Automata-Based Termination Proofs

Radu Iosif, Adam Rogalewicz

▶ **To cite this version:**

Radu Iosif, Adam Rogalewicz. Automata-Based Termination Proofs. Computing and Informatics, 2013, 22, pp.739-775. hal-01418867

HAL Id: hal-01418867

https://hal.science/hal-01418867

Submitted on 17 Dec 2016

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Public Domain

Computing and Informatics, Vol. 22, 2003, 1001–1035, V 2013-Apr-16

AUTOMATA-BASED TERMINATION PROOFS

Radu Iosif, Adam Rogalewicz

VERIMAG, CNRS
2 av. de Vignate
38610 Gières, France

FIT, Brno University of Technology
IT4Innovations Centre of Excellence
Božetěchova 2
612 66 Brno, Czech Republic
e-mail: iosif@imag.fr, rogalew@fit.vutbr.cz

Abstract. This paper describes our generic framework for detecting termination of programs handling infinite and complex data domains, such as pointer structures. The framework is based on a counterexample-driven abstraction refinement loop. We have instantiated the framework for programs handling tree-like data structures, which allowed us to prove automatically termination of programs such as the depth-first tree traversal, the Deutsch-Schorr-Waite tree traversal, or the linking leaves algorithm.

Keywords: Formal verification, Termination analysis, Büchi automata, Tree automata, Programs with pointers

1 INTRODUCTION

Proving termination is an important challenge for the existing software verification tools, requiring specific analysis techniques [26, 8, 24]. The basic principle underlying all these methods is proving that, in every (potentially) infinite computation of the program, there exists a suitable measure which decreases infinitely often.

MSC2000: 68N30, 68Q60, 68Q70, 68W30

The commonly used measures are from so-called well-founded domains. The well-foundedness guarantees that in the domain, there is no infinite decreasing sequence. As a consequence, the measure cannot decrease infinitely often, and hence the program must terminate.

In this paper, we provide an extended and improved description of our termination analysis, originally presented in [21]. The proposed termination analysis is based on the following principles:

1. We consider programs working on infinite data domains $\langle D, \preceq_1, \ldots, \preceq_n \rangle$ equipped with an arbitrary number of well-founded partial orders.
2. For any transformation $\Rightarrow \subseteq D \times D$ induced by a program statement, and any partial order on D $\preceq_i$, $1 \leq i \leq n$, we assume that the problem $\Rightarrow \cap \preceq_i \stackrel{?}{=} \emptyset$ is decidable algorithmically.
3. An abstraction of the program is built automatically and checked for the existence of potential non-terminating execution paths. If such a path exists, then an infinite path of the form $\sigma\lambda^\omega$ (called *lasso*) is exhibited.
4. Due to the over-approximation involved in the construction of the abstraction, the lasso found may be *spurious*, i.e., it may not correspond to a real execution of the program. In this case we use domain-specific procedures to detect spuriousness, and, if the lasso is found to be spurious, the abstraction is refined by eliminating it.

The framework described here needs to be instantiated for particular classes of programs, by providing the following ingredients:

- well-founded relations $\preceq_1, \ldots, \preceq_n$ on the working domain D. In principle, their choice is naturally determined by the working domain. As an example, if D is the set of terms (trees) over a finite ranked alphabet, then $\preceq_i$ can be classical well-founded orderings on terms (e.g., Recursive Path Ordering, Knuth-Bendix Ordering, etc.).
- a decision procedure for the problems $\Rightarrow \cap \preceq_i \stackrel{?}{=} \emptyset$, $1 \leq i \leq n$, where $\Rightarrow$ is any transition relation induced by a program statement. This is typically achieved by choosing suitable symbolic representations for relations on D, which are closed under intersection, and whose emptiness problem is decidable. For instance, this is the case when both $\Rightarrow$ and $\preceq_i$ can be encoded using finite (tree) automata.
- a decision procedure for the spuriousness problem: given a lasso $\sigma\lambda^\omega$, where σ and λ are finite sequences of program statements, does there exist an initial data element $d_0 \in D$ such that the program, started with d_0, has an infinite execution along the path $\sigma\lambda^\omega$?

The main reason for which we currently ask the user to provide the relations is that our technique is geared towards data domains which cannot be encoded by a finite number of descriptors, such as tree-structured domains, and more complex pointer structures. Well-founded relations for classical such domains (e.g., terms

over a ranked alphabet) are provided in the literature. Moreover, we are not aware of efficient techniques for automatic discovery of well-founded relations on such domains, which is an interesting topic for future research.

Providing suitable representations for the well-founded relations, as well as for the program transitions enables the framework to compute an initial abstraction of the program. The initial abstraction is an automaton which has the same control states as the program, and each edge in the control flow graph of the program is covered by one or more transitions labeled with relational symbols.

The abstraction is next checked for the existence of potentially non-terminating executions. This check uses the information provided by the well-founded relations, and excludes all lassos for which there exists a strictly decreasing well-founded relation $\succ_i$, $1 \leq i \leq n$, that holds between the entry and exit of the loop body. This step amounts to checking non-emptiness of the intersection between the abstraction and a predefined Büchi automaton. If the intersection is empty, the original program terminates, otherwise a lasso-shaped counterexample of the form $\sigma\lambda^\omega$ is exhibited.

Deciding spuriousness of lassos is also a domain-dependent problem. For integer domains, techniques exist in cases where the transition relation of the loop is a difference bound matrix (DBM) [7] or an affine transformation with the finite monoid property [18]. For general affine transformations, the problem is currently open, as [10] mentions it[1]. For tree-structured data domains, the problem is decidable in cases, where the loop does not modify the structure of trees [20].

If a lasso is found to be spurious, the program model is refined by excluding the lasso from the abstraction automaton. In our framework based on Büchi Automata, this amounts to intersecting the abstraction automaton with the complement of the Büchi Automaton representing the lasso. Since a lasso is trivially a Weak Deterministic Büchi Automaton (WDBA), complementation increases the size of the automaton by at most one state, and is done in constant time. This refinement scheme can be extended to exclude entire families of spurious lassos, also described by WDBA.

We have instantiated the framework to the verification of programs handling tree data structures. The basic statements we consider are data assignments, non-destructive pointer assignments, creation of leaves, and deletion of leaves. As an extension, we also allow tree rotations. This is a sufficient fragment 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 provide two families of well-founded relations on trees, (i) a lexicographical ordering on positions of program variables and (ii) a subset relation on nodes labeled with a given data element (from a finite domain). Program statements as well as the well-founded relations are encoded using tree automata, which

[1] In fact the spuriousness problem for integer affine transition relations covers another open problem, that of detecting a zero in an integer linear recurrent sequence. The latter has been shown to be NP-hard in [2] but no decidability results have been found so far.

provide an efficient method for checking emptiness of intersections between relations. For programs on trees without destructive updates, the spuriousness problem has been shown decidable in [20].

The presented well-founded relations for trees fits very well for the encoding from [6], where data structures more complex than trees are encoded using a tree-like backbone and a regular set of extra edges. The encoding itself is quite general and allows us, on one hand, to handle structures like, e.g., trees with root pointers, or trees with linked leaves, and, on the other hand, to handle destructive updates.

A prototype tool has been implemented on top of the ARTMC [6, 4] invariant generator for programs with dynamic linked data structures. Experimental results include push-button termination proofs for the Deutsch-Schorr-Waite tree traversal, deleting nodes in red-black trees, as well as for the Linking Leaves procedures. Most of these programs could not be verified by existing approaches.

Contributions of the Paper:

1. The paper presents our original generic approach for termination analysis of programs. The approach is based on Büchi automata, well-founded relations, and CEGAR [11] loop.
2. We provide a set of well-founded relations on trees.
3. We instantiate the generic framework to programs manipulating tree like data structures as well as programs manipulating more general data structures (encoded as a tree-backbone and a regular set of extra edges).
4. We have implemented our technique in the prototype tool based on ARTMC [6, 4]. We can prove termination of examples which are, to the best of our knowledge, not handled by any other tool.

Related Work. Efficient techniques have been developed in the past for proving termination of programs with integer variables [26, 8, 9, 14, 28]. This remains probably the most extensively explored class of programs, concerning termination.

Recently, techniques for programs with singly-linked lists have been developed in [3, 17, 23]. These techniques rely on tracking numeric information related to the sizes of the list segments. An extension of this method to tackle programs handling trees has been given in our previous work [20]. Unlike the works on singly-linked lists from [3, 17], where refinement (of the counter model) is typically not needed, in [20] we considered a basic form of counterexample-driven refinement, based on a symbolic unfolding of the lasso-shaped counterexample.

Abstraction refinement for termination has been first considered in [14], where the refinement consists of discovering and adding new well-founded relations to the set of relations used by the analysis. Since techniques for the discovery of well-founded relations (based on, e.g., spurious program loops) are available only for integer domains, it is not clear for the time being whether the algorithm proposed in [14] can be also applied to programs handling pointer structures.

Several ideas in this paper can be also found elsewhere. Namely, (1) extracting variance assertions from loop invariants was reported in [1], (2) using Büchi automata to encode the non-termination condition of the program was introduced by [24], and (3) proving termination for programs handling tree-like data structures was also considered in [20]. However, the method presented here distinguishes itself from the body of existing work, on the following aspects:

1. The framework is general and can be instantiated to any class of programs, whose semantic domains are known to have well-founded orderings. In particular, we provide well-founded orderings on a domain of trees, e.g., lexicographical orderings, that cannot be directly encoded in quantifier-free linear arithmetic, whereas all variant assertions from [1] are confined to quantifier-free linear arithmetic. Moreover, we consider examples in which the variant relations considered track the evolution of an unbounded number of data elements (tree node labels), whereas in [1] only a finite number of seed variables could be considered.

2. We provide an automated method of abstracting programs into Büchi automata, whereas the size-change graphs from [24] are produced manually. Moreover, the use of Büchi automata to encode the termination condition provides us with a natural way of *refining* the abstract model, by intersection of the model with the complement of the counterexample, encoded by a Weak Deterministic Büchi Automaton.

3. We generalize the refinement based on Weak Deterministic Büchi Automata to exclude infinite sets of spurious counterexamples, all at once. On the other hand, the refinement given in [20] could only eliminate one counterexample at the time, at the cost of expanding the model by unfolding the lasso a number of times exponential in the number of program variables. In our setting, the size of the refined model is theoretically bounded by the product between the size of the model and the size of the lasso. Since, in practice the lassos are found to be rather small, the blowup caused by the refinement does not appear to be critical.

Automated checking of termination of programs manipulating trees has been also considered in [25], where the Deutsch-Schorr-Waite tree traversal algorithm was proved to terminate using a manually created progress monitor, encoded in first-order logic. In our approach, this example could be verified using the common well-founded relations on trees that is, without adding case-specific information to the framework.

The rest of the paper is organized as follows: The Section 2 describes preliminaries. The Section 3 contains general description of our termination analysis framework. The Section 4 describes the way, how the framework is instantiated for trees and more complex data structures. The section 5 presents our experimental results. And finally, we conclude the paper by Section 6.

2 PRELIMINARIES

2.1 Well-founded Relations

Definition 1. Given a set X, a relation $\preceq \subseteq X \times X$ is *well-founded* iff

1. $\preceq$ is a partial order on X
2. $\forall x \in X$ there is no infinite sequence $x_0, x_1, x_2, \dots$ such that (i) $x_0 = x$ and (ii) $\forall i \geq 0 : x_{i+1} \preceq x_i \wedge x_{i+1} \neq x_i$

As an example of a well-founded relation, we can take a standard $\leq$ ordering on natural numbers. But the same standard $\leq$ ordering on positive real numbers is not well-founded, because there exists, e.g., the sequence $1 > 0.1 > 0.01 > 0.001 > \dots$

2.2 Büchi Automata

This section introduces the necessary notions related to the theory of Büchi automata [22]. Let $\Sigma = \{a, b, \dots\}$ be a finite alphabet. We denote by Σ^* the set of finite words over Σ, and by Σ^ω we denote the set of all infinite words over Σ. For an infinite word $w \in \Sigma^\omega$, let $\inf(w)$ be the set of symbols occurring infinitely often in w. If $u, v \in \Sigma^*$ are finite words, uv^ω denotes the infinite word $uvvv\dots$.

Definition 2. A *Büchi automaton* (BA) over Σ is a tuple $A = \langle S, I, \rightarrow, F\rangle$, where: S is a finite set of *states*, $I \subseteq S$ is a set of *initial states*, $\rightarrow \subseteq S \times \Sigma \times S$ is a *transition relation* – we denote $(s, a, s') \in \rightarrow$ by $s \xrightarrow{a} s'$, and $F \subseteq S$ is a set of *final states*. The *size* of the automaton A is denoted as $\|A\|$ and it is equal to the number of states—i.e., $\|A\| = \|S\|$.

A run of A over an infinite word $a_0a_1a_2 \dots \in \Sigma^\omega$ is an infinite sequence of states $s_0s_1s_2 \dots$ such that $s_0 \in I$ and for all $i \geq 0$ we have $s_i \xrightarrow{a_i} s_{i+1}$. A run π of A is said to be *accepting* iff $\inf(\pi) \cap F \neq \emptyset$. An infinite word w is *accepted* by a Büchi automaton A iff A has an accepting run on w. The *language* of A, denoted by $\mathcal{L}(A)$, is the set of all words accepted by A.

It is well-known that Büchi-recognizable languages are closed under union, intersection and complement. For two Büchi automata A and B, let $A \otimes B$ be the automaton recognizing the language $\mathcal{L}(A) \cap \mathcal{L}(B)$. It can be shown that $\|A \otimes B\| \leq 3 \cdot \|A\| \cdot \|B\|^2$.

A Büchi automaton $A = \langle S, I, \rightarrow, F\rangle$ is said to be *complete* if for every $s \in S$ and $a \in \Sigma$ there exists $s' \in S$ such that $s \xrightarrow{a} s'$. A is said to be *deterministic*

[2] Intersection of A and B is done by a synchronous product of these two automata. For each pair of states, one has to remember an extra information whether a finite state has been seen in the automaton A, in the automaton B or nowhere. Therefore the size of the resulting automaton is bounded by $3 \cdot \|A\| \cdot \|B\|$. See, e.g., [22] for details.

