sets of reachable configurations computed using ARTMC. From the annotated control flow graph, we build a counter automaton (CA) that simulates the program. The counters of the CA keep track of different measures within the working tree: the distances from the root to nodes pointed to by certain variables, the sizes of the subtrees below such nodes, and the numbers of nodes with a certain data value. Termination of the CA is analysed by existing tools, e.g., [8, 12, 23].

Our analysis uses a *Counter-example Guided Abstraction Refinement* (CEGAR) loop [10]. If the tool we use to prove termination of the CA succeeds, this implies that the program terminates on any input from the given set. Otherwise, the CA checker tool outputs a lasso-shaped counterexample. For the class of CA generated by our translation scheme, we prove that it is decidable whether there exists a non-terminating run of the CA over the given lasso[4].

However, even if we are given a real lasso in the generated CA, due to the abstraction involved in its construction, we still do not know whether this implies also non-termination of the program. We then map the lasso over the generated CA back into a lasso in the control of the program, and distinguish two cases. If (1) the program lasso does not contain tree rotations, termination of all computations along this path is decidable. Otherwise, (2) if the lasso contains tree rotations, we can decide termination under the additional assumption that there exists a CA (not necessarily known to us) that witnesses termination of the program (i.e., intuitively, in the case when the tree measures we use are strong enough to show termination). In both cases, if the program lasso is found to be spurious, we refine the abstraction and generate a new CA from which an entire family of counterexamples (including this particular one) is excluded.

The analysis loop is not guaranteed to terminate even if the given program terminates due to the fact that our problem is not recursively enumerable. However, experience with our implementation of a prototype tool shows that the method is successfully applicable to proving termination of various real-life programs.

All proofs and more details can be found in the full version [18] of the paper.

Contributions of the Paper: (1) We have developed a systematic translation of programs working on trees into counter automata; the translation is based on an adequate choice of measures that map parts of the memory structures into positive integers. (2) We provide a new CEGAR loop for refining the translation of programs into counter automata on demand. (3) We present new decidability results for the spuriousness problem of lasso-shaped counterexamples for both counter automata and programs with trees. (4) We have implemented our techniques on top of the existing framework of *Abstract Regular Tree Model Checking*; our tool can handle examples of tree manipulating programs that, to the best of our knowledge, are not handled by any existing tool.

Related Work. The area of research on automated verification of programs manipulating dynamic linked data structures is recently quite live. Various approaches to verification of such programs differing in their principles, degree of automation, generality, and scalability have been proposed. They are based on, e.g., monadic second order logic [21], 3-valued predicate logic with transitive closure [24], separation logic [22, 17], or finite automata [16, 6].

With few exceptions, all existing verification methods for programs with recursive data structures tackle verification of safety properties. In [1, 25], specialised ranking functions over the number of nodes reachable from pointer variables are used to verify termination of programs manipulating linked lists. Termination of programs manipulating lists has further been considered in [17, 4] using constraints on the lengths of the list segments not having internal nodes pointed from outside. To the best of our knowledge, automated checking of termination of programs manipulating trees has so-far been considered in [20] only, where the Deutsch-Schorr-Waite tree traversal algorithm

[4] If the analyser used returns a spurious lasso-shaped counterexample for the termination of the CA, we suggest choosing another tool.

was proved to terminate using a manually created progress monitor, encoded in first-order logic.

In the past several years, a number of industrial-scale software model checkers such as SLAM [2], BLAST [19], or MAGIC [9] were built using the CEGAR approach [10]. However, these tools consider verification of safety properties only. On what concerns termination, CEGAR was applied in [12, 13], and implemented in the TERMINATOR [14] and ARMC [23] tools. Both of these tools are designed to prove termination of integer programs without recursive data structures.

Concerning termination of programs with recursive data structures, the available termination checkers for integer programs can be used *provided* that there is a suitable abstraction of such programs into programs over integers, i.e., counter automata. Such abstraction can be obtained by recording some numerical characteristics of the heap in the counters, while keeping the qualitative properties of the heap in the control of the CA. Indeed, this is the approach taken in [4] for checking termination of programs over singly-linked lists. The abstraction used in [4] is based on compacting each list segment into a single abstract node and recording its length in the counters of the generated CA. The number of abstract heap graphs that one obtains this way is finite (modulo the absence of garbage)—therefore they can be encoded in the control of the CA. The translation produces a CA that is *bisimilar* to the original program, and therefore any (positive or negative) result obtained by analysing the CA holds for the program.

However, in the case of programs over trees, one cannot use the idea of [4] to obtain a bisimilar CA since the number of branching nodes in a tree is unbounded. Therefore, the translation to CA that we propose here loses some information about the behaviour of the program, i.e., the semantics of the CA overapproximates the semantics of the original program. Then, if a spurious non-termination counterexample is detected over the generated CA, the translation is to be refined. This refinement is done by a specialised CEGAR loop that considers also structural information about the heaps. To the best of our knowledge, no such CEGAR loop was proposed before in the literature.

As said already above, the approach of [17] is similar to [4] in that it tracks the length of the list segments. However, it does not generate a CA simulating the original program. Instead, it first obtains invariants of the program (using separation logic) and then computes the so-called variance relations that say how the invariants change within each loop when the loop is fired once more. When the computed variance relations are well-founded, termination of the program is guaranteed. Unlike the approach of [4] (bisimulation preserving) and the approach we present here (based on CEGAR), the analysis of [17] fails if the initial abstraction is not precise enough.

The approach of [17] was recently generalised in [3] to a general framework that one can use to extend existing invariance analyses to variance analyses that can in turn be used for checking termination. Up to now, this framework has not been instantiated for programs with trees (by providing the appropriate measures and their abstract semantics). Moreover, it is not clear how the variance analysis framework fits with the CEGAR approach.

2 Preliminaries

Programs with Trees. We consider sequential, non-recursive C-like programs working over tree-shaped data structures with a finite set of pointer variables *PVar*. Each node in a tree contains a data value from a finite set *Data* and three next pointers, denoted `left`, `right`, and `up`.[5] For $x, y \in PVar$ and $d \in Data$, we allow the following statements:

[5] A generalisation of our approach to trees with another arity is straightforward.

(1) assignments $x = \texttt{null}$, $x = y$, $x = y.\{\texttt{left}|\texttt{right}|\texttt{up}\}$, $x.\texttt{data} = d$, (2) conditional statements and loops based on the tests $x == \texttt{null}$, $x == y$, $x.\texttt{data} == d$, and (3) the standard left and right tree rotations [15] (cf. Figure 1). This syntax covers a large set of practical tree-manipulating procedures. For technical reasons, we require w.l.o.g. that *no statements take the control back to the initial line*.

