# Automata-based Termination Proofs

Radu Iosif[1] and Adam Rogalewicz[2]

VERIMAG, CNRS, 2 av. de Vignate, F-38610 Gières, e-mail:iosif@imag.fr
FIT, BUT, Božetěchova 2, CZ-61266 Brno, e-mail:rogalew@fit.vutbr.cz

**Abstract.** This paper proposes a framework for detecting termination of programs handling infinite and complex data domains, such as pointer structures. In this framework, the user has to specify a finite number of well-founded relations on the data domain manipulated by these programs. Our tool then builds an initial abstraction of the program, which is checked for existence of potential infinite runs, by testing emptiness of its intersection with a predefined Büchi automaton. If the intersection is non-empty, a lasso-shaped counterexample is found. This counterexample is checked for spuriousness by a domain-specific procedure, and in case it is found to be spurious, the abstraction is refined, again by intersection with the complement of the Büchi automaton representing the lasso. We have instantiated the framework for programs handling tree-like data structures, which allowed us to prove termination of programs such as the depth-first tree traversal, the Deutsch-Schorr-Waite tree traversal, or the linking leaves algorithm.

## 1 Introduction

Proving termination is an important challenge for the existing software verification tools, requiring specific analysis techniques [18,6,21]. The basic principle underlying these methods is proving that, in every infinite computation of the program, a certain measure, pertaining to a well-founded domain, decreases infinitely often.

We propose here a new termination analysis, 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. If $\Rightarrow \subseteq D \times D$ is any transformation induced by a program statement, and $\preceq_i$, $1 \leq i \leq n$ is any partial order on $D$, i.e. we assume that the problem $\Rightarrow \cap \preceq_i \overset{?}{=} \emptyset$ 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.)
- a decision procedure for the problems $\Rightarrow \cap \preceq_i \overset{?}{=} \emptyset$, $1 \leq i \leq n$, where $\Rightarrow$ is any transition relation induced by a program statement.

- a decision procedure for the spuriousness problem: given a lasso $\sigma\lambda^\omega$, where $\sigma$ and $\lambda$ are finite sequences of program statements, does there exists an infinite execution of the program 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.

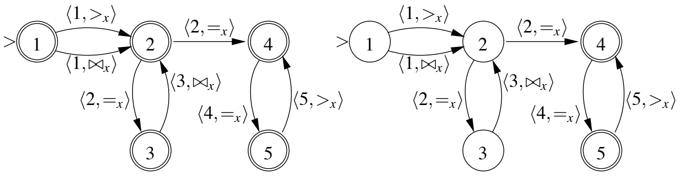
Deciding spuriousness of lassos is also a domain-dependent problem. For integer domains, techniques exist in cases where the transition relation of the loop is a difference bound matrix (DBM) [5] or an affine transformation with the finite monoid property [13]. For tree-structured data domains, we have shown decidability of spuriousness, in cases where the loop does not modify the structure of trees [14].

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 trees and more complex data structures with a tree-like backbone, e.g. doubly-linked lists, trees with parent pointers, or trees with linked leaves. 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 [8], which provide an effective method for checking emptiness of intersections between relations. A prototype tool has been implemented on top of the ARTMC [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.

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

(a)



(b) (c)

**Fig. 1.**

**Related Work.** Efficient techniques have been developed in the past for proving termination of programs with integer variables [18,6]. This remains probably the most extensively explored class of programs, concerning termination.

Recently, techniques for programs with singly-linked lists have been developed in [2,12,16]. 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 [14]. Unlike the works on singly-linked lists from [2,12], where refinement (of the counter model) is typically not needed, in [14] we considered a basic form of counterexample-driven refinement.

Abstraction refinement for termination has been first considered in [9], where the refinement consists in 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 [9] can be also applied to programs handling pointer structures.

Several ideas in this paper can be also found elsewhere. Namely, (1) using Büchi automata to encode the non-termination condition of the program was introduced by [21], and (2) proving termination for programs handling tree-like data structures was also considered in [14]. On one hand, the *size-change* termination approach from [21] does not typically come with a refinement procedure. On the other hand, the method presented here is more general and its refinement schema is more efficient then the one presented in [14]. In particular, the *Red-Black Delete* example (presented in Section 4) could not be shown to terminate using the refinement method from [14].

Automated checking of termination of programs manipulating trees has been also considered in [17], where the Deutsch-Schorr-Waite tree traversal algorithm was proved to terminate using a manually created progress monitor. In our approach, this example could be verified using the common well-founded relations on trees.