(DBA) if I is a singleton, and for each $s \in S$ and $a \in \Sigma$, there exists at most one state $s' \in S$ such that $s \xrightarrow{a} s'$. A is moreover said to be *weak* if, for each strongly connected component $C \subseteq S$, either $C \subseteq F$ or $C \cap F = \emptyset$. It is well-known that complete weak deterministic Büchi automata can be complemented by simply reverting accepting and non-accepting states. Then, for any Weak Deterministic Büchi automaton (WDBA), we have that $\|\overline{A}\| \leq \|A\| + 1$, where $\overline{A}$ is the automaton accepting the language $\Sigma^\omega \setminus \mathcal{L}(A)$—i.e., the *complement* of A.

2.3 Trees and Tree Automata

Definition 3. (Binary alphabet and tree) For a partial mapping $f : A \rightarrow B$ we denote $f(x) = \bot$ the fact that f is undefined at some point $x \in A$. The domain of f is denoted $dom(f) = \{x \in A \mid f(x) \neq \bot\}$. For a set A we denote by $A_\bot$ the set $A \cup \{\bot\}$.

Given a finite set of *colors* $\mathcal{C}$, we define the *binary alphabet* $\Sigma_\mathcal{C} = \mathcal{C} \cup \{\Box\}$, where the *arity* function is $\forall c \in \mathcal{C}.\#(c) = 2$ and $\#(\Box) = 0$. Π denotes the set of tree positions $\{0,1\}^*$. Let $\epsilon \in \Pi$ denote the empty sequence, and $p.q$ denote the concatenation of sequences $p, q \in \Pi$. $p \leq_{pre} q$ denotes the fact that p is a prefix of q and $p \leq_{lex} q$ is used to denote the fact that p is less than q in the lexicographical order. We denote by $p \simeq_{pre} q$ the fact that either $p \leq_{pre} q$, or $p \geq_{pre} q$. A *tree* t over $\mathcal{C}$ is a partial mapping $t : \Pi \rightarrow \Sigma_\mathcal{C}$ such that $dom(t)$ is a finite prefix-closed subset of Π, and for each $p \in dom(t)$:

- if $\#(t(p)) = 0$, then $t(p.0) = t(p.1) = \bot$,
- otherwise, if $\#(t(p)) = 2$, then $p.0, p.1 \in dom(t)$.

When writing $t(p) = \bot$, we mean that t is undefined at position p.

Let t_ϵ be the empty tree, $t_\epsilon(p) = \bot$ for all $p \in \Pi$. A *subtree* of t starting at position $p \in dom(t)$ is a tree $t_{|p}$ defined as $t_{|p}(q) = t(pq)$ if $pq \in dom(t)$, and undefined otherwise. $t[p \leftarrow c]$ denotes the tree that is labeled as t, except at position p where it is labeled with c. $t\{p \leftarrow u\}$ denotes the tree obtained from t by replacing the $t_{|p}$ subtree with u. We denote by $\mathcal{T}(\mathcal{C})$ the set of all trees over the binary alphabet $\Sigma_\mathcal{C}$.

Definition 4. A (binary) *tree automaton* [13, 27] over an alphabet $\Sigma_\mathcal{C}$ is a tuple $A = (Q, F, \Delta)$ where Q is a set of states, $F \subseteq Q$ is a set of final states, and Δ is a set of transition rules of the form:
(i) $\Box \rightarrow q$ or (ii) $c(q_1, q_2) \rightarrow q$, $c \in \mathcal{C}$. The *size* of the automaton A is denoted as $\|A\|$ and it is equal to the number of states—i.e., $\|A\| = \|Q\|$.

A *run* of A over a tree $t : \Pi \rightarrow \Sigma_\mathcal{C}$ is a mapping $\pi : dom(t) \rightarrow Q$ such that for each position $p \in dom(t)$, where $q = \pi(p)$, we have:

- if $\#(t(p)) = 0$ (i.e., if $t(p) = \Box$), then $\Box \rightarrow q \in \Delta$,
- otherwise, if $\#(t(p)) = 2$ and $q_i = \pi(p.i)$ for $i \in \{0,1\}$, then $t(p)(q_0, q_1) \rightarrow q \in \Delta$.

A run π is said to be *accepting* if and only if $\pi(\epsilon) \in F$. The *language* of A, denoted as $\mathcal{L}(A)$, is the set of all trees over which A has an accepting run. A set of trees $T \subseteq \mathcal{T}(\mathcal{C})$ (a tree relation $R \subseteq \mathcal{T}(\mathcal{C}_1 \times \mathcal{C}_2)$) is said to be *rational* if there exists a tree automaton A such that $\mathcal{L}(A) = T$ (respectively, $\mathcal{L}(A) = R$).

Example: Let $\mathcal{C} = \{\circ, \bullet\}$ and the following tree automaton over $\Sigma_{\mathcal{C}}$: $A = (\{q, r\}, \{r\}, \Delta_A)$ with $\Delta_A = \{\square \rightarrow q, \circ(q,q) \rightarrow q, \bullet(q,q) \rightarrow r, \circ(r,q) \rightarrow r, \circ(q,r) \rightarrow r\}$. This automaton accepts all binary trees which contains exactly one node labeled by $\bullet$. An example of runs can be seen in Fig. 1. Note that for some trees there is no run which maps a state to the most-top node (symbol X corresponds to non-existing mapping), and hence such a tree is not accepted.

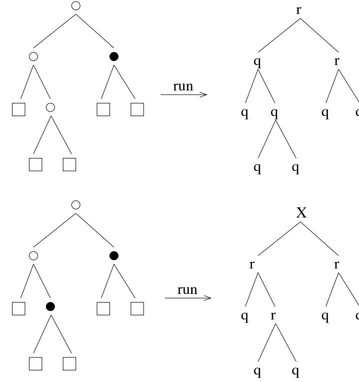


Fig. 1.

Remark: In this paper, we use only tree automata restricted to binary trees. But in general, the tree automata can accept trees of arbitrary arity. See [13, 27] for general definitions. Note that standard finite automata are special case of tree automata, where all rules are unary (except the initial one $\square \rightarrow q$, which represents the initial state in standard finite automaton).

2.3.1 Rational Tree Relations

A pair of trees $(t_1, t_2) \in \mathcal{T}(\mathcal{C}_1) \times \mathcal{T}(\mathcal{C}_2)$ can be encoded by a tree over the alphabet $(\mathcal{C}_1 \cup \{\square, \perp\}) \times (\mathcal{C}_2 \cup \{\square, \perp\})$, where $\#(\langle \perp, \perp \rangle) = 0$, $\#(\langle \alpha, \perp \rangle) = \#(\langle \perp, \alpha \rangle) = \#(\alpha)$ if $\alpha \neq \perp$, and $\#(\langle \alpha_1, \alpha_2 \rangle) = 2$ if $\alpha_1 \in \mathcal{C}_1 \ \wedge \ \alpha_2 \in \mathcal{C}_2$. The projection functions are defined as usual, i.e., for all $p \in dom(t)$ we have $pr_1(t)(p) = c_1$ if $t(p) = \langle c_1, c_2 \rangle$ and $pr_2(t)(p) = c_2$ if $t(p) = \langle c_1, c_2 \rangle$. Finally, let $\mathcal{T}(\mathcal{C}_1 \times \mathcal{C}_2) = \{t \mid pr_1(t) \in \mathcal{T}(\mathcal{C}_1) \text{ and } pr_2(t) \in \mathcal{T}(\mathcal{C}_2)\}$ be a set of all pair trees. A set $R \subseteq \mathcal{T}(\mathcal{C}_1 \times \mathcal{C}_2)$ is further called a *tree relation*.

Definition 5. A tree relation $R \subseteq \mathcal{T}(\mathcal{C}_1 \times \mathcal{C}_2)$ is said to be *rational* if there exists a tree automaton A such that $\mathcal{L}(A) = R$.

Definition 6. For two relations $R' \subseteq \mathcal{T}(\mathcal{C} \times \mathcal{C}')$ and $R'' \subseteq \mathcal{T}(\mathcal{C}' \times \mathcal{C}'')$ we define the composition $R' \circ R'' = \{\langle pr_1(t'), pr_2(t'')\rangle \mid t' \in R',\ t'' \in R'',\ pr_2(t') = pr_1(t'')\}$.

It is well-known that rational tree languages are closed under union, intersection, complement and projection. As a consequence, rational tree relations are closed under composition.

3 THE TERMINATION ANALYSIS FRAMEWORK

In this section, we describe our termination analysis framework and demonstrate its principles on a simple running example.

3.1 Programs and Abstractions

First we introduce a model for programs handling data from a possibly infinite domain D equipped by a set of partial orders $\preceq_1, \ldots, \preceq_n$, where $\preceq_i \subseteq D \times D$ for $1 \leq i \leq n$. In the following, we use the notion $\langle D, \preceq_1, \ldots, \preceq_n\rangle$ to denote the data domain with the partial orders. Then we define program abstractions as Büchi automata.

Definition 7. (Instruction) Let $\langle D, \preceq_1, \ldots, \preceq_n\rangle$ be a data domain. An instruction is a pair $\langle g, a\rangle$ where $g \subseteq D$ is called the *guard* and $a : D \to D$ is called the *action*.

The guard represents a condition, which must be true before the action is executed. The guards are used to model conditional statements from programming languages—e.g., one instruction $\langle g, a_1\rangle$ is used for *then* branch and another one $\langle D \setminus g, a_2\rangle$ for *else* branch. An unspecified guard is assumed to be the entire domain D.

Definition 8. (Program) Let $\langle D, \preceq_1, \ldots, \preceq_n\rangle$ be a data domain and $\mathfrak{I}$ be a set of instructions. A program over the set of instructions $\mathfrak{I}$ is a graph $P = \langle \mathfrak{I}, L, l_0, \Rightarrow\rangle$, where L is the set of *control locations*, $l_0 \in L$ is the *initial location*, and $\Rightarrow \subseteq L \times \mathfrak{I} \times L$ is the *edge relation* denoted as $l \overset{g:a}{\Longrightarrow} l'$. We assume furthermore, that there is at most one instruction in between any two control locations, i.e., if $l \overset{g_1:a_1}{\Longrightarrow} l'$ and $l \overset{g_2:a_2}{\Longrightarrow} l'$ then $g_1 = g_2$ and $a_1 = a_2$. This condition is common in programming languages.

In the following, we will use both textual and graphical representations of programs.

Definition 9. (Configuration, execution, reachable configuration)
Let $\langle D, \preceq_1, \ldots, \preceq_n\rangle$ be a data domain, $P = \langle \mathfrak{I}, L, l_0, \Rightarrow\rangle$ a program and $D_0 \subseteq D$ be a set of initial data values. A *program configuration* is a pair $\langle l, d\rangle \in L \times D$, where l is a control location and d is a data value.

An *execution* is a (possibly infinite) sequence of program configurations $\langle l_0, d_0\rangle, \langle l_1, d_1\rangle, \langle l_2, d_2\rangle, \ldots$ starting with the initial program location l_0 and some configuration $d_0 \in D$ such that, for all $i \geq 0$ there exists an edge $l_i \stackrel{g:a}{\Longrightarrow} l_{i+1}$ in the program, such that $d_i \in g$ and $d_{i+1} = a(d_i)$.

A configuration $\langle l, d\rangle$ is said to be *reachable* if there exists $d_0 \in D_0$, and the program P has an execution from $\langle l_0, d_0\rangle$ to $\langle l, d\rangle$.

Definition 10. (Invariant) Let $\langle D, \preceq_1, \ldots, \preceq_n\rangle$ be a data domain, $P = \langle \mathfrak{I}, L, l_0, \Rightarrow\rangle$ a program and $D_0 \subseteq D$ be a set of initial data values. An *invariant* of the program (with respect to the set D_0) is a function $\iota : L \rightarrow 2^D$ such that, for each $l \in L$, if $\langle l, d\rangle$ is reachable, then $d \in \iota(l)$. If the dual implication holds, we say that ι is an *exact invariant*.

Definition 11. Given a program $P = \langle \mathfrak{I}, L, l_0, \Rightarrow\rangle$ working over a domain $\langle D, \preceq_1, \ldots, \preceq_n\rangle$ we define the alphabet $\Sigma_{(P,D)} = L \times \{>, \bowtie, =\}^n$. For a tuple $\rho \in \{>, \bowtie, =\}^n$, we define $[\rho] \in D \times D$ as : $d\ [\rho]\ d'$ if and only if, $\rho = \langle r_1, \ldots, r_n\rangle$ and for all $1 \leq i \leq n$:

- $d \succ_i d'$ iff r_i is $>$,
- $d \not\preceq_i d'$ iff r_i is $\bowtie$,
- $d \approx_i d'$ iff r_i is $=$.

Definition 12. (Abstraction) Let $P = \langle \mathfrak{I}, L, l_0, \Rightarrow\rangle$ be a program, and $\langle D, \preceq_1, \ldots, \preceq_n\rangle$ be a domain. A Büchi automaton $A = \langle S, I, \rightarrow, F\rangle$ over $\Sigma_{(P,D)}$ is said to be an abstraction of P if and only if, for every infinite execution of P : $\langle l_0, d_0\rangle\langle l_1, d_1\rangle\langle l_2, d_2\rangle \ldots$, there exists an infinite word $\langle l_0, \rho_0\rangle\langle l_1, \rho_1\rangle\langle l_2, \rho_2\rangle \ldots \in \mathcal{L}(A)$ such that $d_i\ [\rho_i]\ d_{i+1}$,for all $i \geq 0$.

Consequently, if P has a non-terminating execution, then its abstraction A will be non-empty. However, for reasons related to the complexity of the universal termination problem, one cannot in general build an abstraction of a program that will be empty if and only if the program terminates.