Memory configurations of the considered programs can be represented as trees with nodes labelled by elements of the set $\mathcal{C} = Data \times 2^{PVar} \cup \{\Box\}$—a node is either null and represented by $\Box$ or it contains a data value and a set of pointer variables pointing to it. Each pointer variable can point to at most one tree node (if it is null, it does not appear in the tree). Let $\mathcal{T}(\mathcal{C})$ be the set of all such trees and *Lab* the set of all program lines. A configuration of a program is a pair $\langle l, t\rangle \in Lab \times \mathcal{T}(\mathcal{C})$. For space reasons, the semantics of the program statements considered is given in [18].

Some program statements may influence the counters of the CA that we build to simulate programs in several different ways. For instance, after $x = x.\texttt{left}$, the distance of x from the root may increase by one, but it may also become undefined (which we represent by the special value -1) if $x.\texttt{left}$ is null. Similarly, a single rotation statement may change the distance of a node pointed by some variable from the root in several different ways according to where the node is located in a particular input tree. For technical reasons related to our abstraction refinement scheme, we need a one-to-one mapping between actions of a program and the counter manipulations simulating them. In order to ensure the existence of such a mapping, we decompose each program statement into several *instructions*. The semantics of a statement is the union of the semantics of its composing instructions, and exactly one instruction is always executable in each program configuration.

In particular, the assignments $x = \texttt{null}$ and $x = y$ are instructions. Conditional statements of the form $x == \texttt{null}$ and $x == y$ are decomposed into two instructions each, corresponding to their true and false branches. A conditional statement $x.\texttt{data} == d$ is decomposed into three instructions, corresponding to its true and false branches, and an error branch for the case $x == \texttt{null}$. Each statement $x = y.\texttt{left}$ is decomposed into instructions $\texttt{goLeftNull}(x,y)$ for the case when $y.\texttt{left} == \texttt{null}$, $\texttt{goLeftNonNull}(x,y)$ for the case $y.\texttt{left}! = \texttt{null}$, and $\texttt{goLeftErr}(x,y)$ for the case of a null pointer dereference. The statements $x = y.\texttt{right}$ and $x = y.\texttt{up}$ are treated in a similar way. The statements $x.\texttt{data} = d$ are decomposed into a set of instructions $\texttt{changeData}(x,d',d)$ for all $d' \in Data$. A special instruction $\texttt{changeDataErr}(x)$ for the null pointer dereference is also introduced.

Finally, a left rotation on a node pointed by a variable $x \in PVar$ is decomposed into a set of instructions $\texttt{leftRotate}(x,X,Y,A,B)$ where X contains variables aliased to x, Y variables pointing to the right son of x, A variables pointing inside the left subtree of x, and B variables pointing into the right subtree of the right son of x (Figure 1). The instruction $\texttt{leftRotateErr}(x)$ is introduced for the case of a null dereference within the rotation. Right rotations are decomposed analogously.

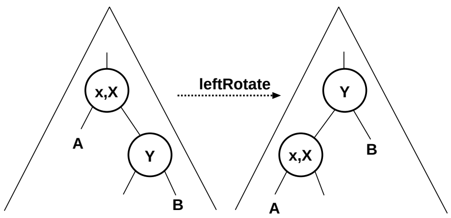

Fig. 1. $\texttt{leftRotate}(x,X,Y,A,B)$

Given a program P, we denote by *Instr* the set of instructions that appear in P and by $\langle l,t\rangle \xrightarrow[P]{i} \langle l',t'\rangle$ the fact that P has a transition from $\langle l,t\rangle$ to $\langle l',t'\rangle$ caused by firing an instruction $i \in Instr$. By $i(t)$ we denote the effect of i on a tree $t \in \mathcal{T}(\mathcal{C})$. We denote by $\xrightarrow[P]{}$ the union $\bigcup_{i \in Instr} \xrightarrow[P]{i}$, and by $\xrightarrow[P]{*}$ the reflexive and transitive closure of $\xrightarrow[P]{}$. For $i \in Instr$ and $I \subseteq \mathcal{T}(\mathcal{C})$, let $post(i,I) = \{i(t) \mid t \in I\}$. We also generalise *post* to sequences of instructions.

Counter Automata. For an arithmetic formula φ, let $FV(\varphi)$ denote the set of free variables of φ. For a set of variables X, let $\Phi(X)$ denote the set of arithmetic formulae with free variables from $X \cup X'$ where $X' = \{x' \mid x \in X\}$. If $\nu : X \to \mathbb{Z}$ is an assignment of $FV(\varphi) \subseteq X$, we denote by $\nu \models \varphi$ the fact that ν is a satisfying assignment of φ.

A counter automaton (CA) is a tuple $A = \langle X, Q, q_0, \varphi_0, \rightarrow \rangle$ where X is the set of counters, Q is a finite set of control locations, $q_0 \in Q$ is a designated initial location, φ_0 is an arithmetic formula such that $FV(\varphi_0) \subseteq X$, describing the initial assignments of the counters, and $\rightarrow \in Q \times \Phi(X) \times Q$ is the set of transition rules.

A configuration of a CA is a pair $\langle q, \nu \rangle \in Q \times (X \to \mathbb{Z})$. The set of all configurations is denoted by $\mathfrak{C}$. The transition relation $\xrightarrow[A]{\varphi} \subseteq \mathfrak{C} \times \mathfrak{C}$ is defined by $(q,\nu) \xrightarrow[A]{\varphi} (q',\nu')$ iff there exists a transition $q \xrightarrow{\varphi} q'$ such that if σ is an assignment of $FV(\varphi)$, where $\sigma(x) = \nu(x)$ and $\sigma(x') = \nu'(x)$, we have that $\sigma \models \varphi$ and $\nu(x) = \nu'(x)$ for all variables x with $x' \notin FV(\varphi)$. We denote by $\xrightarrow[A]{}$ the union $\bigcup_{\varphi \in \Phi} \xrightarrow[A]{\varphi}$, and by $\xrightarrow[A]{*}$ the reflexive and transitive closure of $\xrightarrow[A]{}$. A *run* of A is a sequence of configurations $(q_0,\nu_0),(q_1,\nu_1),(q_2,\nu_2)\ldots$ such that $(q_i,\nu_i) \xrightarrow[A]{} (q_{i+1},\nu_{i+1})$ for each $i \geq 0$ and $\nu_0 \models \varphi_0$. We denote by $\mathfrak{R}_A$ the set of all configurations reachable by A, i.e., $\mathfrak{R}_A = \{(q,\nu) \mid (q_0,\nu_0) \xrightarrow[A]{*} (q,\nu) \text{ for some } \nu_0 \models \varphi_0\}$.