## 2 The Termination Analysis Framework

We first explain the approach informally, with the aid of an example. Let us consider the program in Fig. 1 (a), 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 reach the `root` and then become null.

Let us suppose that the only well-founded ordering considered is the following: for any two trees $t_1$ and $t_2$, we have $t_1 \geq_x t_2$ if and only if the position of $x$ in $t_2$ is a prefix of the position of $x$ in $t_1$. Then we build the abstraction of the program given in Fig. 1 (b), where $=_x$ holds if both $\leq_x$ and $\geq_x$ hold, and $\bowtie_x$ stands for $\not\geq_x$.

The states in the abstract model correspond to line numbers in the original program, and every state is considered to be accepting, initially. Checking non-termination of the abstract model amounts to checking the existence of an infinite run that *does not* have a suffix of the form $(=_x^* >_x)^\omega$, for otherwise, the well-foundedness of $\geq_x$ would prevent this execution from occurring in reality. Checking non-termination 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. 2). For technical reasons that will become clear in the sequel we label the edges of the automaton with the identifier of the source states, which correspond to program lines. In our case, the intersection is not empty, counterexamples being $\langle 1, >_x\rangle(\langle 2, =_x\rangle\langle 3, \bowtie_x\rangle)^\omega$ and $\langle 1, \bowtie_x\rangle(\langle 2, =_x\rangle\langle 3, \bowtie_x\rangle)^\omega$, which both correspond to the infinite execution of the first loop, i.e. lines $1(23)^\omega$.

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 [14] could be used here. 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 $\{\langle 1, \geq_x\rangle, \langle 1, \bowtie_x\rangle\}(\langle 2, =_x\rangle\langle 3, \bowtie_x\rangle)^\omega$, which corresponds to the program path $1(23)^\omega$. The result of this intersection is shown Fig. 1 (c). 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. 2 will establish that the refined abstraction does not have further non-terminating executions, proving thus termination of the original program.

## 2.1 Büchi Automata

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

A *Büchi automaton* (BA) over $\Sigma$ 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*.

A run of $A$ over an infinite word $a_0 a_1 a_2 \ldots \in \Sigma^\omega$ is an infinite sequence of states $s_0 s_1 s_2 \ldots$ such that $s_0 \in I$ and for all $i \geq 0$ we have $s_i \xrightarrow{a_i} s_{i+1}$. A run $\pi$ 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 $\mathcal{L}(A) \cap \mathcal{L}(B)$. If $\|A\|$ denotes the number of states (size) of $A$, it can be shown that $\|A \otimes B\| \leq 3 \cdot \|A\| \cdot \|B\|$.

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* (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 $A$.

### 2.2 Programs and Abstractions

In this section we introduce a model for programs handling data from a possibly infinite domain $D$, and define program abstractions as Büchi automata. Let $\mathfrak{I}$ be a finite set of *instructions* over a data domain $\langle D, \preceq_1, \ldots, \preceq_n\rangle$, where $\preceq_i \subseteq D \times D$ is a partial order, for $1 \leq i \leq n$. An instruction $i \in \mathfrak{I}$ is a pair $\langle g, a\rangle$ where $g \subseteq D$ is called the *guard* and $a : D \rightarrow D$ is called the *action*. An unspecified guard is assumed to be the entire domain.

A program over $\mathfrak{I}$ 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 $a_1 = a_2$.

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 $l_i \overset{g:a}{\Longrightarrow} l_{i+1}$ in the program, such that $d_i \in g$ and $d_{i+1} = a(d_i)$.

Let $D_0 \subseteq D$ be a set of initial data values. Then a configuration $\langle l, d\rangle$ is said to be *reachable* if there exists $d_0 \in D_0$, and the program has an execution from $\langle l_0, d_0\rangle$ to $\langle l, d\rangle$. 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 $d \in \iota(l)$.

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 : $d\ [\langle r_1, \ldots, r_n\rangle]\ d'$ if and only if, for all $1 \leq i \leq n$: (i) $d \succ_i d'$ iff $r_i$ is $>$, (ii) $d \not\succeq_i d'$ iff $r_i$ is $\bowtie$, and (iii) $d \approx_i d'$ iff $r_i$ is $=$.

**Definition 1.** *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.

### 2.3 Building Abstractions Automatically

A first question is how to build abstractions of programs effectively. We propose a method that performs under the assumption that program instructions, as well as the relations of the working domain can be symbolically represented by structures that are closed under projection, intersection and complement, and which, moreover, have a decidable emptiness problem.