3.1.1 The Running Example: A Program and Its Abstraction

We will now demonstrate the abstraction of a real program on our running example. Let us consider the program in Fig. 2, working on a binary tree data structure, in which each node has two pointers to its `left`- and `right`-sons and one pointer `up` to its parent. We assume that leaves have null `left` and `right` pointers, and the `root` has a null `up` pointer.

The first loop (lines 2,3) terminates because the variable x is bound to reach a node with `x.left = null` (or `x.left.right = null`), since the tree is finite and no new nodes are created. The second loop (lines 4,5) terminates because no matter where x points to in the beginning, by going up, it will eventually reach the `root`

```
1 x := root;
2 while (x.left != null) and (x.left.right != null)
3         x := x.left.right;
4 while (x != null)
5         x := x.up;
```

Fig. 2.

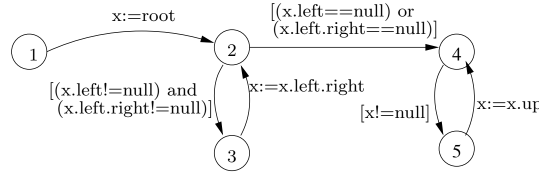


Fig. 3. Guards are represented inside "[ ]" brackets, and actions without brackets. Unspecified action is equal to identity relation.

and then become null. Fig. 3 represents the c-like program from Fig. 2 as a program according to the definition 8.

Now, we are going to create an abstraction of program in Fig. 3 Let us suppose the following well-founded ordering: for any two trees t_1 and t_2, we have $t_1 \geq_x t_2$ if and only if the position of the pointer variable x in t_2 is a prefix of the position of the variable x in t_1. Using the $\geq_x$ ordering, we build an abstraction of the program given in Fig. 4. The states in the abstract program correspond to line numbers in the original program, and every state is considered to be accepting (w.r.t Büchi accepting condition), initially.

Note that the action $x := root$ is abstracted by two edges—first, labeled by $=_x$, describes the case when x was originally placed in the root node and the other one, labeled by $>_x$, describes the cases when x was originally deeper in the tree.

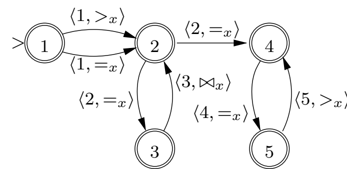


Fig. 4.

3.2 Building Abstractions Automatically

The first question is how to build abstractions of programs effectively. Here, we propose a method that performs well under the following assumptions: There exists

a symbolic representation $\mathcal{S}$ (e.g., some logic or automata) such that (a) the program instructions can be represented using the $\mathcal{S}$, (b) the well-founded relations on the working domain can be also represented using $\mathcal{S}$, (c) $\mathcal{S}$ is closed under projection, intersection and complement, and (d) the emptiness problem for $\mathcal{S}$ is decidable. In the concrete case of programs with trees (see Section 4), we use finite tree automata [13] as a suitable symbolic representation $\mathcal{S}$.

Definition 13. (Initial abstraction) Given a program $P = \langle \mathfrak{I}, L, l_0, \Rightarrow \rangle$ working over the domain $\langle D, \preceq_1, \ldots, \preceq_n \rangle$, and an invariant $\iota : L \rightarrow 2^D$, with respect to a set of initial data values D_0, the *initial abstraction* is the Büchi automaton $A_P^\iota = \langle L, \{l_0\}, \rightarrow, L \rangle$, where, for all $l, l' \in L$ and $\rho \in \{>, \bowtie, =\}^n$, we have :

$$l \xrightarrow{\langle l,\rho \rangle} l' \iff l \overset{g:a}{\Longrightarrow} l' \text{ and } pr_1(R_{\langle g,a \rangle} \ \cap \ [\rho]) \ \cap \ \iota(l) \neq \emptyset \tag{1}$$

where $R_{\langle g,a \rangle} = \{(d, d') \in D \mid d \in g, \ d' = a(d)\}$ and, for a relation $R \subseteq D \times D$, we denote by $pr_1(R) = \{x \mid \exists y \in D \ . \ \langle x, y \rangle \in R\}$.

Intuitively, a transition between l and l' is labeled with a tuple of relational symbols ρ if and only if there exists a program instruction between l and l' and a pair of reachable configurations $\langle l, d \rangle, \langle l', d' \rangle \in L \times D$ such that $d[\rho]d'$ and the program can move from $\langle l, d \rangle$ to $\langle l', d' \rangle$ by executing the instruction $\langle g, a \rangle$. The intuition is that every transition relation induced by the program is "covered" by all partial orderings that have a non-empty intersection with it. Notice also that, for any $d, d' \in D$, there exists $\rho \in \{>, \bowtie, =\}^n$ such that $d \ [\rho] \ d'$. For reasons related to abstraction refinement, that will be made clear in the following, the transition in the Büchi automaton A_P^ι is also labeled with the source program location l[3]. As an example, Fig. 2 (b) gives the initial abstraction for the program in Fig. 2 (a).

The program invariant $\iota(l)$ from (1) is needed in order to limit the coverage only to the relations involving configurations reachable at line l. In principle, we can compute a very coarse initial abstraction by considering that $\iota(l) = D$ at each program line. However, using stronger invariants enables us to compute more precise program abstractions. The following lemma proves that the initial abstraction respects Def. 12.

Lemma 1. Let P be a program working over the domain $\langle D, \preceq_1, \ldots, \preceq_n \rangle$, and $D_0 \subseteq D$ be an initial set, and $\iota : L \rightarrow 2^D$ be an invariant with respect to the initial set D_0, the Büchi automaton A_P^ι is an abstraction of P.

Proof. Let $\langle l_0, d_0 \rangle \langle l_1, d_1 \rangle \langle l_2, d_2 \rangle \ldots$ is an arbitrary infinite execution of the program P. Then for each $i \geq 0$: $l_i \overset{g_i:a_i}{\Longrightarrow} l_{i+1}$, $d_i \in g_i$, and $a_i(d_i) = d_{i+1}$—i.e., $(d_i, d_{i+1}) \in R_{\langle g_i,a_i \rangle}$, where $R_{\langle g,a \rangle} = \{(d, d') \in D \mid d \in g, \ d' = a(d)\}$. Moreover $d_i \in \iota(l_i)$ and

[3] After the abstraction refinement, the relation between program locations and büchi automata states is no more $1:1$ (as in the initial abstraction) but it is $m:n$. The labels are then used to relate the edges in the abstract model with the corresponding program actions.

$d_{i+1} \in \iota(l_{i+1})$. We know that there exists ρ_i such that $d_i[\rho_i]d_{i+1}$. Therefore for each $i \geq 0$ there exists an edge $l_i \xrightarrow{\langle l_i, \rho_i \rangle} l_{i+1}$ in the automaton A_P^ι, i.e., the word $\langle l_0, \rho_0 \rangle \langle l_1, \rho_1 \rangle \langle l_2, \rho_2 \rangle \ldots \in \mathcal{L}(A_P^\iota)$, where $d_i\ [\rho_i]\ d_{i+1}$. □

3.3 Checking Termination on Program Abstractions

In light of Def. 12, if a Büchi automaton A is an abstraction of a program P, then each accepting run of A reveals a *potentially* infinite execution of P. However, the set of accepting runs of a Büchi automaton is, in general infinite, therefore an effective termination analysis cannot attempt to check whether each run of A corresponds to a real computation of P. We propose an effective technique, based on the following assumption:

Assumption 1. The given domain is $\langle D, \preceq_1, \ldots, \preceq_n \rangle$ for a fixed $n > 0$, and the partial orders $\preceq_i$ are well-founded, for all $i = 1, \ldots, n$.

Consequently, any infinite word $\langle l_0, \rho_0 \rangle \langle l_1, \rho_1 \rangle \langle l_2, \rho_2 \rangle \ldots \in \mathcal{L}(A)$ from which we can extract[4] a sequence $(\rho_0)_i (\rho_1)_i (\rho_2)_i \ldots \in (=^* >)^\omega$, for some $1 \leq i \leq n$, cannot correspond to a real execution of the program, in the sense of Definition 12. Therefore, we must consider only the words for which, for all $1 \leq i \leq n$, either:

1. there exists $K \in \mathbb{N}$ such that, $(\rho_k)_i$ is $=$, for all $k \geq K$, or
2. for infinitely many $k \in \mathbb{N}$, $(\rho_k)_i$ is $\bowtie$.

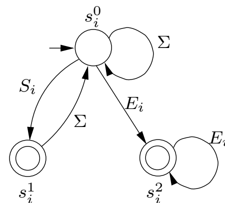


Fig. 5.

The condition above can be encoded by a Büchi automaton defined as follows. Consider that $\Sigma_{(P,D)} = L \times \{>, \bowtie, =\}^n$ is fixed. Let $S_i = \{\langle l, (r_1, \ldots, r_n) \rangle \in \Sigma_{(P,D)} \mid r_i \text{ is } \bowtie\}$ and $E_i = \{\langle l, (r_1, \ldots, r_n) \rangle \in \Sigma_{(P,D)} \mid r_i \text{ is } =\}$, for $1 \leq i \leq n$. With this notation, let B_i be the Büchi automaton recognizing the ω-regular language $\Sigma^*(S_i \Sigma^*)^\omega \cup \Sigma^* E_i^\omega$. This automaton is depicted in Fig. 5. Since the above condition holds for all $1 \leq i \leq n$, we need to compute $B = \bigotimes_{i=1}^n B_i$. Finally, the automaton which accepts all words witnessing potentially non-terminating runs of the original program is $A \otimes B$.

[4] Each element $\langle l, \rho \rangle$ in the sequence is first converted into ρ by omitting the l-part. Then all the relations except the ith one are projected out of ρ. The result is denoted as $(\rho)_i$

If A is an abstraction of P and $\mathcal{L}(A \otimes B) = \mathcal{L}(A) \cap \mathcal{L}(B) = \emptyset$, we can infer that P has no infinite runs. Otherwise, it is possible to exhibit a lasso-shaped non-termination witness of the form $\sigma\lambda^\omega \in \mathcal{L}(A \otimes B)$, where $\sigma, \lambda \in \Sigma^*$ are finite words labeling finite paths in $A \otimes B$. In the rest of the paper, we refer to σ as to the *stem* and to λ as to the *loop* of the lasso. The following lemma proves the existence of lasso-shaped counterexamples.

Lemma 2. Let $\langle D, \preceq_1, \ldots, \preceq_n \rangle$ be a well-founded domain, A be a Büchi automaton representing the abstraction of a program and $B = \bigotimes_{i=1}^n B_i$ be a Büchi automaton representing the non-termination property such that $\mathcal{L}(A \otimes B) \neq \emptyset$. Then there exists $\sigma\lambda^\omega \in \mathcal{L}(A \otimes B)$ for some $\sigma, \lambda \in \Sigma^*_{(P,D)}$, such that $|\sigma|, |\lambda| \leq \|A\| \cdot (n+1) \cdot 2^n$.

Proof. If $\mathcal{L}(A \otimes B) \neq \emptyset$, then $A \otimes B$ has a run π in which at least one final state s occurs infinitely often. Let σ be the word labeling the prefix of π from the beginning to the first occurrence of s, and λ be the word labeling the segment between the first and second occurrences of s on π. Then $\sigma\lambda^\omega \in \mathcal{L}(A \otimes B)$.

To prove the bound on $|\lambda|$, we consider that B is the result of a generalized product $\bigotimes_{i=1}^n B_i$, whose states are of the form $\langle s_1, \ldots, s_n, k \rangle$ where $s_i \in \{s_i^0, s_i^1, s_i^2\}$ is a state of B_i (cf. Fig 5) and $k \in \{0, 1, \ldots, n\}$. Since λ is the label of a cycle in $A \otimes B$, the projection of the initial and final states on $B_1, \ldots, B_n$ must be the same. Then for each state in the cycle, either $s_i \in \{s_i^0, s_i^1\}$ or $s_i = s_i^2$, for each $1 \leq i \leq n$. This is because the projection of a cycle from $A \otimes B$ on B_i, $1 \leq i \leq n$ is again a cycle, and the only cycles in B_i are composed either of $\{s_i^0, s_i^1\}$ or $\{s_i^2\}$. Hence $|\lambda| \leq \|A\| \cdot (n+1) \cdot 2^n$. A similar reasoning is used to establish the bound on $|\sigma|$. $\square$

Despite the exponential bound on the size of the counterexamples, in practice it is possible to use efficient algorithms for finding lassos in Büchi automata on-the-fly, such as for instance the Nested Depth First Search algorithm [16].

3.3.1 The Running Example: The Termination Check

Checking (non-)termination of the abstract program in Fig. 4 is done by checking emptiness of the intersection between the abstraction and the complement of the Büchi automaton recognizing the language $(\langle _, =_x \rangle^* \langle _, >_x \rangle)^\omega$ (cf. Fig. 6 (b)). In the case of our running example, the intersection is not empty, counterexamples being $\langle 1, >_x \rangle (\langle 2, =_x \rangle \langle 3, \bowtie_x \rangle)^\omega$ and $\langle 1, =_x \rangle (\langle 2, =_x \rangle \langle 3, \bowtie_x \rangle)^\omega$, which both correspond to the infinite execution of the first loop, i.e., lines $1(23)^\omega$.