3 The Termination Analysis Loop

Our termination analysis procedure based on abstraction refinement is depicted in Fig. 2. We start with the control flow graph (CFG) of the given program and use ARTMC to generate invariants for its control points. Then, the CFG annotated with the invariants (an abstract CFG, see Section 4) is converted into a CA[6], which is checked for termination using an existing tool (e.g., [23]). If the CA is proved to terminate, termination of the program is proved too. Otherwise, the termination analyser outputs a lasso-shaped counterexample. We check whether this counterexample is real in the CA—if not, we suggest the use of another CA termination checker (for brevity, we skip this in Fig. 2). If the counterexample is real on the CA, it is translated back into a sequence of program instructions and analysed for spuriousness on the program. If the counterexample is found to be real even there, the procedure reports non-termination. Otherwise, the program CFG is refined by splitting some of its nodes (actually, the sets of program configurations associated with certain control locations), and the loop is reiterated. Moreover, ARTMC may also be re-run to refine the invariants used (as briefly discussed in Section 6).

If our termination analysis stops with either a positive or a negative answer, the answer is exact. However, we do not guarantee termination for any of these cases. Indeed, this is the best we can achieve as the problem we handle is not recursively enumerable even when destructive updates (i.e., tree rotations) are not allowed. This can be proved by a reduction from the complement of the halting problem for 2-counter automata.

[6] The use of invariants in the abstract CFGs allows us to remove impossible transitions and therefore improves the accuracy of the translation to CA.

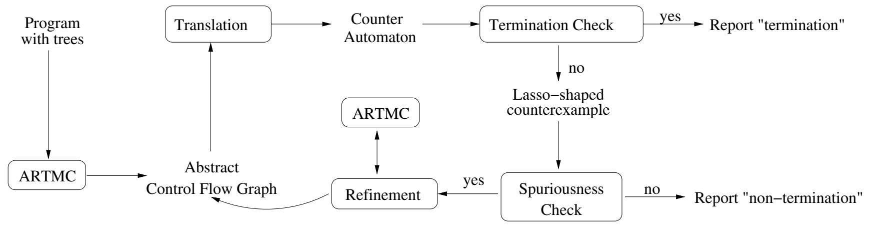

Fig. 2. The proposed abstract-check-refine loop

Theorem 1. *The problem whether a program with trees without destructive updates terminates on any input tree is not recursively enumerable.*

Therefore we do not further discuss termination guarantees for our analysis procedure in this paper, and postpone the research on potential partial guarantees, in some restricted cases, for the future. However, despite the theoretical limits, the experimental results reported in Section 7 indicate a practical usefulness of our approach.

ARTMC. We use *abstract regular tree model checking* to overapproximate the sets of configurations reachable at each line of a program (i.e., to compute *abstract invariants* for these lines) and also to check that the program is *free of basic memory inconsistencies* like null pointer dereferences. Due to space limitations, we only give a very brief overview of ARTMC here—more details can be found in [6, 18]. The idea is to represent each program configuration as a tree over a finite alphabet, regular sets of such configurations by finite tree automata, and program instructions as operations on tree automata. Starting from a regular set of initial configurations, these operations are then iteratively applied until a fixpoint is reached. In order to make the computation stop, the sets of reachable configurations (i.e., finite tree automata) are abstracted at each step. Several abstraction schemes based on collapsing states of the encountered tree automata may be used. For example, the *finite-height abstraction* collapses the automata states that accept exactly the same trees up to some height. All the abstractions are finite-range, guaranteeing termination of the abstract fixpoint computation, and can be automatically refined (e.g., in the mentioned case, by increasing the abstraction height).

For the needs of ARTMC, we encode configurations of the considered programs simply as trees over the alphabet $\mathcal{C} = Data \times 2^{PVar} \cup \{\Box\}$. Most of the instructions can be encoded as structure-preserving tree transducers. A transducer can check conditions like $x == y$ or $x.\mathtt{data} == d$ by checking node labels. Transducers can also be used to move symbols representing the variables to nodes marked by some other variable ($x = y$), remove a symbol representing a variable from the tree ($x = \mathtt{null}$), move it one level up or down ($x = y.\{\mathtt{left}|\mathtt{right}|\mathtt{up}\}$), or change the data element in the node marked by some variable ($x.\mathtt{data} = d$). The rotations are a bit more complex. They cannot be implemented as tree transducers. However, they can still be implemented as special operations on tree automata. First, a test of the mutual positioning of the variables in the tree required by their distribution in the sets X,Y,A,B is implemented as an intersection with a tree automaton that remembers which variables were seen, and in which branches. Then, we locate the automata states that accept the tree node representing the root of the rotation (cf. Figure 1), their children, and their right grandchildren. Finally, we reconnect these states in the automaton control structure in order to match the semantics of the tree rotations.

4 Abstraction of Programs with Trees into Counter Automata

In this section, we provide a translation from tree manipulating programs to counter automata such that existing techniques for proving termination of counter automata can be used to prove termination of the programs. Before describing the translation, we define the simulation notion that we will use to formalise correctness of the translation.

Let P be a program with a set of instructions $Instr$, an initial label $l_0 \in Lab$, a set of input trees $I_0 \subseteq \mathcal{T}(\mathcal{C})$, and a set of reachable configurations $\mathcal{R}_P \subseteq Lab \times \mathcal{T}(\mathcal{C})$. Let us also have a counter automaton $A = \langle X, Q, q_0, \varphi_0, \rightarrow \rangle$ with $\rightarrow \in Q \times \Phi(X) \times Q$, and a set of reachable configurations $\mathfrak{R}_A$. A function $M : X \times \mathcal{T}(\mathcal{C}) \rightarrow \mathbb{Z}$ is said to be a *measure* assigning counters integer values for a particular tree[7]. Let $\mathbf{M}(t) = \{M(x,t) \mid x \in X\}$.

Definition 1. *The program P is simulated by the counter automaton A w.r.t. $M : X \times \mathcal{T}(\mathcal{C}) \rightarrow \mathbb{Z}$ and $\theta : Instr \rightarrow \Phi$ iff there exists a relation $\sim \subseteq \mathcal{R}_P \times \mathfrak{R}_A$ such that (1) $\forall t_0 \in I_0 : \mathbf{M}(t_0) \models \varphi_0 \wedge \langle l_0, t_0 \rangle \sim \langle q_0, \mathbf{M}(t_0) \rangle$ and (2) $\forall (l_1,t_1),(l_2,t_2) \in \mathcal{R}_P \; \forall i \in Instr \; \forall (q_1,\nu_1) \in \mathfrak{R}_A : (l_1,t_1) \xrightarrow{i} (l_2,t_2) \wedge (l_1,t_1) \sim (q_1,\nu_1) \Rightarrow \exists (q_2,\nu_2) \in \mathfrak{R}_A : (q_1,\nu_1) \xrightarrow{\theta(i)} (q_2,\nu_2) \wedge (l_2,t_2) \sim (q_2,\nu_2)$.*

The measure M ensures that the counters are initially correctly interpreted over the input trees, whereas θ ensures that the counters are updated in accordance with the manipulations done on the trees. Simulation in the sense of Definition 1 guarantees that if we prove termination of the CA, the program will terminate on any $t \in I_0$.