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 :

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

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 $\rho$ 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. For reasons related to abstraction refinement, that will be made clear in the following, the transition in the Büchi automaton $A_P^\iota$ is also labeled with the source program location $l$. As an example, Fig. 1 (b) gives the initial abstraction for the program in Fig. 1 (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. 1. For space reasons, all proofs have been deferred to technical report [22].

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

### 2.4 Checking Termination on Program Abstractions

In light of Def. 1, 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 not enumerable, 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:

***Hypothesis 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 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 1. 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 $\bowtie$.

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. 2. Since the above condition holds for all $1 \leq i \leq n$, we need to compute $B = \bigotimes_{i=1}^n B_i$.

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 $A \otimes B$. The following lemma proves the existence of lasso-shaped counterexamples.

**Lemma 2.** *Given a well-founded domain* $\langle D, \preceq_1, \ldots, \preceq_n\rangle$*,* $A$ *and* $B = \bigotimes_{i=1}^n B_i$ *Büchi automata over the alphabet* $\Sigma_{(P,D)}$*, if* $\mathcal{L}(A \otimes B) \neq \emptyset$ *then* $\sigma\lambda^\omega \in \mathcal{L}(A \otimes B)$ *for some* $\sigma, \lambda \in \Sigma^*_{(P,D)}$*, where* $|\sigma|, |\lambda| \leq \|A\| \cdot (n+1) \cdot 2^n$.

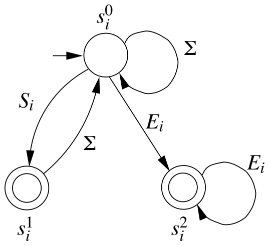


**Fig. 2.**

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 [11].

### 2.5 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. 1), $D_0 \in D$ 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 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 [14].

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:

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

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 $2^{O(n \log n)}$, where $n$ is the size of the automaton, and this is also a lower bound [20]. 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 $\|A'\| \leq 3 \cdot (|\sigma| + |\lambda| + 1) \cdot \|A\|$.

**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 $P$.*

This refinement technique, based on the closure of ω-regular languages, 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 interested reader is pointed to [22] for another refinement heuristic.

**Infeasible Elementary Loop Refinement** We suppose that there exists an upper bound $B > 0$ on the number of times $\lambda$ can be iterated, starting with any data value from $\iota(l_{|\sigma|})$. The existence of such a bound can be discovered by e.g. a symbolic execution of the loop. In case such a bound exists, the language $\Sigma^*_{(P,D)} \cdot L_\lambda^B \cdot \Sigma^\omega_{(P,D)}$ is easily shown to be recognizable by a WDBA $C$, and 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.** *Given a program $P = \langle \mathfrak{I}, L, l_0, \Rightarrow \rangle$, $\iota : L \rightarrow 2^D$ and invariant of $P$, $A$ an abstraction of $P$, and $\lambda \in \Sigma^*_{(P,D)}$ a lasso starting and ending with $\ell \in L$. If there exists $B > 0$ such that $\lambda^B$ is infeasible, for any $d \in \iota(\ell)$, then the Büchi automaton recognizing the language $\mathcal{L}(A) \setminus \Sigma^*_{(P,D)} \cdot L_\lambda^B \cdot \Sigma^\omega_{(P,D)}$ is an abstraction of $P$.*

This heuristic was used to prove termination of the *Red-black delete* algorithm, reported in Section 4. Interestingly, this algorithm could not be proved to terminate using standard refinement (cf. Lemma 3).

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

This set of instructions covers a large class of practical tree-manipulating procedures. For instance, Fig. 3 shows a depth-first tree traversal procedure, commonly used in real-life programs. In particular, here $PVar = \{x\}$ and $Data = \{marked, unmarked\}$.

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 [14], for lassos without destructive updates (i.e. instructions $x.\texttt{left}|\texttt{right} := \texttt{new}|\texttt{null}$). Recently, we have developed a spuriousness detection method that works also these destructive updates [15].