3.4 Counterexample-based Abstraction Refinement

If a Büchi automaton A is an abstraction of a program $P = \langle \mathfrak{I}, L, l_0, \Rightarrow \rangle$ (cf. Def. 12), $D_0 \in D$ is a set of initial values, and $\sigma\lambda^\omega \in \mathcal{L}(A)$ is a lasso, where $\sigma = \langle l_0, \rho_0 \rangle \ldots \langle l_{|\sigma|-1}, \rho_{|\sigma|-1} \rangle$ and $\lambda = \langle l_{|\sigma|}, \rho_{|\sigma|} \rangle \ldots \langle l_{|\sigma|+|\lambda|-1}, \rho_{|\sigma|+|\lambda|-1} \rangle$, the *spuriousness problem* asks whether P has an execution along the infinite path $(l_0 \ldots l_{|\sigma|-1})(l_{|\sigma|} \ldots l_{|\sigma|+|\lambda|-1})^\omega$ starting with some value $d_0 \in D_0$. Notice that each pair of control

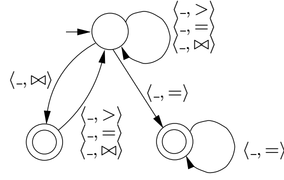


Fig. 6.

locations corresponds to exactly one program instruction, therefore the sequence of instructions corresponding to the infinite unfolding of the lasso is uniquely identified by the sequences of locations $l_0, \ldots, l_{|\sigma|-1}$ and $l_{|\sigma|}, \ldots, l_{|\sigma|+|\lambda|-1}$.

Algorithms for solving the spuriousness problem exist, depending on the structure of the domain D and on the semantics of the program instructions. Details regarding spuriousness problems for integer and tree-manipulating lassos can be found in [20].

Given a lasso $\sigma\lambda^\omega \in \mathcal{L}(A)$, the refinement builds another abstraction A' of P such that $\sigma\lambda^\omega \notin \mathcal{L}(A')$. Having established that the program path $(l_0 \ldots l_{|\sigma|-1})(l_{|\sigma|} \ldots l_{|\sigma|+|\lambda|-1})^\omega$, corresponding to $\sigma\lambda^\omega$, cannot be executed for any value from the initial set, allows us to refine by excluding potentially more spurious witnesses, than just $\sigma\lambda^\omega$. Let C be the Büchi automaton recognizing the language $L_\sigma L_\lambda^\omega$, where:

$$\begin{array}{rcl} L_\sigma & = & \{\langle l_0, \rho_0\rangle \ldots \langle l_{|\sigma|-1}, \rho_{|\sigma|-1}\rangle \mid \rho_i \in \{>, \bowtie, =\}^n,\ 0 \le i < |\sigma|\} \\ L_\lambda & = & \{\langle l_{|\sigma|}, \rho_0\rangle \ldots \langle l_{|\sigma|+|\lambda|-1}, \rho_{|\lambda|-1}\rangle \mid \rho_i \in \{>, \bowtie, =\}^n,\ 0 \le i < |\lambda|\} \end{array}$$

Then $A' = A \otimes \overline{C}$, where $\overline{C}$ is the complement of C, is the refinement of A that excludes the lasso $\sigma\lambda^\omega$, and all other lassos corresponding to the program path $(l_0 \ldots l_{|\sigma|-1})(l_{|\sigma|} \ldots l_{|\sigma|+|\lambda|-1})^\omega$.

On the down side, complementation of Büchi automata is, in general, a costly operation: the size of the complement is bounded by $2^{\mathcal{O}(n \log n)}$, where n is the size of the automaton [29]. However, the particular structure of the automata considered here comes to rescue. It can be seen that $L_\sigma L_\lambda^\omega$ can be recognized by a WDBA, hence complementation is done in constant time, and $\|A'\| \le 3 \cdot (|\sigma| + |\lambda| + 1) \cdot \|A\|$.

Lemma 3. Let A be a Büchi automaton that is an abstraction of a program P, and $\sigma\lambda^\omega \in \mathcal{L}(A)$ be a spurious counterexample. Then the Büchi automaton recognizing the language $\mathcal{L}(A) \setminus L_\sigma \cdot L_\lambda^\omega$ is an abstraction of P.

Proof. Let $\langle l_0, d_0\rangle\langle l_1, d_1\rangle\langle l_2, d_2\rangle \ldots$ be an infinite run of P. Since A is an abstraction of P, by Definition 12, there exists an infinite word $\langle l_0, \rho_0\rangle\langle l_1, \rho_1\rangle\langle l_2, \rho_2\rangle \ldots \in \mathcal{L}(A)$ such that $d_i\ [\rho_i]\ d_{i+1}$, for all $i \ge 0$. Since $\sigma\lambda^\omega$ is a spurious lasso, then the sequence of control locations $l_0, l_1, l_2, \ldots$ cannot correspond to the sequence of first positions from $\sigma\lambda^\omega$. Hence $\langle l_0, \rho_0\rangle\langle l_1, \rho_1\rangle\langle l_2, \rho_2\rangle \ldots \notin L_\sigma \cdot L_\lambda^\omega$. Since the infinite run $\langle l_0, d_0\rangle\langle l_1, d_1\rangle\langle l_2, d_2\rangle \ldots$ was chosen arbitrarily, it follows that the automaton recognizing $\mathcal{L}(A) \setminus L_\sigma \cdot L_\lambda^\omega$ is an abstraction of P. □

3.4.1 The Running Example: The Refinement

In the case of our running example, we discovered the lasso $1(23)^\omega$. This execution is found to be spurious by a specialized procedure that checks whether a given program lasso can be fired infinitely often. For this purpose, the method given in [20] could be used here, since the loop $1(23)^\omega$ does not change the structure of the tree. The refinement of the abstraction consists in eliminating the infinite path $1(23)^\omega$ from the model. This is done by intersecting the model with the automaton that recognizes the *complement* of the language $\{\langle 1, >_x\rangle, \langle 1, =_x\rangle\}(\langle 2, =_x\rangle\langle 3, \bowtie_x\rangle)^\omega$, which corresponds to the program path $1(23)^\omega$. The result of this intersection is shown Fig. 7. Notice that, in this case, the refinement does not increase the size of the abstraction. Since now, only 4 and 5 are accepting states, another intersection with the automaton in Fig. 6 will be empty and hence the refined abstraction does not have further non-terminating executions, proving thus termination of the original program.

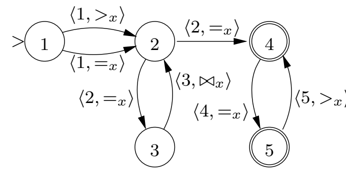


Fig. 7.

3.5 Refinement Heuristics

The refinement technique, based on the closure of ω-regular languages under intersection and complement, can be generalized to exclude an entire family of counterexamples, described as an ω-regular language, all at once. In the following we provide such a refinement heuristics.

The main difficulty here is to generalize from lasso-shaped counterexamples to more complex sets of counterexamples. In the following, we provide two refinement heuristics that eliminate entire families of counterexamples, at once. We assume in the following that we are given an invariant $\iota : L \to 2^D$ of the program.

Infeasible Elementary Loop Refinement Let $\sigma\lambda^\omega$ be a lasso representing a spurious counterexample. To apply this heuristic method, we suppose that there exists an upper bound $B \in \mathbb{N}, B > 0$, on the number of times λ can be iterated, starting with any data value from $\iota(l_{|\sigma|})$, where $\iota(l_{|\sigma|})$ denotes the program invariant on location reachable in the program by the sequence of instructions σ from the initial location. The existence of such a bound can be discovered by, e.g., a symbolic execution of the loop. In case of such a bound B exists, let C be a WDBA such

that $\mathcal{L}(C) = \Sigma^*_{(P,D)} \cdot L^B_\lambda \cdot \Sigma^\omega_{(P,D)}$. Then the Büchi automaton $A \otimes \overline{C}$ is an abstraction of P, which excludes the spurious trace $\sigma\lambda^\omega$, as shown by the following Lemma:

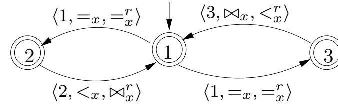
Lemma 4. Let $P = \langle \mathfrak{I}, L, l_0, \Rightarrow \rangle$ be a program, $\iota : L \to 2^D$ be an invariant of P, A be an abstraction of P, and $\lambda \in \Sigma^*_{(P,D)}$ be a lasso starting and ending with $\ell \in L$. If there exists $B > 0$ such that λ^B is infeasible, for any $d \in \iota(\ell)$, then the Büchi automaton recognizing the language $\mathcal{L}(A) \setminus \Sigma^*_{(P,D)} \cdot L^B_\lambda \cdot \Sigma^\omega_{(P,D)}$ is an abstraction of P.

Proof. Let $\langle l_0, d_0 \rangle \langle l_1, d_1 \rangle \langle l_2, d_2 \rangle \dots$ be an infinite run of P. Since A is an abstraction of P, by Definition 12, there exists an infinite word $\langle l_0, \rho_0 \rangle \langle l_1, \rho_1 \rangle \langle l_2, \rho_2 \rangle \dots \in \mathcal{L}(A)$ such that $d_i \ [\rho_i] \ d_{i+1}$, for all $i \geq 0$. Since λ^B cannot be fired for any data element $d \in \iota(\ell)$, then the infinite sequence $l_0, l_1, l_2, \dots$ may not contain the subsequence $(l_{|\sigma|} l_{|\sigma|+1} \dots l_{|\sigma|+|\lambda|-1})^B$, corresponding to the first positions from λ^B. Consequently $\langle l_0, \rho_0 \rangle \langle l_1, \rho_1 \rangle \langle l_2, \rho_2 \rangle \dots \notin \Sigma^*_{(P,D)} \cdot L^B_\lambda \cdot \Sigma^\omega_{(P,D)}$. Since the infinite run $\langle l_0, d_0 \rangle \langle l_1, d_1 \rangle \langle l_2, d_2 \rangle \dots$ was chosen arbitrarily, it follows that the automaton recognizing $\mathcal{L}(A) \setminus \Sigma^*_{(P,D)} \cdot L^B_\lambda \cdot \Sigma^\omega_{(P,D)}$ is an abstraction of P. $\square$

As an example, let us consider the following program:

```
1 while (x != null)
        if (root.data)
2          x := x.left;
3       else x := x.up;
```

Here we assume that the root of the tree has a boolean `data` field, which is used by the loop to determine the direction (i.e., `left`, `up`) of the variable x. The initial abstraction for this program is:

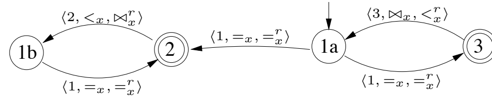


This abstraction uses two well-founded relations, $\leq_x$ defined in Section 3, and $\leq^r_x$ which is a stronger version of the reversed relation $(\geq_x)$: $t_1 \leq^r_x t_2$ iff $dom(t_1) = dom(t_2)$[5] and $t_2 \leq_x t_1$.

Then a spurious lasso is $\sigma\lambda^\omega$, where σ is the empty word, and λ is $\langle 1, =_x, =^r_x \rangle \langle 2, <_x, \bowtie^r_x \rangle \langle 1, =_x, =^r_x \rangle \langle 3, \bowtie_x, <^r_x \rangle$, which corresponds to the program path 1213. This path is infeasible for any tree, and any position of x, therefore we refine the abstraction by eliminating the language $\Sigma^*_{(P,D)} L_\lambda \Sigma^\omega_{(P,D)}$. The refined automaton is given bellow:

The intersection of the refined automaton with the automaton $B = B_1 \otimes B_2$ is empty, where B_i, $i = 1, 2$ are the automata from Fig 5, hence, by Definition 12, we can conclude that the program terminates.

[5] For a tree t, $dom(t)$ is the set of positions in the tree.



This heuristic was used to prove termination of the *Red-black delete* algorithm, reported in Section 5. Interestingly, this algorithm could not be proved to terminate using standard refinement (cf. Lemma 3).

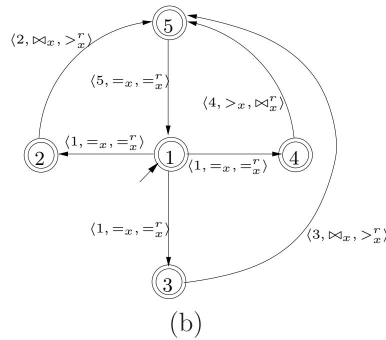
Infeasible Nested Loops Refinement Let us assume that the location $l_{|\sigma|}$ is the source (and destination) of $k > 1$ different elementary loops in A: $\lambda_1, \ldots, \lambda_k$. Moreover, let us assume that:

1. these loops can only be fired in a given total order, denoted $\lambda_{i_1} \rhd \lambda_{i_2} \rhd \ldots \lambda_{i_k}$, for each input value in the set $D_\sigma = \{d \mid \langle l_0, d_0 \rangle \stackrel{\sigma}{\Rightarrow} \langle l_{|\sigma|}, d \rangle,\ d_0 \in D_0\}$[6].
2. the infinite word $\sigma(\lambda_{i_1} \cdot \ldots \cdot \lambda_{i_k})^\omega$ is a spurious counterexample for non-termination, i.e., the corresponding program path is infeasible for any initial value from D_0.

Under these assumptions, let C be the Büchi automaton recognizing the language $L_\sigma \cdot (L_{\lambda_1} \cup \ldots \cup L_{\lambda_k})^\omega$. The following lemma shows that $A \otimes \overline{C}$ is an abstraction of the program, that excludes the spurious lasso $\sigma\lambda^\omega$:

```
1 while (x != null)
        if (root.data == 0)
2           x := x.left;
        else if (root.data == 1)
3           x := x.right;
        else if (root.data == 2)
4           x := x.up;
5       root.data := (root.data+1) % 3;
```

(a)



(b)

Fig. 8.

Lemma 5. Let $P = \langle \Im, L, l_0, \Rightarrow \rangle$ be a program, $\iota : L \to 2^D$ be an invariant of P, A be an abstraction of P, and $\lambda_1, \ldots, \lambda_k \in \Sigma^*_{(P,D)}$ be words labeling different cycles of A starting and ending with the same location $\ell \in L$. Moreover, let $\sigma \in \Sigma^*_{(P,D)}$ be the label of a path from l_0 to ℓ in A. If there exists a total order $\lambda_{i_1} \rhd \ldots \rhd \lambda_{i_k}$ in which the cycles can be executed, for any $d \in \iota(\ell)$, and $\sigma \cdot (\lambda_{i_1} \cdot \ldots \cdot \lambda_{i_k})^\omega$ is moreover spurious, then the Büchi automaton recognizing the language $\mathcal{L}(A) \setminus L_\sigma \cdot (L_{\lambda_1} \cup \ldots \cup\ L_{\lambda_k})^\omega$ is an abstraction of P.