4.1 Abstract Control Flow Graphs

According to Figure 2, we construct the CA simulating a program in two steps: we first construct the so-called *abstract control flow graph* (ACFG) of a program, and then translate it into a CA. Initially, the ACFG of a program is computed from its CFG by decorating its nodes with ARTMC-overapproximated sets of configurations reachable at each line (we keep the initial set of trees exact exploiting the fact that w.l.o.g. there are no statements leading back to the initial line). These sets allow us to exclude impossible (not fireable) transitions from the ACFG and thus derive a more exact CA. Further, in subsequent refinement iterations, infeasible termination counterexamples are excluded by splitting these sets (if this appears to be insufficient, we re-run ARTMC to compute a better overapproximation of the reachable sets of configurations). Below, we first define the notion of ACFG, then we provide its translation to counter automata.

In what follows, let P be a program with instructions $Instr$, working on trees from $\mathcal{T}(\mathcal{C})$, and let $l_0 \in Lab$ be the initial line of P. The *control flow graph* (CFG) of P is a labelled graph $F = \langle Instr, Lab, l_0, \Rightarrow \rangle$ where $l \stackrel{i}{\Rightarrow} l'$ denotes the presence of an instruction i between control locations $l, l' \in Lab$. We further suppose that the input tree configurations for P are described by the user as a (regular) set of trees $I_0 \subseteq \mathcal{T}(\mathcal{C})$. An *abstract control flow graph* (ACFG) for P is then a graph $G = \langle Instr, LI, \langle l_0, I_0 \rangle, \longmapsto \rangle$ where LI is a *finite* subset of $Lab \times 2^{\mathcal{T}(\mathcal{C})}$, $\langle l_0, I_0 \rangle \in LI$, and there is an edge $\langle l, I \rangle \stackrel{i}{\longmapsto} \langle l', I' \rangle$ iff $l \stackrel{i}{\Rightarrow} l'$ in the CFG of P and $post(i, I) \cap I' \neq \emptyset$.

[7] Intuitively, certain counters will measure, e.g., the distance of a certain node from the root, the size of the subtree below it, etc.

Note that since we work with ACFGs annotated with regular sets of configurations and since we can implement the effect of each instruction on a regular set as an operation on tree automata, we can effectively check that $post(i,I) \cap I' \neq \emptyset$, which is needed for computing the edges of ACFGs. Note also that a location in P may correspond to more than one locations in G.

We say that *G covers the invariants of P* whose set of reachable states is $\mathcal{R}_P$ iff each tree $t \in \mathcal{T}(\mathcal{C})$ that is reachable at a program line $l \in Lab$ (i.e., $\langle l,t\rangle \in \mathcal{R}_P$), appears in some of the sets of program configurations associated with l in the locations of G. Formally, $\forall l \in Lab : \mathcal{R}_P \cap (\{l\} \times \mathcal{T}(\mathcal{C})) \subseteq \{l\} \times \bigcup_{\langle l,I\rangle \in LI} I$. The following lemma captures the relation between the semantics of a program and that of an ACFG.

Lemma 1. *Let P be a program with trees and G an ACFG that covers the invariants of P. Then, the semantics of G simulates that of P in the classical sense.*

4.2 Translation to Counter Automata

We now describe the construction of a CA $A_{rsc}(G) = \langle X, Q, q_0, \varphi_0, \rightarrow\rangle$ from an ACFG $G = \langle Instr, LI, \langle l_0, I_0\rangle, \longmapsto\rangle$ of a program P such that $A_{rsc}(G)$ simulates P in the sense of Def. 1. We consider two sorts of counters, i.e., $X = X_{PVar} \cup X_{Data}$, where $X_{PVar} = \{r_x \mid x \in PVar\} \cup \{s_x \mid x \in PVar\}$ and $X_{Data} = \{c_d \mid d \in Data\}$. The role of these counters is formalised via a measure $M_{rsc} : X \times \mathcal{T}(\mathcal{C}) \rightarrow \mathbb{Z}$ in [18]. Intuitively, $M_{rsc}(r_x,t)$ and $M_{rsc}(s_x,t)$ record the distance from the root of the node n pointed to by x and the size of the subtree below n, respectively, and $M_{rsc}(c_d,t)$ gives the number of nodes with data d in a tree $t \in \mathcal{T}(\mathcal{C})$.

We build $A_{rsc}(G)$ from G by simply replacing the instructions on edges of G by operations on counters. Formally, this is done by the translation function θ_{rsc} defined in Table 1. The mapping for the instructions $x = y$.`right` and `rightRotate`(x,X,Y,A,B) is skipped in Table 1 as it is analogous to that of $x = y$.`left` and `leftRotate`(x,X,Y,A,B), respectively. Also, for brevity, we skip the instructions leading to the error state Err. As a convention, if the future value of a counter is not explicitly defined, we consider that the future value stays the same as the current value. Moreover, in all formulae, we assume an implicit guard $-1 \leq r_x < TreeHeight \wedge -1 \leq s_x < TreeSize$ for each $x \in PVar$[8] and $0 \leq c_d \leq TreeSize \wedge \sum_{d \in Data} c_d = TreeSize$ for each $d \in Data$. $TreeHeight$ and $TreeSize$ are parameters restricting the range in which the other counters can change according to a given input tree. They are needed as a basis on which termination of the resulting automaton can be shown.

Next, we define $Q = LI$, $q_0 = \langle l_0, I_0\rangle$, and $q \xrightarrow{\theta_{rsc}(i)} q'$ iff $q \stackrel{i}{\longmapsto} q'$ for all $i \in Instr$. The initial constraint φ_0 on the chosen counters can be automatically computed[9] from the regular set of input trees I_0 such that it satisfies requirement (1) of Definition 1. The following theorem shows the needed simulation relation between the counter automata we construct and the programs.

Theorem 2. *Given a program P and an ACFG G of P covering its invariants, the CA $A_{rsc}(G)$ simulates P in the sense of Definition 1 wrt. θ_{rsc} and M_{rsc}.*

[8] -1 corresponds to x being null.

[9] This can be done by computing the Parikh image of a context-free language $\mathcal{L}(I_0)$ corresponding to the regular tree language I_0. For each tree $t \in I_0$ there is a word in $\mathcal{L}(I_0)$ consisting of all nodes of t. We use special symbols to denote the position of a node in the tree relative to a given variable (under the variable, between it and the root) and the data values of nodes.