[6] Notice that checking the existence of such an ordering amounts to performing at most k^2 feasibility checks, for all paths of the form $\lambda_i \cdot \lambda_j,\ 1 \le i, j \le k,\ i \ne j$.

Proof. Let $\langle l_0, d_0\rangle\langle l_1, d_1\rangle\langle l_2, d_2\rangle \ldots$ be an infinite run of P. Since A is an abstraction of P, by Definition 12, there exists an infinite word $\langle l_0, \rho_0\rangle\langle l_1, \rho_1\rangle\langle l_2, \rho_2\rangle \ldots \in \mathcal{L}(A)$ such that $d_i\ [\rho_i]\ d_{i+1}$, for all $i \geq 0$. We show that $\langle l_0, \rho_0\rangle\langle l_1, \rho_1\rangle\langle l_2, \rho_2\rangle \ldots \notin L_\sigma \cdot (L_{\lambda_1} \cup \ldots \cup\ L_{\lambda_k})^\omega$. Assume by contradiction that $\langle l_0, \rho_0\rangle\langle l_1, \rho_1\rangle\langle l_2, \rho_2\rangle \ldots \in L_\sigma \cdot (L_{\lambda_1} \cup \ldots \cup\ L_{\lambda_k})^\omega$. Then the infinite sequence $l_0 l_1 l_2 \ldots$ of first positions can be decomposed into a prefix corresponding to σ, followed by an infinite succession of blocks corresponding to some $\lambda_i,\ 1 \leq i \leq k$. Since $\lambda_{i_1} \rhd \lambda_{i_2} \rhd \ldots \lambda_{i_k}$ is the only possible execution order, these blocks must respect the order $\rhd$. However this is in contradiction with the fact that $\sigma(\lambda_{i_1} \cdot \ldots \cdot \lambda_{i_k})^\omega$ is a spurious lasso. The conclusion follows. □

As an example, let us consider the program in Fig 8(a). Its initial abstraction, using the relations $\leq_x$ and $\leq_x^r$, is given in Fig. 8(b). The three nested loops are, in the unique execution order: $1251 \rhd 1351 \rhd 1451 \rhd 1251$. Moreover, the composed loop (1251)(1351)(1451) cannot be iterated infinitely often because the pointer variable x, moving twice down and once up in the tree, is bound to become null. This fact can be also detected automatically using, e.g., the technique from [20]. Therefore we can refine the initial abstraction by eliminating the language $\{\langle 1, =_x, =_x^r\rangle\langle 2, \bowtie_x, >_x^r\rangle\langle 5, =_x, =_x^r\rangle, \langle 1, =_x, =_x^r\rangle\langle 3, \bowtie_x, >_x^r\rangle\langle 5, =_x, =_x^r\rangle, \langle 1, =_x, =_x^r\rangle\langle 4, >_x, \bowtie_x^r\rangle\langle 5, =_x, =_x^r\rangle\}^\omega$. The refined abstraction is empty, hence the program has no infinite runs.

Remark In light of the fact that the universal halting problem in not r.e., in general, the abstraction-refinement loop is not guaranteed to terminate, even if the program terminates.

4 PROVING TERMINATION OF PROGRAMS WITH TREES

In this section we instantiate our termination verification framework for programs manipulating tree-like data structures. 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 field, ranging over a finite set $Data$ and three selector fields, denoted `left`, `right`, and `up`.[7] For $x, y \in PVar$ and $d \in Data$, we consider the programs over the set of instructions $\mathfrak{I}_T$ composed of the following :

- **guards** : $x ==$ `null`, $x == y$, x.`data` $== d$, and boolean combinations of the above,
- **actions** : $x =$ `null`, $x = y$, $x = y$.{`left`|`right`|`up`}, x.`data` $= d$, x.{`left` |`right`} = `new` and x.{`left`|`right`} = `null`.

[7] Generalizing to trees with another arity is straightforward.

```
0   x := root;
1   while (x!=null)
2       if (x.left!=null) and
            (x.left.data!=mark)
3           x:=x.left;
4       else if (x.right!=null) and
            (x.right.data!=mark)
5           x:=x.right;
        else
6           x.data:=marked;
7           x:=x.up;
```

Fig. 9. Depth-first tree traversal

This set of instructions covers a large class of practical tree-manipulating procedures[8]. For instance, Fig. 9 shows a depth-first tree traversal procedure, commonly used in real-life programs. In particular, here $PVar = \{x\}$ and $Data = \{marked, unmarked\}$. This program will be used as a running example in the rest of the section.

In order to use our framework for analyzing termination of programs with trees, we need to provide (1) well-founded partial orderings on the tree domain, (2) symbolic encodings for the partial orderings as well as for the program semantics and (3) a decision procedure for the spuriousness problem. The last point was tackled in our previous work [20], for lassos without destructive updates (i.e., instructions of the form $x.\texttt{left}|\texttt{right} := \texttt{new}|\texttt{null}$).

4.1 Abstracting Programs with Trees into Büchi Automata

A memory configuration is a binary tree with nodes labeled by elements of the set $\mathcal{C} = Data \times 2^{PVar} \cup \{\Box\}$, i.e., a node is either null ($\Box$) or it contains a data value and a set of pointer variables pointing to it ($\langle d, V\rangle \in D \times 2^{PVar}$). Each pointer variable can point to at most one tree node (if it is null, it does not appear in the tree). For a tree $t \in \mathcal{T}(\mathcal{C})$ and a position $p \in dom(t)$ such that $t(p) = \langle d, V\rangle$, we denote $\delta_t(p) = d$ and $\nu_t(p) = V$. For clarity reasons, the semantics of the program instructions considered is given in Figure 10. First we show that all program actions considered here can be encoded as rational tree relations.

Lemma 6. For any program instruction $i = \langle g, a\rangle \in \mathfrak{I}_T$, the tree relation $R_i = \{\langle t, t'\rangle \mid t \in g,\ t' = a(t)\}$ is rational.

[8] Handling general updates of the form $x.\texttt{left}|\texttt{right} := y$ is more problematic, since in general the result of such instructions is not a tree anymore. Up to some extent, this kind of programs can be handled by the extension described in the Section 4.3.

$$\frac{\forall p \in dom(t) : x \notin \nu(t(p))}{\langle l, t\rangle \rightarrow \langle l', t\rangle} \quad \frac{p \in dom(t) \quad x \in \nu(t(p))}{\langle l, t\rangle \rightarrow \langle l', t[p \leftarrow \langle \delta(t(p)), \nu(t(p)) \setminus \{x\}\rangle]\rangle} \; x = \mathtt{null}$$

$$\frac{\begin{array}{c} p \in dom(t) \quad y \in \nu(t(p)) \\ t(p.0) \neq \Box \quad \langle l, t\rangle \overset{x=null}{\longrightarrow} \langle l', t'\rangle \end{array}}{\langle l, t\rangle \rightarrow \langle l', t'[p.0 \leftarrow \langle \delta(t(p.0)), \nu(t(p.0)) \cup \{x\}\rangle]\rangle} \; x = y.\mathtt{left}(a)$$

$$\frac{\begin{array}{c} p \in dom(t) \quad y \in \nu(t(p)) \\ t(p.0) = \Box \quad \langle l, t\rangle \overset{x=null}{\longrightarrow} \langle l', t'\rangle \end{array}}{\langle l, t\rangle \rightarrow \langle l', t'\rangle} \quad \frac{\forall p \in dom(t) \,.\, y \notin \nu(t(p))}{\langle l, t\rangle \rightarrow Err} \; x = y.\mathtt{left}(b)$$

$$\frac{p \in dom(t) \quad x \in \nu(t(p))}{\langle l, t\rangle \rightarrow \langle l', t[p \leftarrow \langle d, \nu(t(p))\rangle]\rangle} \quad \frac{\forall p \in dom(t) \,.\, x \notin \nu(t(p))}{\langle l, t\rangle \rightarrow Err} \; x.\mathtt{data} = d$$

$$\frac{\begin{array}{cc} d \in Data & p \in dom(t) \\ x \in \nu(t(p)) & t(p.0) = \Box \end{array}}{\langle l, t\rangle \rightarrow \langle l', t[p.0 \leftarrow \langle d, \emptyset\rangle, p.0.0 \leftarrow \Box, p.0.1 \leftarrow \Box]\rangle} \quad \frac{\begin{array}{l} (\forall p \in dom(t) \,.\, x \notin \nu(t(p))) \vee \\ (p \in dom(t) \;\wedge\; x \in \nu(t(p)) \;\wedge\; t(p.0) \neq \Box) \end{array}}{\langle l, t\rangle \rightarrow Err} \; x.\mathtt{left} = new$$

$$\frac{\begin{array}{c} p \in dom(t) \quad x \in \nu(t(p)) \quad p.0 \in dom(t) \\ t(p.0.0) = t(p.0.1) = \Box \end{array}}{\langle l, t\rangle \rightarrow \langle l', t[p.0 \leftarrow \Box, p.0.0 \leftarrow \bot, p.0.1 \leftarrow \bot]\rangle} \quad \frac{\begin{array}{l} (\forall p \in dom(t) \,.\, x \notin \nu(t(p))) \vee \\ (p \in dom(t) \;\wedge\; x \in \nu(t(p)) \;\wedge \\ \;(t(p.0) = \Box \vee t(p.0.0) \neq \Box \;\vee\; t(p.0.1) \neq \Box)) \end{array}}{\langle l, t\rangle \rightarrow Err} \; x.\mathtt{left} = null$$

Fig. 10. The concrete semantics of program statements—the upper part of a rule represents a guard and the lower part an action. *Err* is equal to abnormal termination of the program (it is not possible to execute the given statement).

Proof. We will now show, how to construct a tree automaton for each guard and each statement. The automaton for the guarded action is then created as composition of the corresponding guard and the action. Both the guards and actions are tree automata $A = (Q, F, \Delta)$ over the pair alphabet $(\mathcal{C} \cup \{\Box, \bot\}) \times (\mathcal{C} \cup \{\Box, \bot\})$. We use the following subsets of the alphabet $\mathcal{C}$ within the proof:

- $N_V = \{\langle d, X\rangle \in C | \forall x \in V.x \notin X\}$
- $P_V = \{\langle d, X\rangle \in C | \forall x \in V.x \in X\}$

guards

- $x ==$ `null`: $A = (\{q_1\}, \{q_1\}, \Delta)$ with $\Delta = \{$
  - $\langle \Box, \Box\rangle \rightarrow q_1$
  - $\forall p \in N_{\{x\}} \langle p, p\rangle(q_1, q_1) \rightarrow q_1\}$
- $x == y$: $A = (\{q_1\}, \{q_1\}, \Delta)$ with $\Delta = \{$
  - $\langle \Box, \Box\rangle \rightarrow q_1$
  - $\forall p \in N_{\{x,y\}} \langle p, p\rangle(q_1, q_1) \rightarrow q_1$
  - $\forall p \in P_{\{x,y\}} \langle p, p\rangle(q_1, q_1) \rightarrow q_1\}$
- $x.$`data` $== d$: $A = (\{q_1, q_2\}, \{q_2\}, \Delta)$ with $\Delta = \{$

- $\langle \Box, \Box \rangle \to q_1$
- $\forall p \in N_{\{x\}}.\langle p, p \rangle (q_1, q_1) \to q_1\}$
- $\forall \langle d, X \rangle \in P_{\{x\}}.\langle \langle d, X \rangle, \langle d, X \rangle \rangle (q_1, q_1) \to q_2$
- $\forall p \in N_{\{x\}} \langle p, p \rangle (q_2, q_1) \to q_2$ and $\langle p, p \rangle (q_1, q_2) \to q_2\}$

actions

- $x = \texttt{null}$: $A = (\{q_1\}, \{q_1\}, \Delta)$ with $\Delta = \{$
  - $\langle \Box, \Box \rangle \to q_1$
  - $\forall p \in N_{\{x\}} \langle p, p \rangle (q_1, q_1) \to q_1$
  - $\forall \langle d, X \rangle \in P_{\{x\}} \langle \langle d, X \rangle, \langle d, X \setminus \{x\} \rangle \rangle (q_1, q_1) \to q_1\}$
- $y = x$: $A = (\{q_1\}, \{q_1\}, \Delta)$ with $\Delta = \{$
  - $\langle \Box, \Box \rangle \to q_1$
  - $\forall p \in N_{\{x,y\}}.\langle p, p \rangle (q_1, q_1) \to q_1$
  - $\forall \langle d, X \rangle \in P_{\{x\}}.\langle \langle d, X \rangle, \langle d, X \cup \{y\} \rangle \rangle (q_1, q_1) \to q_1$
  - $\forall \langle d, X \rangle \in \{P_{\{y\}} \setminus P_{\{x\}}\}.\langle \langle d, X \rangle, \langle d, X \setminus \{y\} \rangle \rangle (q_1, q_1) \to q_1\}$
- $y = x.\texttt{left}$, $A = (\{q_1, q_y, q_2\}, \{q_2\}, \Delta)$ with $\Delta = \{$
  - $\langle \Box, \Box \rangle \to q_1$
  - $\langle \Box, \Box \rangle \to q_y$
  - $\forall \langle d, X \rangle \in N_{\{x\}}.\langle \langle d, X \rangle, \langle d, X \cup \{y\} \rangle \rangle (q_1, q_1) \to q_y$
  - $\forall \langle d, X \rangle \in N_{\{x\}}.\langle \langle d, X \rangle, \langle d, X \setminus \{y\} \rangle \rangle (q_1, q_1) \to q_1$
  - $\forall \langle d, X \rangle \in P_{\{x\}}.\langle \langle d, X \rangle, \langle d, X \setminus \{y\} \rangle \rangle (q_y, q_1) \to q_2$
  - $\forall \langle d, X \rangle \in N_{\{x\}}.\langle \langle d, X \rangle, \langle d, X \setminus \{y\} \rangle \rangle (q_2, q_1) \to q_2$
  - $\forall \langle d, X \rangle \in N_{\{x\}}.\langle \langle d, X \rangle, \langle d, X \setminus \{y\} \rangle \rangle (q_1, q_2) \to q_2\}$
- $x.\texttt{data} = d$: $A = (\{q_1, q_2\}, \{q_2\}, \Delta)$ with $\Delta = \{$
  - $\langle \Box, \Box \rangle \to q_1$
  - $\forall p \in N_{\{x\}}.\langle p, p \rangle (q_1, q_1) \to q_1$
  - $\forall \langle d_{orig}, X \rangle \in P_{\{x\}}.\langle \langle d_{orig}, X \rangle, \langle d, X \rangle \rangle (q_1, q_1) \to q_2$
  - $\forall p \in N_{\{x\}}.\langle p, p \rangle (q_2, q_1) \to q_2$ and $\langle p, p \rangle (q_1, q_2) \to q_2\}$
- $x.\texttt{left} = \texttt{new}$: $A = (\{q_1, q_{new}, q_\perp, q_2\}, \{q_2\}, \Delta)$ with $\Delta = \{$
  - $\langle \Box, \Box \rangle \to q_1$
  - $\langle \perp, \Box \rangle \to q_\perp$
  - $\langle \Box, \langle d_{init}, \emptyset \rangle \rangle (q_\perp, q_\perp) \to q_{new}$
  - $\forall p \in N_{\{x\}}.\langle p, p \rangle (q_1, q_1) \to q_1$
  - $\forall p \in P_{\{x\}} \langle p, p \rangle (q_{new}, q_1) \to q_2$
  - $\forall p \in N_{\{x\}}.\langle p, p \rangle (q_2, q_1) \to q_2$ and $\langle p, p \rangle (q_1, q_2) \to q_2\}$

- $x.\texttt{left} = \texttt{null}$. $A = (\{q_1, q_{del}, q_\perp, q_2\}, \{q_2\}, \Delta)$ with $\Delta = \{$
  - $\langle \Box, \Box \rangle \to q_1$
  - $\langle \Box, \perp \rangle \to q_\perp$
  - $\forall p \in N_{\{x\}} \langle p, \Box \rangle (q_\perp, q_\perp) \to q_{del}$
  - $\forall p \in N_{\{x\}}.\langle p, p \rangle (q_1, q_1) \to q_1$
  - $\forall p \in P_{\{x\}} \langle p, p \rangle (q_{del}, q_1) \to q_2$
  - $\forall p \in N_{\{x\}}.\langle p, p \rangle (q_2, q_1) \to q_2$ and $\langle p, p \rangle (q_1, q_2) \to q_2 \}$

Instructions $y = x.\texttt{right}$ and $y = x.\texttt{up}$ are similar to the $y = x.\texttt{left}$, the instruction $x.\texttt{right} = \texttt{new}$ to the $x.\texttt{left} = \texttt{new}$, and the instruction $x.\texttt{right} = \texttt{null}$ to the $x.\texttt{left} = \texttt{null}$, □

In order to abstract programs with trees as Büchi automata (cf. Def. 12), we must introduce the well-founded partial orders on the working domain—i.e., trees. These well founded orders are captured by Def. 14, and the working domain will be $D_T = \langle \mathcal{T}(\mathcal{C}), \{\preceq_x, \preceq_x^r\}_{x \in PVar}, \{\preceq_d, \preceq_d^r\}_{d \in Data} \rangle$.

Definition 14. (Well-founded orders on trees)

- $t_1 \preceq_x t_2$, for some $x \in PVar$ iff (i) $dom(t_1) \subseteq dom(t_2)$, and (ii) there exists positions $p_1 \in dom(t_1)$, $p_2 \in dom(t_2)$ such that $x \in \nu_{t_1}(p_1)$, $x \in \nu_{t_2}(p_2)$ and $p_1 \leq_{lex} p_2$. In other words t_1 is smaller than t_2 if all nodes in t_1 are also present in t_2 and the position of x in t_1 is lexicographically smaller than the position of x in t_2.
- $t_1 \preceq_x^r t_2$, for some $x \in PVar$ iff (i) $dom(t_1) \subseteq dom(t_2)$, and (ii) there exists positions $p_1 \in dom(t_1)$, $p_2 \in dom(t_2)$ such that $x \in \nu_{t_1}(p_1)$, $x \in \nu_{t_2}(p_2)$ and $p_1 \geq_{lex} p_2$. In other words, t_1 is smaller than t_2 if all nodes in t_1 are also present in t_2 and the position of x in t_1 is lexicographically bigger than the position of x in t_2.
- $t_1 \preceq_d t_2$, for some $d \in Data$ iff for any position $p \in dom(t_1)$ such that $\delta_{t_1}(p) = d$ we have $p \in dom(t_2)$ and $\delta_{t_2}(p) = d$. In other words, t_1 is smaller than t_2 if the set of nodes colored with d in t_1 is a subset of the set of nodes colored with d in t_2.
- $t_1 \preceq_d^r t_2$, for some $d \in Data$ iff (i) $dom(t_1) \subseteq dom(t_2)$, and (ii) for any position $p \in dom(t_2)$ such that $\delta_{t_2}(p) = d$ we have $p \in dom(t_1)$ and $\delta_{t_1}(p) = d$. In other words, t_1 is smaller than t_2 if all nodes in t_1 are also present in t_2 and the set of nodes colored with d in t_2 is a subset of the set of nodes colored with d in t_1.

Lemma 7. The relations $\preceq_x$, $\preceq_x^r$, $x \in PVar$ and $\preceq_d$,$\preceq_d^r$, $d \in Data$ are well-founded.

Proof.

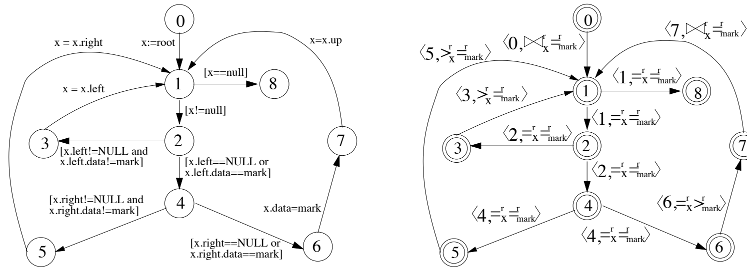


Fig. 11. The Depth-first tree traversal procedure and its initial abstraction

- Let us suppose that there exists an infinite sequence of trees $t_0 \succ_x t_1 \succ_x t_2 \succ_x \ldots$. Then according to the definition of $\succeq_x$: $\forall i \geq 0\ \exists p_i \in dom(t_i)$ such that $x \in \nu_{t_i}(p_i)$ and $p_i \geq_{lex} p_{i+1}$ and $dom(t_i) \supseteq dom(t_{i+1})$. Moreover, due to the fact that $t_{i+1} \not\preceq_x t_i$, one of the following holds for each $i \geq 0$: (i) $p_i >_{lex} p_{i+1}$ or (ii) $dom(t_i) \supset dom(t_{i+1})$. Note that at least one of the cases (i) or (ii) holds for infinite many different values of i.

  Therefore in the infinite sequence of trees $t_0 \succ_x t_1 \succ_x t_2 \succ_x \ldots$, there must exists an infinite subsequence[9] $\overline{t_0} \succ_x \overline{t_1} \succ_x \overline{t_2} \succ_x \ldots$ such that either (i) or (ii) holds for all $i \geq 0$ in this sequence.

  Let us suppose that we have the infinite sequence $\overline{t_0} \succ_x \overline{t_1} \succ_x \overline{t_2} \succ_x \ldots$ where the condition (i) holds for all $i \geq 0$. Note that for all $i \geq 0. dom(\overline{t_i}) \subseteq dom(\overline{t_0})$ and $dom(\overline{t_0})$ is finite. Therefore there exists only a *finite number of different position* $\overline{p_0}, \overline{p_1}, \ldots, \overline{p_k}$, hence the sequence $\overline{t_0} \succ_x \overline{t_1} \succ_x \overline{t_2} \succ_x \ldots$ cannot be infinite.

  Let us suppose that the condition (ii) holds for all $i \geq 0$. Then there must be an infinite sequence $dom(\overline{t_0}) \supset dom(\overline{t_1}) \supset dom(\overline{t_2}) \supset \ldots$. The $dom(\overline{t_0})$ is finite, hence the sequence cannot exists.

  The proof of well-foundedness of $\preceq^r_x$ is similar.

- Let us suppose that for some $d \in Data$ there exists an infinite sequence $t_0 \succ_d t_1 \succ_d t_2 \succ_d \ldots$. Let $S_d(t) = \{p \in dom(t) \mid \delta_t(p) = d\}$ be a set of all position labeled by a data value d in the tree t. Then according to the definition of $\preceq_d$: $S_d(t_0) \supset S_d(t_1) \supset S_d(t_2) \supset \ldots$. The $dom(t_0)$ is finite, hence the set $S_d(t_0)$ is finite also. Therefore the sequence $t_0 \succ_d t_1 \succ_d t_2 \succ_d \ldots$ must be finite.

- Let us suppose that for some $d \in Data$ there exists an infinite sequence $t_0 \succ^r_d t_1 \succ^r_d t_2 \succ^r_d \ldots$. Let $S_d(t) = \{p \in dom(t) \mid \delta_t(p) = d\}$ be a set of all position labeled by a data value d in the tree t. Then according to the definition of $\preceq^r_d$: $S_d(t_0) \subseteq S_d(t_1) \subseteq S_d(t_2) \subseteq \ldots$ and $dom(t_0) \supseteq dom(t_1) \supseteq dom(t_2) \supseteq \ldots$. Moreover, due to the fact that $\forall i \geq 0.\ t_{i+1} \not\preceq_x t_i$, one of the following situations holds for each value of i: (i)$S_d(t_i) \subset S_d(t_{i+1})$ or (ii) $dom(t_i) \supset dom(t_{i+1})$. Therefore in

[9] An infinite sequence $a_0, a_1, a_2, \ldots$ is a subsequence of the infinite sequence $b_0, b_1, b_2, \ldots$ iff exists a mapping $\sigma : \mathbb{N} \to \mathbb{N}$ such that $\forall i \geq 0.\ a_i = b_{\sigma(i)}$ and $i < j \Rightarrow \sigma(i) < \sigma(j)$.

the infinite sequence of trees $t_0 \succ_d^r t_1 \succ_d^r t_2 \succ_d^r \ldots$, there must exists an infinite subsequence $\overline{t_0} \succ_d^r \overline{t_1} \succ_d^r \overline{t_2} \succ_d^r \ldots$ such that either (i) or (ii) holds for all $i \geq 0$ in this sequence.

(i) cannot be true in the whole infinite sequence, because $\forall i > 0.\ S_i \subseteq dom(t_0)$ and $dom(t_0)$ is finite. (ii) cannot be true, because $dom(t_0)$ is finite. Hence the sequence $t_0 \succ_d^r t_1 \succ_d^r t_2 \succ_d^r \ldots$ must be finite.

$\square$

The Lemma 7 implies that the the Assumption 1 is valid for the working domain $D_T = \langle \mathcal{T}(\mathcal{C}), \{\preceq_x, \preceq_x^r\}_{x \in PVar}, \{\preceq_d, \preceq_d^r\}_{d \in Data} \rangle$, and hence the whole termination analysis framework presented in the section 3 can be employed.

The choice of relations for the time being is ad-hoc. In practice these relations are sufficient for proving termination of an important class of programs handling trees.

Lemma 8. The relations $\preceq_x$, $\preceq_x^r$, $x \in PVar$ and $\preceq_d$,$\preceq_d^r$, $d \in Data$ are rational.

Proof. By the construction of tree automata $A = (Q, F, \Delta)$ over the pair alphabet $(\mathcal{C} \cup \{\square, \perp\}) \times (\mathcal{C} \cup \{\square, \perp\})$ for each of the proposed relations.

- $t_1 \preceq_x t_2$: $A = (\{q_1, q_L, q_R, q_{acc}\}, \{q_{acc}\}, \Delta)$ with $\Delta = \{$
  - $\langle \square, \square \rangle \to q_1$
  - $\langle \perp, \square \rangle \to q_1$
  - $\forall p_1, p_2 \in N_{\{x\}} \cup \{\square, \perp\}. \langle p_1, p_2 \rangle (q_1, q_1) \to q_1$
  - $\forall p_1, p_2 \in N_{\{x\}} \cup \{\square, \perp\}. \langle p_1, p_2 \rangle (q_L, q_1) \to q_L$
  - $\forall p_1, p_2 \in N_{\{x\}} \cup \{\square, \perp\}. \langle p_1, p_2 \rangle (q_1, q_L) \to q_L$
  - $\forall p_1, p_2 \in N_{\{x\}} \cup \{\square, \perp\}. \langle p_1, p_2 \rangle (q_R, q_1) \to q_R$
  - $\forall p_1, p_2 \in N_{\{x\}} \cup \{\square, \perp\}. \langle p_1, p_2 \rangle (q_1, q_R) \to q_R$
  - $\forall p_x \in P_{\{x\}} \forall p \in N_{\{x\}} \cup \{\square, \perp\}. \langle p_x, p \rangle (q_1, q_1) \to q_L$
  - $\forall p_x \in P_{\{x\}} \forall p \in N_{\{x\}} \cup \{\square, \perp\}. \langle p, p_x \rangle (q_1, q_1) \to q_R$
  - $\forall p_1, p_2 \in P_{\{x\}}. \langle p_1, p_2 \rangle (q_1, q_1) \to q_{acc}$
  - $\forall p_x \in P_{\{x\}} \forall p \in N_{\{x\}}. \langle p_x, p \rangle (q_R, q_1) \to q_{acc}$
  - $\forall p_x \in P_{\{x\}} \forall p \in N_{\{x\}}. \langle p_x, p \rangle (q_1, q_R) \to q_{acc}$
  - $\forall p_1, p_2 \in N_{\{x\}}. \langle p_1, p_2 \rangle (q_L, q_R) \to q_{acc}$
  - $\forall p_1, p_2 \in N_{\{x\}}. \langle p_1, p_2 \rangle (q_{acc}, q_1) \to q_{acc}$
  - $\forall p_1, p_2 \in N_{\{x\}}. \langle p_1, p_2 \rangle (q_1, q_{acc}) \to q_{acc}$
- $t_1 \preceq_x^r t_2$: $A = (\{q_1, q_L, q_R, q_{acc}\}, \{q_{acc}\}, \Delta)$ with $\Delta = \{$
  - $\langle \square, \square \rangle \to q_1$
  - $\langle \perp, \square \rangle \to q_1$
  - $\forall p_1, p_2 \in N_{\{x\}} \cup \{\square, \perp\}. \langle p_1, p_2 \rangle (q_1, q_1) \to q_1$
  - $\forall p_1, p_2 \in N_{\{x\}} \cup \{\square, \perp\}. \langle p_1, p_2 \rangle (q_L, q_1) \to q_L$