Table 1. The mapping θ_{rsc} from program instructions to counter manipulations

instruction i	counter manipulation $\theta_{rsc}(i)$
$\mathtt{if}(x == \mathtt{null})$	$r_x = -1$
$\mathtt{if}(x! = \mathtt{null})$	$r_x \geq 0$
$\mathtt{if}(x == y)$	$r_x = r_y \wedge s_x = s_y$
$\mathtt{if}(x! = y)$	$true$
$\mathtt{if}(x.\mathtt{data} == d)$	$r_x \geq 0 \wedge c_d \geq 1$
$\mathtt{if}(x.\mathtt{data}! = d)$	$r_x \geq 0 \wedge c_d < TreeSize$
$x = \mathtt{null}$	$r'_x = s'_x = -1$
$x = y$	$r'_x = r_y \wedge s'_x = s_y$
$\mathtt{goLeftNull}(x,y)$	$r_y \geq 0 \wedge s_y \geq 1 \wedge r'_x = s'_x = -1$
$\mathtt{goLeftNonNull}(x,y)$	$r_y \geq 0 \wedge s_y \geq 2 \wedge r'_x = r_y + 1 \wedge s'_x < s_y$
$\mathtt{goUpNull}(x,y)$	$r_y = 0 \wedge s_y \geq 1 \wedge r'_x = s'_x = -1$
$\mathtt{goUpNonNull}(x,y)$	$r_y \geq 1 \wedge s_y \geq 1 \wedge r'_x = r_y - 1 \wedge s'_x > s_y$
$\mathtt{changeData}(x,d,d)$	$r_x \geq 0 \wedge s_x \geq 1 \wedge c_d > 0$
$\mathtt{changeData}(x,d_1,d_2), d_1 \neq d_2$	$r_x \geq 0 \wedge s_x \geq 1 \wedge c_{d_1} > 0 \wedge c'_{d_2} = c_{d_2} + 1 \wedge c'_{d_1} = c_{d_1} - 1$
$\mathtt{leftRotate}(x,X,Y,A,B)$	$gLeftRotate(x,X,Y,A,B) \wedge aLeftRotate(x,X,Y,A,B)$

$$
\begin{aligned}
&gLeftRotate(x,X,Y,A,B) = \\
&\quad r_x \geq 0 \ \wedge\ s_x \geq 2 \ \wedge \\
&\quad (\forall v \in X : r_v = r_x \ \wedge\ s_v = s_x) \ \wedge \\
&\quad (\forall v, v' \in Y : r_v = r_x + 1 \ \wedge\ s_v < s_x \ \wedge \\
&\qquad\qquad r_v = r_{v'} \ \wedge\ s_v = s_{v'}) \ \wedge \\
&\quad (\forall v \in A : r_v \geq r_x + 1 \ \wedge\ s_v < s_x) \ \wedge \\
&\quad (\forall v \in B : r_v \geq r_x + 2 \ \wedge\ s_v < s_x - 1)
\end{aligned}
\qquad
\begin{aligned}
&aLeftRotate(x,X,Y,A,B) = \\
&\quad (\forall v \in X : r'_v = r_v + 1 \ \wedge\ s'_v < s_v) \ \wedge \\
&\quad (\forall v \in Y : r'_v = r_v - 1 \ \wedge\ s'_v > s_v) \ \wedge \\
&\quad (\forall v \in A : r'_v = r_v + 1 \ \wedge\ s'_v = s_v) \ \wedge \\
&\quad (\forall v \in B : r'_v = r_v - 1 \ \wedge\ s'_v = s_v)
\end{aligned}
$$

The generated CA $A_{rsc}(G)$ has the property that each transition $q \xrightarrow{\varphi} q'$ can be mapped back into the program instruction from which it originates. This is because the instructions onto which we decompose each program statement are assigned different formulae, by the translation function θ_{rsc}, and there is at most one statement between each two control locations of the program. Formally, we capture this by a function $\xi : Q \times \Phi \times Q \to Instr$ such that $\forall q_1, q_2 \in Q, \varphi \in \Phi : q \xrightarrow{\varphi} q' \Rightarrow q \xmapsto{\xi(q_1,\varphi,q_2)} q'$. We generalise θ_{rsc} and ξ to sequences of transitions, i.e., for a path π in A_{rsc}, $\xi(\pi)$ denotes the sequence of program instructions leading to π, and $\theta_{rsc}(\xi(\pi))$ denotes the sequence of counter operations on π obtained by projecting out the control locations from π.

5 Checking Spuriousness of Counterexamples

Since the CA A_{rsc} generated from a program P with trees is a simulation of P (cf. Theorem 2), proving termination of A_{rsc} suffices to prove termination of P. However, if A_{rsc} is not proved to terminate by the termination checker of choice, there are three possibilities: (1) A_{rsc} terminates, but the chosen termination checker did not find a termination argument, (2) both A_{rsc} as well as P do not terminate, and (3) P terminates, but A_{rsc} does not, as a consequence of the abstraction used in its construction. In all cases, the CA termination checker outputs a counterexample consisting of a finite path (stem) that leads to a cycle, both paths forming a lasso. Formally, a lasso $S.L$ over the control structure of a CA A_{rsc} is said to be *spurious* iff there exists a non-terminating run of A_{rsc} along $S.L$, and for no $t \in I_0$ does P have an infinite run along the path $\xi(S).\xi(L)$.

The three cases are dealt with in the upcoming paragraphs.

Deciding termination of CA lassos. We first show that termination of a given control loop is decidable in a CA whose transition relations are conjunctions of difference constraints, i.e. formulae of the forms $x-y \leq c$, $x'-y \leq c$, $x-y' \leq c$, or $x'-y' \leq c$ where x' denotes the future value of the counter x and $c \in \mathbb{Z}$ is an arbitrary integer constant. For this type of CA, the composed transition relation of the given control loop is also expressible as a conjunction of difference constraints. Then, this relation can be encoded as a constraint graph G such that the control loop terminates iff G contains a negative cycle (for details see [18]). Using the results of [11, 7], this fact can be encoded as a Presburger formula and hence decided. At the same time, it is clear that the CA generated via the translation function θ_{rsc} fall into the described class of CA. In particular, the constraint that each counter is bounded from below by -1 and from above by the $TreeHeight$ or $TreeSize$ parameters is expressible using difference constraints.[10]

Theorem 3. *Let $A = \langle X, Q, q_0, \varphi_0, \rightarrow \rangle$ be a counter automaton with transition relations given as difference constraints. Then, given a control loop in A, the problem whether there exists an infinite computation along the loop is decidable.*

Checking termination of program lassos. Due to the above result, we may henceforth assume that the lasso $S.L$ returned by the termination analyser has a real non-terminating run in the CA. The lasso is mapped back into a sequence of program instructions $\xi(S).\xi(L)$ forming a program lasso. Two cases may arise: either the lasso is real on the program or it is spurious.

Non-spurious program lassos. Since we do not consider dynamic allocation, the number of configurations that a program can reach from any input tree is finite. Consequently, if there is a tree t_ω from which the program will have an infinite run along a given lasso, then we can discover it by an exhaustive enumeration of trees. We handle the discovery of t_ω by evaluating the lasso for all trees of up to a certain (increasingly growing) height at the same time (by encoding them as regular tree language and using the implementation of program instructions over tree automata that we have). As we work with finite sets of trees, we are bound to visit the same set twice after a finite number of iterations, if there exists a non-terminating run along the lasso.