- $\forall p_1, p_2 \in N_{\{x\}} \cup \{\Box, \bot\}. \langle p_1, p_2 \rangle (q_1, q_L) \to q_L$
- $\forall p_1, p_2 \in N_{\{x\}} \cup \{\Box, \bot\}. \langle p_1, p_2 \rangle (q_R, q_1) \to q_R$
- $\forall p_1, p_2 \in N_{\{x\}} \cup \{\Box, \bot\}. \langle p_1, p_2 \rangle (q_1, q_R) \to q_R$
- $\forall p_x \in P_{\{x\}} \forall p \in N_{\{x\}} \cup \{\Box, \bot\}. \langle p_x, p \rangle (q_1, q_1) \to q_L$
- $\forall p_x \in P_{\{x\}} \forall p \in N_{\{x\}} \cup \{\Box, \bot\}. \langle p, p_x \rangle (q_1, q_1) \to q_R$
- $\forall p_1, p_2 \in P_{\{x\}}. \langle p_1, p_2 \rangle (q_1, q_1) \to q_{acc}$
- $\forall p_x \in P_{\{x\}} \forall p \in N_{\{x\}}. \langle p, p_x \rangle (q_L, q_1) \to q_{acc}$
- $\forall p_x \in P_{\{x\}} \forall p \in N_{\{x\}}. \langle p, p_x \rangle (q_1, q_L) \to q_{acc}$
- $\forall p_1, p_2 \in N_{\{x\}}. \langle p_1, p_2 \rangle (q_R, q_L) \to q_{acc}$
- $\forall p_1, p_2 \in N_{\{x\}}. \langle p_1, p_2 \rangle (q_{acc}, q_1) \to q_{acc}$
- $\forall p_1, p_2 \in N_{\{x\}}. \langle p_1, p_2 \rangle (q_1, q_{acc}) \to q_{acc}$

- $t_1 \preceq_d t_2$: $A = (\{q_1\}, \{q_1\}, \Delta)$ with $\Delta = \{$
  - $\langle \Box, \Box \rangle \to q_1$
  - $\langle \bot, \Box \rangle \to q_1$
  - $\langle \Box, \bot \rangle \to q_1$
  - $\forall p_1, p_2 \in \mathcal{C} \cup \{\bot\}, p_1 \neq \langle d, X \rangle$ for some $X \in 2^{PVar}$ $\langle p_1, p_2 \rangle (q_1, q_1) \to q_1$.
  - $\forall X_1, X_2 \in 2^{PVar}. \langle \langle d, X_1 \rangle, \langle d, X_2 \rangle \rangle (q_1, q_1) \to q_1 \}$

- $t_1 \preceq_d^r t_2$: $A = (\{q_1\}, \{q_1\}, \Delta)$ with $\Delta = \{$
  - $\langle \Box, \Box \rangle \to q_1$
  - $\langle \bot, \Box \rangle \to q_1$
  - $\forall p_1, p_2 \in \mathcal{C} \cup \{\bot\}, p_2 \neq \langle d, X \rangle$ for some $X \in 2^{PVar}$ $\langle p_1, p_2 \rangle (q_1, q_1) \to q_1$.
  - $\forall X_1, X_2 \in 2^{PVar}. \langle \langle d, X_1 \rangle, \langle d, X_2 \rangle \rangle (q_1, q_1) \to q_1 \}$

□

The Büchi automaton representing the initial abstraction of the depth-first tree traversal procedure is depicted in Fig. 11. To simplify the figure, we use only the orders $\preceq_x^r$ and $\preceq_{mark}^r$. Thanks to these orders, there is no potential infinite run in the abstraction.

Remark. In practice, one often obtains a more precise initial abstraction by composing several program steps into one. In cases where the program instructions induce rational tree relations, it is guaranteed that composition of two or more program steps induces also a rational tree relation.

In order to decide which transitions will be composed, one can use the concept of so-called cutpoints. Formally, given a program $P = \langle \mathfrak{I}, L, l_0, \Rightarrow \rangle$, a set of cutpoints $S \subseteq L$ is a set of control locations such that each loop in the program contains at least one location $l \in S$ [12, 1]. The set of cutpoints can be provided manually or

discovered automatically by means of some heuristics. In our heuristics we provide one cutpoint on each branch in the control flow.

4.2 Adding Tree Rotations

As a possible extension of the proposed framework, one can allow tree left- and right-rotations as program statements [15]. The effect of a left tree rotations on a node pointed by variable x is depicted in Fig. 12 (the effect of the right rotation is analogous). The concrete semantics of the rotations is depicted in Fig. 13, where $u_{left} = t_\epsilon\{\lambda \leftarrow t_{|p.1}\}\{0 \leftarrow t_{|p}\}\{0.1 \leftarrow t_{|p.1.0}\}$, and $u_{right} = t_\epsilon\{\lambda \leftarrow t_{|p.0}\}\{1 \leftarrow t_{|p}\}\{1.0 \leftarrow t_{|p.0.1}\}$.

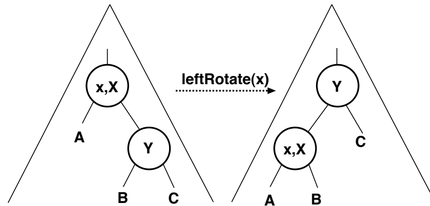


Fig. 12. The left tree rotation on the node pointed by variable x. X, and Y denoted two concrete nodes affected by the rotation and A, B, and C three subtrees.

$$\frac{p \in dom(t) \quad x \in \nu(t(p)) \quad t(p.1) \neq \square}{\langle l, t\rangle \rightarrow \langle l', t\{p \leftarrow u_{left}\}\rangle} \quad \frac{\begin{array}{c}(\forall p \in dom(t)\ .\ x \notin \nu(t(p))) \vee \\ (p \in dom(t)\ \wedge\ x \in \nu(t(p))\ \wedge\ t(p.1) = \square)\end{array}}{\langle l, t\rangle \rightarrow Err} \ \texttt{leftRotate}(x)$$

$$\frac{p \in dom(t) \quad x \in \nu(t(p)) \quad t(p.0) \neq \square}{\langle l, t\rangle \rightarrow \langle l', t\{p \leftarrow u_{right}\}\rangle} \quad \frac{\begin{array}{c}(\forall p \in dom(t)\ .\ x \notin \nu(t(p))) \vee \\ (p \in dom(t)\ \wedge\ x \in \nu(t(p))\ \wedge\ t(p.0) = \square)\end{array}}{\langle l, t\rangle \rightarrow Err} \ \texttt{rightRotate}(x)$$

Fig. 13. The concrete semantics of tree rotations. As in the case of Fig. 10, the upper part of a rule represents a guard and the lower part of a rule represents an action on a tree. *Err* is equal to abnormal termination of the program (the rotation is not possible).

Since rotations cannot be described by rational tree relations, we cannot check whether $\preceq_x, \preceq_x^r, \preceq_d$ and $\preceq_d^r$ hold, simply by intersection. However we know that rotations do not change the number of nodes in the tree, therefore we can label them a-posteriori with $=_d$,$=_d^r$, $d \in Data$, and $\bowtie_x$, $\bowtie_x^r$, $x \in PVar$, since the relative positions of the variables after the rotations are not known[10]. This extension has

[10] We could make this abstraction more precise by labeling with $=_x, =_x^r$ for all $x \in PVar$ situated *above* the rotation point – the latter condition can be checked by intersection with a rational tree language.

been used to verify termination of the *Red-black delete* and *Red-black insert* examples reported in Section 5.

4.2.1 Example: Red-black Trees—Rebalancing After Delete

Red-black trees [15] are binary search trees with the following *red-black* balanceness properties:

1. Every node carry an extra flag set eider to *red*, or *black*.
2. The root of the tree is black.
3. Every leaf node (node without a child) is black.
4. If a node is red, then both its children are black.
5. For each node in the tree, all simple paths from this node to the leaves have equal number of black nodes.

After the deletion of a node from such a tree, the balanceness property can be broken. In order to restore the red-black properties, the rebalance procedure displayed in Figure 15 is executed. In order to simplify the presentation of our termination analysis framework, we choose a set of cutpoints from the original program. The initial abstraction using these cutpoints is depicted in Figure 14. We show only the relation $\preceq_x$, which is important for the termination proof. Note, that there is no edge $14 \rightarrow 3$ and $35 \rightarrow 3$ in the abstraction, because there is no execution of the program following these edges.

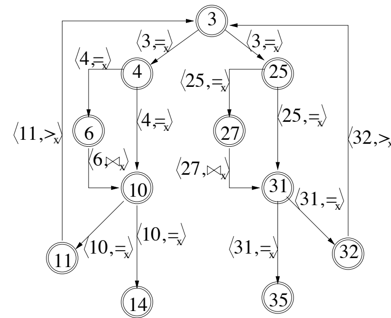


Fig. 14. Red-Black: rebalance after delete - initial abstraction on the simplified control flow

One can see in the Figure 14 that there are counterexamples of the following forms:

1. $\Sigma^*(\Sigma^*\langle 3,=_x\rangle\langle 4,=_x\rangle\langle 6,\bowtie_x\rangle\langle 10,=_x\rangle\langle 11,>_x\rangle\Sigma^*)^\omega$
2. $\Sigma^*(\Sigma^*\langle 3,=_x\rangle\langle 25,=_x\rangle\langle 27,\bowtie_x\rangle\langle 31,=_x\rangle\langle 32,>_x\rangle\Sigma^*)^\omega$

```
    procedure rb_rebalance(root)
0   while true
1       if x==root then return;
2       if x.color=RED return;
3       if x==x.parrent.left then
4           w=x.parrent.right
5           if w.color==red
6               w.color=black
7               x.parrent.color=red
8               leftrotate(x.parent)
9               w=x.parrent.right
10          if (w.left==null or w.left.color==black) and
                (w.right==null or w.right.color==black) then
11              w.color=red
12              x=x.parent
13              continue
14          if (w.right==null or w.right.color==black) then
15              w.left.color=black
16              w.color=red
17              rightrotate(w)
18              w=x.parrent.right
19          w.color=px.color
20          x.parrent.color=black
21          w.right.color=black
22          leftrotate(x.parent)
23          x=root
24      else
25          w=x.parrent.left
26          if w.color==red
27              w.color=black
28              x.parrent.color=red
29              rightrotate(x.parent)
30              w=x.parrent.left
31          if (w.left==null or w.left.color==black) and
                (w.right==null or w.right.color==black) then
32              w.color=red
33              x=x.parent
34              continue
35          if (w.left==null or w.left.color==black) then
36              w.right.color=black
37              w.color=red
38              leftrotate(w)
39              w=x.parrent.left
40          w.color=x.parrent.color
41          x.parrent.color=black
42          w.left.color=black
43          rightrotate(x.parent)
44          x=root
```

Fig. 15. Red-Black: rebalance after delete

The abstraction refinement works as follows. First, the counterexample $(\langle 3, =_x \rangle\langle 4, =_x\rangle\langle 6, \bowtie_x\rangle\langle 10, =_x\rangle\langle 11, >_x\rangle)^\omega$ is taken. There is no execution of the program following the path 3, 4, 6, 10, 11, 3 in the program, so we can apply lemma 4 and remove the whole set of counterexamples $\Sigma^*\langle 3, _\rangle\langle 4, _\rangle\langle 6, _\rangle\langle 10, _\rangle\langle 11, _\rangle(\Sigma^*)^\omega$. Note that all counterexamples of the type 1 are included. The next counterexample is $(\langle 3, =_x\rangle\langle 25, =_x\rangle\langle 27, \bowtie_x\rangle\langle 31, =_x\rangle\langle 32, >_x\rangle)^\omega$. As for the previews one, there is no execution in the program following the path 3, 25, 27, 31, 32, 3 so according to the lemma 4, we can remove the counterexamples $\Sigma^*\langle 3, _\rangle\langle 25, _\rangle\langle 27, _\rangle\langle 31, _\rangle\langle 32, _\rangle(\Sigma^*)^\omega$ (all counterexamples of the type 2 are included).

After this second refinement, there is no counterexample left in the abstraction, so we can conclude that the Red-Black rebalance procedure terminates.

4.3 From Trees to Complex Data Structure

In general, programs with pointer data types do not manipulate just lists and trees. The structures that occur at execution time can be arbitrarily oriented graphs. In this section we use the termination detection framework for programs that manipulate such complex structures. To compute program invariants, we use the approach from [6], based on an encoding of graphs as trees with extra edges. The basic idea of this encoding is that each each structure has an underlying tree (called a *backbone*) which stays unchanged during the whole computation.