Spurious program lassos. We handle this case also by a symbolic iteration of a given program lasso $\sigma.\lambda$ starting with the initial set of trees. We compute iteratively the sets $post(\sigma.\lambda^k, I_0), k = 1, 2, \ldots$. In the case of lassos without destructive updates, this computation is shown to reach the empty set after a number of iterations that is bounded by a double exponential in the length of the lasso (cf. Section 5.1). In the case of lassos with destructive updates, we can guarantee termination of the iteration with the empty set provided there exists some CA A_u (albeit unknown) keeping track of the particular tree measures we consider here (formalised via the functions M_{rsc} and θ_{rsc} in Section 4.2) that simulates the given program and that terminates[11] (cf. Section 5.2). In the latter case, even though we cannot guarantee the discovery of A_u, we can at least ensure that the sequence $post(\sigma.\lambda^k, I_0)$, $k = 1, 2, \ldots$ terminates with the empty set. This gives us a basis for refining the current ACFG such that we get rid of the spurious lasso encountered, and we can go on in the search for a CA showing the termination of the given program.

[10] To encode conditions of the form $x \leq c$ we add a new variable z, initially set to zero, with the condition $z' = z$ appended to each transition, and rewrite the original condition as $x - z \leq c$.

[11] We can relax this condition by saying that A_u does not have any infinite run, not corresponding to a run of the program. For the sake of clarity, we have chosen the first stronger condition.

5.1 Deciding Spuriousness of Lassos without Destructive Updates

In this section, we show that the spuriousness problem for a given lasso in a program with trees is decidable, if the lasso does not contain destructive updating instructions, i.e., tree rotations. The argument for decidability is that, if there exists a non-terminating run along the loop, then there exists also a non-terminating run starting from a tree of size bounded by a constant depending on the program. Thus, there exists a tree within this bound that will be visited infinitely many often.[12]

Given a loop without destructive updates, we first build an abstraction of it by replacing the $\texttt{go}\{\texttt{Left}|\texttt{Right}|\texttt{Up}\}\texttt{Null}(x,y)$ instructions by $x = \texttt{null}$, and by eliminating all $\texttt{changeData}(x,d_1,d_2)$ instructions and the tests. Clearly, if the original loop has a non-terminating computation, then its abstraction will also have a non-terminating run starting with the same tree. The loop is then encoded as an iterative linear transformation which, for each pointer variable $x \in PVar$, has a counter p_x encoding the binary position of the pointer in the current tree using 0/1 as the left/right directions. Additionally, the most significant bit of the encoding is required to be always one, which allows for differentiating between, e.g., the position 001 encoded by $9 = (1001)_2$, and 0001 encoded by $17 = (10001)_2$. Null pointers are encoded by the value 0. The program instructions are translated into counter operations as follows:

$$x = \texttt{null} : p_x = 0 \qquad x = y : p_x = p_y \qquad \texttt{goLeftNonNull}(x,y) : p_x = 2\star p_y$$
$$\texttt{goRightNonNull}(x,y) : p_x = 2\star p_y + 1 \qquad \texttt{goUpNonNull}(x,y) : p_x = \tfrac{1}{2} p_y$$

where $2\star$ and $\frac{1}{2}$ denote the integer functions of multiplication ($x \mapsto 2x$) and division ($x \mapsto x/2$). Assuming that we have n pointer variables, each program instruction is modelled by a linear transformation of the form $\mathbf{p}' = A\mathbf{p} + B$ where A is an $n \times n$ matrix with at most one non-null element, which is either $1, 2$ or $\frac{1}{2}$, and B is an n-column vector with at most one 1 and the rest 0[13]. The composition of the instructions on the loop is also a linear transformation, except that A has at most one non-null element on each line, which is either I, or a composition of $2\star$'s and $\frac{1}{2}$'s.

Since A has at most one non-null element on each line, one can extract an $m \times m$ matrix A_0 for some $m \leq n$ that has exactly one non-null element on each line and column. Our proof is based on the fact that there exists some constant k bounded by $O(3^m)$ such that ${A_0}^k$ is a diagonal matrix. Intuitively, this means that the position of each pointer at step $i+k$ is given by a linear function of the position of the pointer at i. Then A^i is an exponential function of i. As there is no dynamic allocation of nodes in the tree, the non-termination hypothesis implies that the positions of pointers have to stay in-between bounds. But this is only possible if the elements of the main diagonal of ${A_0}^k$ are either I or compositions of the same number of $2\star$ and $\frac{1}{2}$. Intuitively, this means that all pointers are confined to move inside bounded regions of the working tree.

Theorem 4. *Let P be a program over trees, $PVar$ and $Data$ be its sets of pointer variables and data elements, $\mathcal{C} = Data \times 2^{PVar} \cup \{\square\}$, $I_0 \subseteq \mathcal{T}(\mathcal{C})$ be an initial set of trees, and $\sigma.\lambda$ be a lasso of P. Then, if P has an infinite run along the path $\sigma.\lambda^\omega$ for some $t_0 \in I_0$, then there exists a tree $t_{b0} \in \mathcal{T}(\mathcal{C})$ of height bounded by $(\|PVar\| + 1) \cdot \big(|\sigma| + |\lambda| \cdot 3^{\|PVar\|}\big)$ such that P, started with t_{b0}, has an infinite run along the same path.*

---

[12] Since there is no dynamic allocation, all trees visited starting with a tree of size k will also have size k. Hence each run of the program will either stop, or re-visit the same program configuration after a bounded number of steps.

[13] We interpret the matrix operations over the semiring of integer functions $\langle \mathbb{N} \to \mathbb{N}, +, \circ, 0, I \rangle$, where $\circ$ is functional composition and I is the identity function.

Decidability of spuriousness is an immediate consequence of this theorem. Also, there is a bound on the number of symbolic unfoldings of a spurious lasso starting with the initial set of trees.

Corollary 1. *Let P be a program over trees, $PVar$ and $Data$ its sets of pointer variables and data elements, $\mathcal{C} = Data \times 2^{PVar} \cup \{\square\}$, and $I_0 \subseteq \mathcal{T}(\mathcal{C})$ an initial set of trees. Given a lasso $S.L$ in the CA $A_{rsc}(G)$ built from an ACFG G of P, let $\sigma = \xi(S)$ and $\lambda = \xi(L)$. Then, if $\sigma.\lambda$ does not contain destructive updates, its spuriousness is decidable. Moreover, if the lasso is spurious, for all $k \geq |\lambda| \cdot \mathbf{max}(2, \|Data\|)^{2^{(\|PVar\|+1)\cdot(|\sigma|+|\lambda|\cdot 3^{\|PVar\|})}}$, we have $post(\sigma.\lambda^k, I_0) = \emptyset$.*

Despite the double exponential bound, experimental evidence (see Section 7) shows that the number of unfoldings necessary to eliminate a spurious lasso is fairly small.

5.2 Analysing Lassos with Destructive Updates

Theorem 5 stated below shows that spuriousness of a lasso that contains destructive updates, i.e., tree rotations, is decidable if there exists a *terminating* CA A_u, not necessarily known to us, simulating the program wrt. θ_{rsc} and M_{rsc}. That is, if there exists a termination argument for the program based on the tree measures we use, then we can prove spuriousness of the lasso by a symbolic iteration of the initial set.

Theorem 5. *Let P be a program with an ACFG G and let $S.L$ be a spurious lasso in $A_{rsc}(G)$. If there exists a CA A_u that simulates P wrt. θ_{rsc} and M_{rsc} and that terminates on all inputs, then there exists $k \in \mathbb{N}$ such that $post(\xi(S).\xi(L)^k, I_0) = \emptyset$.*

Indeed, imagine that for any $l \in \mathbb{N}$ there is an input tree $t_l \in I_0$ for which $\xi(S).\xi(L)^l$ is fireable. Then, the CA A_u, having a finite-control, and simulating P has to contain a lasso with a stem $S.L^{n_1}$ and a loop L^{n_2} for some $n_1, n_2 \in \mathbb{N}$ (i.e., a possibly partially unfold $S.L$). However, as the set of initial counter valuations of A_u must include the one of $A_{rsc}(G)$ (as that is the smallest possible wrt. M_{rsc}), this means that A_u has a non-terminating run also, which is a contradiction.

6 Abstraction Refinement

Let P be a program, $G = \langle Instr, LI, \langle l_0, I_0 \rangle, \longmapsto \rangle$ be an ACFG of P, and $A_{rsc}(G)$ be the CA obtained by the translation described in Section 4. Let $S.L$ be a spurious lasso, i.e., a path over which $A_{rsc}(G)$ has an infinite run, while P does not have an infinite run over the corresponding program path $\sigma.\lambda$ where $\sigma = \xi(S)$ and $\lambda = \xi(L)$. Then, we produce a new ACFG $G_{S.L}$ of P such that $A_{rsc}(G_{S.L})$ will not exhibit any lasso-shaped path with a stem labelled with the sequence of counter operations $\theta_{rsc}(\sigma).\theta_{rsc}(\lambda)^p$ and a loop labelled with $\theta_{rsc}(\lambda)^q$ for any $p, q \geq 0$. Provided that the spuriousness of the lasso $S.L$ is detected using either Corollary 1 or Theorem 5, we know that there exists $k > 0$ such that $post(\sigma.\lambda^l, I_0) = \emptyset$ for all $l \geq k$. To build the refined ACFG $G_{S.L}$, we use the sets $post(\sigma.\lambda^i, I_0)$, $0 \leq i < k$, computed in the spuriousness analysis of $S.L$ (cf. Section 5).

We refine G into $G_{S.L}$ by *splitting* some of its locations $\langle l_i, I_i \rangle \in LI$ into several locations of the form $\langle l_i, I_{ij} \rangle$, and by recomputing the edges according to the definition of ACFG (cf. Section 4.1). Intuitively, the sets I_{ij} form a partition of I_i such that I_{ij} will contain all trees from I_i that are visited in *at most* j iterations of the loop. As we prove in [18], since we keep apart the sets of trees for which the lasso may be iterated a different number of times, $A_{rsc}(G_{S.L})$ will not contain lassos of the form $\theta_{rsc}(\sigma).\theta_{rsc}(\lambda)^p.\big(\theta_{rsc}(\lambda)^q\big)^\omega$ for any $p,q > 0$.

Due to the fact that our verification problem is not r.e. (cf. Theorem 1), the Abstract-Check-Refine loop might diverge. One situation in which this can happen is when, at each refinement step, a certain line invariant I_i is split such that one of the parts, say I_{ij}, is finite (e.g., it contains only trees up to some height). However, to exclude a spurious counterexample, it might be the case that I_i has to be split into infinite sets according to some more general property (e.g., the same number of red and black nodes).[14] In such situations, we use a heuristic *acceleration* method consisting in applying the finite height abstraction α of ARTMC (cf. Section 3) to split the line invariants. That is, apart from splitting I_i wrt. the sets $post(\sigma.\lambda^i, I_0)$, we split wrt. the sets $\alpha(post(\sigma.\lambda^i, I_0))$ too. In Section 7, we report on an example in which the accelerated refinement based on the finite height abstraction was successfully used to make the analysis converge.

Another problem that may occur is that the invariants computed by ARTMC may not be precise enough for proving termination of the given program. That is why after a predefined number of steps of refining ACFGs by splitting, we *repeat ARTMC with a more precise abstraction* (e.g., we increase the abstraction height). We re-run ARTMC on the underlying CFG of the last computed ACFG G and restrict the computed reachability sets to the sets appearing in the locations of G in order to preserve the effect of the refinement steps done so far. Due to the space restrictions, we provide a detailed description of these issues in [18].

7 Implementation and Experimental Results

To demonstrate the applicability of our approach, we tested it on several real procedures manipulating trees. We restricted the ARTMC tool from [6] to binary trees with parent pointers and added support for tree rotations, instead of using general purpose destructive updates. The absence of null pointer dereferences was verified fully automatically. Termination of the generated CA was checked using the ARMC tool [23].

We first considered the following set of case studies (for more details see [18]): (1) a non-recursive *depth-first tree traversal*, (2) a procedure for *searching a data value in a red-black tree* [15] (with the actual data abstracted away and all the comparisons done in a random way), and (3) the procedure that *rebalances red-black trees after inserting a new element* using tree rotations [15]. In the latter two cases, the set of input trees was a regular overapproximation of all red-black trees (we abstracted away the balancedness condition).

The results of the experiments that we performed on a PC with a 1.4 GHz Intel Xeon processor are summarised in Table 2. The table contains the ARTMC running times (T_{ARTMC}), the number of states of the largest invariant generated by ARTMC ($|Q|_{Inv}$), the time spent by the ARMC tool to show termination (T_{CA}), and the number of counters (N_{cnt}), locations (N_{loc}), and transitions (N_{tr}) of the CA.

Table 2. Experimental Results

Example	T_{ARTMC}	$\lvert Q\rvert_{Inv}$	T_{CA}	N_{cnt}	N_{loc}	N_{tr}
Depth-first tree traversal	43s	67	10s	5	15	20
RB-search	2s	22	1s	3	8	11
RB-rebalance after insert	1m 9s	87	36s	7	44	66

[14] A similar case is encountered in classical abstraction refinement for checking safety properties.

For the three above programs, it turned out to be possible to prove the termination without a need of refinement. In the third experiment, we could even remove checking of the condition of the red-black trees (a red node has only black sons), leading to smaller verification times, with less precise invariants, which were, however, still precise enough for the termination proof.