As in the previous, let $PVar$ be a finite set of pointer variables, $Data$ a finite set of data values, N the maximal number of selectors allowed in a single memory node, and S is a finite set of *pointer descriptors*, which are references to regular expressions (called routing expressions) over the alphabet of directions in the tree (e.g., left, right, left-up, right-up, etc.). The backbone is a tree labeled by symbols of the alphabet $\mathcal{C} = (Data \times 2^{PVar} \times S^N) \cup \{\Box, \Diamond\}$. The arity function is defined as follows: $\#(\Box) = 0$ and $\#(c) = b$ for all $c \in \mathcal{C} \setminus \{\Box\}$, where $b > 0$ is a branching factor fixed for the concrete encoding[11]. Each node of such a tree is either *active*, *removed*, or *unused*, as follows:

- *active* nodes represent memory cells present in the memory. These nodes are labeled by symbols from the alphabet $Data \times 2^{PVar} \times S^N$.
- *removed* nodes are the nodes which were deleted from the memory. They are labeled by the symbol $\Diamond$. These nodes are kept in the data structure in order to preserve original shape of the underlying tree.
- *unused* nodes are labeled by $\Box$ and they are placed in leaves. These nodes can be changed into active ones by a `new` statement.

Pointers between the active memory nodes are represented by pointer descriptors (placed in the active nodes). A pointer descriptor corresponding to $1 \le i \le N$ determines the destination of the i-th selector field of the node. There are two special designated pointer descriptors for the *null* and *undefined* pointers. The

[11] Usually, the branching factor b is equal to the number of selectors N.

destructive updates on the data structure can be then performed simply by changing the pointer descriptors inside the tree followed by an update of the corresponding routing expression.

A set of such data structures (*trees with routing expressions*) can be represented by tree automata. This allows one to apply the framework of *abstract regular tree model checking* (ARTMC) [5] in order to compute (over-approximations of) the invariants of programs handling complex data structures. The ARTMC tool derives automatically the set of underlying trees and the corresponding routing expressions.

We apply the termination analysis framework along the same lines as in the previous. For a tree $t \in \mathcal{T}(\mathcal{C})$ and a position $p \in dom(t)$ such that $t(p) = \langle d, V, s_1, \ldots s_N \rangle$, we denote $\delta_t(p) = d$, $\nu_t(p) = V$, and $\xi_t(p)[i] = s_i$. If $t(p) = \Diamond$, $\delta_t(p) = \bot$, $\nu_t(p) = \emptyset$, and $\xi_t(p)[i] = undefined$, $1 \leq i \leq N$, i.e., all descriptors represent *undefined* pointers.

Now, we need to provide well-founded orders on trees with routing expressions. We use the fact that the tree backbone is not changed during the whole computation, hence we can easily employ the orders $\preceq_x$, $\preceq_x^r$, $\preceq_d$ and $\preceq_d^r$ defined in Def. 14. In addition, we define the following well-founded orders based on pointer descriptors values captured by the Def. 15. The working domain will be $D_T = \langle \mathcal{T}(\mathcal{C}), \{\preceq_x, \preceq_x^r\}_{x \in PVar}, \{\preceq_d, \preceq_d^r\}_{d \in Data}, \{\preceq_{1:s}, \preceq_{1:s}^r\}_{s \in S}, \ldots, \{\preceq_{N:s}, \preceq_{N:s}^r\}_{s \in S} \rangle$.

Definition 15. (Well-founded orders based on pointer descriptors)

- $t_1 \preceq_{i:s} t_2$, for some $1 \leq i \leq N$ and $s \in S$ iff for any position $p \in dom(t_1)$ such that $\xi_{t_1}(p)[i] = s$ we have $p \in dom(t_2)$ and $\xi_{t_2}(p)[i] = s$. In other words, t_1 is smaller than t_2 if the set of nodes in t_1, where the ith descriptor is set to s, is a subset of the set of nodes in t_2, where the ith descriptor is set to s.
- $t_1 \preceq_{i:s}^r t_2$, for some $1 \leq i \leq N$ and $s \in S$ iff (i) $dom(t_1) \subseteq dom(t_2)$ and (ii) for any position $p \in dom(t_2)$ such that $\xi_{t_2}(p)[i] = s$ we have $p \in dom(t_1)$ and $\xi_{t_1}(p)[i] = s$. In other words t_1 is smaller than t_2 if all nodes in t_1 are also present in t_2 (i.e., no new nodes were created) and the set of nodes in t_2, where the ith descriptor is set to s, is a subset of the set of nodes in t_1, where the ith descriptor is set to s.

Lemma 9. The relations on pointer descriptor fields $\preceq_{i:s}$ and $\preceq_{i:s}^r$ are well-founded and rational.

Proof. The pointer descriptor is syntactically a data value from a finite set S. Therefore the proofs of rationality and well-foundedness of $\preceq_{i:s}$ (resp $\preceq_{i:s}^r$) are similar to the proofs of $\preceq_d$ (resp. $\preceq_d^r$).

□

To understand the use of the relations $\preceq_{i:s}, \preceq_{i:s}^r$, consider the program from Fig. 16. This procedure traverses a binary tree in depth-first order and links all its nodes into a cyclic singly-linked list using the selector *next*. In the beginning, all *next* selector are set to null. Then the processed nodes has the selector *next*

non-null. Due to this fact, one can establish termination proof using the orders $\preceq_x^R$ and $\preceq_{next:null}$. Note that the order based on data values (as in classical depth-first traversal) cannot be used anymore to establish termination.

```
   procedure link_nodes(root)
0  x := root;
1  last:=root;
2  while (x!=null)
3     if (x.left!=null) and (x.left.next==null)
4        x:=x.left;
5     else if (x.right!=null) and (x.right.next==null)
6        x:=x.right;
      else
7        x.next:=last;
8        last:=x;
9        x:=x.up;
```

Fig. 16. Linking nodes in Depth-first order

5 IMPLEMENTATION AND EXPERIMENTAL RESULTS

We have implemented a prototype tool that uses this framework to detect termination of programs with trees and trees with extra edges. The tool was built as an extension of the ARTMC [6] verifier for safety properties (null-pointer dereferences, memory leaks, etc.). We applied our tool to several programs that manipulate:

Example	Time	N_{refs}
DLL-insert	2s	0
DLL-delete	1s	0
DLL-reverse	2s	0
Depth-first search	17s	0
Linking leaves in trees	14s	0
Deutsch-Schorr-Waite	1m 24s	0
Linking Nodes	5m 47s	0
Red-black delete	4m 54s	2
Red-black insert	29s	0

Table 1. Experimental results

- **doubly-linked lists**: *DLL-insert* (*DLL-delete*) which inserts (deletes) a node in (from) a doubly-linked list, and *DLL-reverse* which is the in-place reversal of the doubly linked list.

- **trees**: *Depth-first search* and *Deutsch-Schorr-Waite* which are tree traversals, *Red-black delete* (*insert*) which rebalances a red-black tree after the deletion (insertion) of a node.
- **tree with extra edges**: *Linking leaves* (*Linking nodes*) which insert all leaves (nodes) of a tree in a singly-linked list.

The results obtained on a Intel Core 2 PC with 2.4 GHz CPU and 2 GB RAM memory are given in the table 1. The field *time* represents the time necessary to generate invariants and build the initial abstraction. The field N_{refs} represents number of refinements. The only case in which refinement was needed is the *Red-black delete* example, which was verified using the *Infeasible Elementary Loop* refinement heuristic (Section 3.4).

6 CONCLUSIONS

We proposed a new generic termination-analysis framework. In this framework, infinite runs of a program are abstracted by Büchi automata. This abstraction is then intersected with a predefined automaton representing potentially infinite runs. In case of non-empty intersection, a counterexample is exhibited. If the counterexamples is proved to be spurious, the abstraction is refined. We instantiated the framework for programs manipulating tree-like data structures and we experimented with a prototype implementation, on top of the ARTMC invariant generator. Test cases include a number of classical algorithms that manipulate tree-like data structures.

Future work includes instantiation of the method for the class of programs handled by a tool called *Forester* [19] based on a tuples of tree automata. The encoding of complex data structures used in Forester is more flexible then the one used in ARTMC and it would allows us to handle much bigger programs, as well as more complex and tricky pointer manipulations. Using the proposed method, we would like also to tackle the termination analysis for concurrent programs. Moreover, we would like to investigate methods for automated discovery of well-founded orderings on the complex data domains as trees and graphs.

Acknowledgment. This work was supported by the Czech Science Foundation (project P103/10/0306), the Czech Ministry of Education, Youth, and Sports (project MSM 0021630528), the BUT FIT project FIT-S-12-1, the EU/Czech IT4Innovations Centre of Excellence project CZ.1.05/1.1.00/02.0070., and the French National Research Agency (project VERIDIC ANR-09-SEGI-016 (2009-2013)).

REFERENCES

[1] J. Berdine, A. Chawdhary, B. Cook, D. Distefano, and P. O'Hearn. Variance analyses from invariance analyses. In *Proc. of POPL'07*. ACM Press, 2007.

[2] V. D. Blondel and N. Portier. The Presence of a Zero in an Integer Linear Recurrent Sequence is NP–hard to Decide. In *Linear Algebra and Its Applications*, volume 351-352, pages 91–98, 2002.

[3] A. Bouajjani, M. Bozga, P. Habermehl, R. Iosif, P. Moro, and T. Vojnar. Programs with Lists are Counter Automata. In *Proc. of CAV'06*, volume 4144 of *LNCS*. Springer, 2006.

[4] A. Bouajjani, P. Habermehl, A. Rogalewicz, and T. Vojnar. ARTMC: Abstract Regular Tree Model Checking.
URL: `http://www.fit.vutbr.cz/research/groups/verifit/tools/artmc/`.

[5] A. Bouajjani, P. Habermehl, A. Rogalewicz, and T. Vojnar. Abstract Regular Tree Model Checking. *ENTCS*, 149:37–48, 2006. A preliminary version was presented at Infinity'05.

[6] A. Bouajjani, P. Habermehl, A. Rogalewicz, and T. Vojnar. Abstract Regular Tree Model Checking of Complex Dynamic Data Structures. In *Proc. of SAS'06*, volume 4134 of *LNCS*. Springer, 2006.

[7] M. Bozga, R. Iosif, and Y. Lakhnech. Flat Parametric Counter Automata. In *Proc. of ICALP'06*, volume 4052 of *LNCS*. Springer, 2006.

[8] A. R. Bradley, Z. Manna, and H. B. Sipma. Termination of Polynomial Programs. In *Proc. of VMCAI'2005*, volume 3385 of *LNCS*. Springer, 2005.

[9] A.R. Bradley and Z. Manna nad H.B. Sipma. The Polyranking Principle. In *Proc. of ICALP'05*, volume 3580 of *LNCS*. Springer, 2005.

[10] M. Braverman. Termination of Integer Linear Programs. In *Proc of CAV'06*, volume 4144 of *LNCS*. Springer, 2006.

[11] E.M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-Guided Abstraction Refinement. In *Proc. of CAV 2000*, volume 1855 of *LNCS*. Springer, 2000.

[12] L. Colussi, E. Marchiori, and M. Marchiori. On Termination of Constraint Logic Programs. In *Proc. of CP'95*, volume 976 of *LNCS*. Springer, 1995.

[13] H. Comon, M. Dauchet, R. Gilleron, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi. Tree Automata Techniques and Applications, 2005.
URL: `http://www.grappa.univ-lille3.fr/tata`.

[14] B. Cook, A. Podelski, and A. Rybalchenko. Abstraction Refinement for Termination. In *Proc. of SAS'05*, volume 3672 of *LNCS*, 2005.

[15] T.H. Cormen, C.E. Leiserson, and R.L. Rivest. *Introduction to Algorithms*. The MIT Press, 1990.

[16] C. Courcoubetis, M. Y. Vardi, P. Wolper, and M. Yannakakis. Memory Efficient Algorithms for the Verification of Temporal Properties. In *Proc. of CAV'90*, volume 531 of *LNCS*. Springer, 1991.

[17] D. Distefano, Josh Berdine, Byron Cook, and P.W. O'Hearn. Automatic Termination Proofs for Programs with Shape-shifting Heaps. In *Proc. of CAV'06*, volume 4144 of *LNCS*. Springer, 2006.

[18] A. Finkel and J. Leroux. How to Compose Presburger-accelerations: Applications to Broadcast Protocols. In *Proc. of FSTTCS'02*, volume 2556 of *LNCS*. Springer, 2002.

[19] P. Habermehl, L. Holík, A. Rogalewicz, J. Šimáček, and T. Vojnar. Forest automata for verification of heap manipulation. In *Proc. of CAV 2011*, volume 6806 of *LNCS 6806*. Springer Verlag, 2011.

[20] P. Habermehl, R. Iosif, A. Rogalewicz, and T. Vojnar. Proving Termination of Tree Manipulating Programs. In *Proc. of ATVA'07*, volume 4762 of *LNCS*. Springer, 2007.

[21] R. Iosif and A. Rogalewicz. Automata-based Termination Proofs. In *Proc. of CIAA'2005*, volume 5642 of *LNCS*. Springer, 2009.

[22] B. Khoussainov and A. Nerode. *Automata Theory and Its Applications*. Birkhauser Boston, 2001.

[23] S.K. Lahiri and S. Qadeer. Verifying Properties of Well-Founded Linked Lists. In *Proc. of POPL'06*. ACM Press, 2006.

[24] C.S. Lee, N.D. Jones, and A.M. Ben-Amram. The Size-Change Principle for Program Termination. In *Proc. of POPL'2001*. ACM Press, 2001.

[25] A. Loginov, T.W. Reps, and M. Sagiv. Automated Verification of the Deutsch-Schorr-Waite Tree-Traversal Algorithm. In *Proc. of SAS'06*, volume 4134 of *LNCS*. Springer, 2006.

[26] A. Podelski and A. Rybalchenko. Transition Invariants. In *Proc. of LICS'04*. IEEE, 2004.

[27] Grzegorz Rozenberg and Arto Salomaa, editors. *Handbook of formal languages, vol. 3: beyond words*. Springer-Verlag New York, Inc., New York, NY, USA, 1997.

[28] A. Rybalchenko. ARMC: Abstraction Refinement Model Checker. URL: `http://www.mpi-inf.mpg.de/~rybal/armc/`.

[29] M. Y. Vardi. The Büchi Complementation Saga. In *Proc. of STACS'07*, volume 4393 of *LNCS*. Springer, 2007.