To test our refinement procedure, we applied it on another case study where the initial invariants were not sufficient to prove termination. In particular, we considered the procedure in Figure 3 that marks the elements of a singly-linked list as even or odd, depending on whether their distance to the end of the list is even or odd. As the procedure does not know the length of the list and cannot use back-pointers, it tries to mark the first element as even, and at the end of the list, it checks whether the last element was marked as odd. If this is true, the marking is correct, otherwise the marking has to be reversed.

```
bool odd = false;
if (list != null) then
  while (true) do
    x = list;
    while (x != null) do
      x.data = odd;
      odd = not(odd);
      x = x.next;
    od
    if (not(odd)) then break;
  od
```

Fig. 3. A procedure marking elements of a list as odd or even from the end of the list

For this procedure, even if one builds the CA starting with the exact line invariants, termination cannot be established. To establish termination, one has to separate configurations where the procedure is marking the list in a correct way from those where the marking is incorrect. Then, the outer loop of the procedure will not appear in the CA at all since, in fact, it can be fired at most twice: once when the initial guess is correct and twice otherwise. The challenge is to recognise this fact automatically.

We managed to verify termination of the procedure on an arbitrary input list after excluding 9 spurious lassos (in 2 cases, the refinement was accelerated by the use of the finite-height abstraction on the I_{ij} sets that resulted from splitting line invariants when excluding certain spurious lassos).

8 Conclusion

We addressed the problem of proving termination of a significant class of tree manipulating programs. We provide a counter-example guided abstraction refinement loop based on the ARTMC framework and on exploiting the existing work on checking termination of counter automata. A number of results related to the decidability of the spuriousness problem of lasso-shaped termination counterexamples were given. Our method is not guaranteed to stop (as the problem is not r.e.), but when it stops, it provides a precise answer (both in the positive and negative case). The method was experimentally tested to be successful on several interesting practical programs.

Future work includes a more efficient implementation of our framework as well as its extension to more complex programs (like, e.g., programs with unbounded dynamic allocation and general destructive pointer updates). We would also like to further investigate cases in which the universal termination problem is r.e. (and hence allowing complete verification techniques).

Acknowledgement. The work was supported by the French Ministry of Research (ACI Securité Informatique), the Czech Grant Agency (projects 102/07/0322, 102/05/H050), the Czech-French Barrande project 2-06-27, and the Czech Ministry of Education by project MSM 0021630528 *Security-Oriented Research in Information Technology*.

References

1. I. Balaban, A. Pnueli, and L.D. Zuck. Shape Analysis by Predicate Abstraction. In *Proc. of VMCAI'05*, *LNCS* 3385. Springer, 2005.
2. T. Ball and S.K. Rajamani. The SLAM Toolkit. In *Proc. of CAV'01*, *LNCS* 2102. Springer, 2001.
3. Josh Berdine, Aziem Chawdhary, Byron Cook, Dino Distefano, and Peter O'Hearn. Variance Analyses from Invariance Analyses. In *Proc. of POPL'07*. ACM Press, 2007.
4. A. Bouajjani, M. Bozga, P. Habermehl, R. Iosif, P. Moro, and T. Vojnar. Programs with Lists are Counter Automata. In *Proc. of CAV'06*, *LNCS* 4144. Springer, 2006.
5. A. Bouajjani, P. Habermehl, A. Rogalewicz, and T. Vojnar. Abstract Regular Tree Model Checking. *ENTCS*, 149:37–48, 2006.
6. A. Bouajjani, P. Habermehl, A. Rogalewicz, T. Vojnar. Abstract Regular Tree Model Checking of Complex Dynamic Data Structures. In *Proc. of SAS'06*, *LNCS* 4134. Springer, 2006.
7. M. Bozga, R. Iosif, and Y. Lakhnech. Flat Parametric Counter Automata. In *Proc. of ICALP'06*, *LNCS* 4052. Springer, 2006.
8. A.R. Bradley and Z. Manna and H.B. Sipma. The Polyranking Principle. In *Proc. of ICALP'05*, *LNCS* 3580. Springer, 2005.
9. S. Chaki, E. Clarke, A. Groce, J. Ouaknine, O. Strichman, K. Yorav. Efficient Verification of Sequential and Concurrent C Programs. *Formal Methods in System Design*, 25(2–3), 2004.
10. E.M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-Guided Abstraction Refinement. In *Proc. of CAV'00*, *LNCS* 1855. Springer, 2000.
11. H. Comon and Y. Jurski. Multiple Counters Automata, Safety Analysis and Presburger Arithmetic. In *Proc. of CAV'98*, *LNCS* 1427. Springer, 1998.
12. B. Cook, A. Podelski, and A. Rybalchenko. Abstraction Refinement for Termination. In *Proc. of SAS'05*, *LNCS* 3672. Springer, 2005.
13. Byron Cook, Andreas Podelski, and Andrey Rybalchenko. Termination Proofs for Systems Code. In *Proc. of PLDI'06*. ACM Press, 2006.
14. Byron Cook, Andreas Podelski, and Andrey Rybalchenko. Terminator: Beyond Safety. In *Proc. of CAV'06*, *LNCS* 4144. Springer, 2006.
15. T.H. Cormen, C.E. Leiserson, R.L. Rivest. *Introduction to Algorithms*. The MIT Press, 1990.
16. J.V. Deshmukh, E.A. Emerson, and P. Gupta. Automatic Verification of Parameterized Data Structures. In *Proc. of TACAS'06*, *LNCS* 3920. Springer, 2006.
17. D. Distefano, J. Berdine, B. Cook, and P.W. O'Hearn. Automatic Termination Proofs for Programs with Shape-shifting Heaps. In *Proc. of CAV'06*, *LNCS* 4144. Springer, 2006.
18. P. Habermehl, R. Iosif, A. Rogalewicz, and T. Vojnar. Proving Termination of Tree Manipulating Programs. Verimag, TR-2007-1. `www-verimag.imag.fr/index.php?page=techrep-list`.
19. T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Software Verification with Blast. In *Proc. of 10th SPIN Workshop*, *LNCS* 2648. Springer, 2003.
20. A. Loginov, T.W. Reps, and M. Sagiv. Automated Verification of the Deutsch-Schorr-Waite Tree-Traversal Algorithm. In *Proc. of SAS'06*, *LNCS* 4134. Springer, 2006.
21. A. Møller and M.I. Schwartzbach. The Pointer Assertion Logic Engine. In *Proc. of PLDI'01*. ACM Press, 2001.
22. J.C. Reynolds. Separation Logic: A Logic for Shared Mutable Data Structures. In *Proc. of LICS'02*. IEEE CS Press, 2002.
23. A. Rybalchenko. The ARMC tool. URL: `www.mpi-inf.mpg.de/~rybal/armc/`.
24. S. Sagiv, T.W. Reps, and R. Wilhelm. Parametric Shape Analysis via 3-valued Logic. *TOPLAS*, 24(3), 2002.
25. E. Yahav, T. Reps, M. Sagiv, and R. Wilhelm. Verifying Temporal Heap Properties Specified via Evolution Logic. In *Proc. of ESOP'03*, *LNCS* 2618. Springer, 2003.