### 3.1 Trees and Tree Automata

```
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. 3.** Depth-first tree traversal

For a partial mapping $f : A \rightarrow B$ we denote $f(x) = \perp$ the fact that $f$ is undefined at some point $x \in A$. The domain of $f$ is denoted $dom(f) = \{x \in A \mid f(x) \neq \perp\}$.

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 $\#(c) = 2$ and $\#(\Box) = 0$. $\Pi$ denotes the set of tree positions $\{0,1\}^*$. Let $\varepsilon \in \Pi$ 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 $\Pi$, and for each $p \in dom(t)$:

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

When writing $t(p) = \perp$, we mean that $t$ is undefined at position $p$. We denote by $\mathcal{T}(\mathcal{C})$ the set of all trees over the alphabet $\Sigma_{\mathcal{C}}$.

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 $(\mathcal{C}_1 \cup \{\Box, \perp\}) \times (\mathcal{C}_2 \cup \{\Box, \perp\})$, where $\#(\langle \perp, \perp \rangle) = 0$, $\#(\langle \alpha, \perp \rangle) = \#(\langle \perp, \alpha \rangle) = \#(\alpha)$ if $\alpha \neq \perp$, and $\#(\langle \alpha_1, \alpha_2 \rangle) = \max(\#(\alpha_1), \#(\alpha_2))$. 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 $\mathcal{T}(\mathcal{C}_1 \times \mathcal{C}_2) = \{t \mid pr_1(t) \in \mathcal{T}(\mathcal{C}_1) \text{ and } pr_2(t) \in \mathcal{T}(\mathcal{C}_2)\}$.

A *tree automaton* [8] 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 $\Delta$ is a set of transition rules of the form:
(i) $\Box \rightarrow q$ or (ii) $c(q_1, q_2) \rightarrow q$, $c \in \mathcal{C}$.

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 $\pi$ is said to be *accepting* if and only if $\pi(\varepsilon) \in F$. 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, $\mathcal{L}(A) = R$).

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.

### 3.2 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

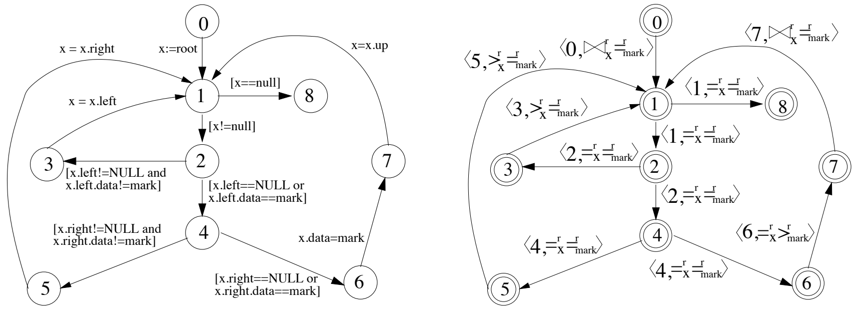

0
x = x.right
x:=root
x=x.up
x = x.left
1
[x==null]
8
[x!=null]
3
2
7
[x.left!=NULL and x.left.data!=mark]
[x.left==NULL or x.left.data==mark]
[x.right!=NULL and x.right.data!=mark]
4
x.data=mark
[x.right==NULL or x.right.data==mark]
5
6
0
$\langle 0, \bowtie^r_x =^r_{mark} \rangle$
$\langle 7, \bowtie^r_x =^r_{mark} \rangle$
$\langle 5, >^r_x =^r_{mark} \rangle$
$\langle 1, =^r_x =^r_{mark} \rangle$
$\langle 3, >^r_x =^r_{mark} \rangle$
1
8
$\langle 1, =^r_x =^r_{mark} \rangle$
$\langle 2, =^r_x =^r_{mark} \rangle$
3
2
7
$\langle 2, =^r_x =^r_{mark} \rangle$
4
$\langle 6, =^r_x >^r_{mark} \rangle$
$\langle 4, =^r_x =^r_{mark} \rangle$
$\langle 4, =^r_x =^r_{mark} \rangle$
5
6

**Fig. 4.** The Depth-first tree traversal procedure and its initial abstraction

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 $\nu_t(p) = V$. First we show that all program actions considered here can be encoded as rational tree relations[1].

**Lemma 5.** *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.*

In order to abstract programs with trees as Büchi automata (cf. Def. 1), we must introduce the well-founded partial orders on the working domain. Let $D_T = \langle \mathcal{T}(\mathcal{C}), \{\preceq_x, \preceq^r_x\}_{x \in PVar}, \{\preceq_d, \preceq^r_d\}_{d \in Data} \rangle$, where:

- $t_1 \preceq_x t_2$, for some $x \in PVar$ iff 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 $p_1 \leq_{lex} p_2$.
- $t_1 \preceq^r_x t_2$, 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 $p_1 \geq_{lex} p_2$.
- $t_1 \preceq_d t_2$, 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 $\delta_{t_2}(p) = d$.
- $t_1 \preceq^r_d t_2$, 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 $\delta_{t_1}(p) = d$.

It can easily be shown that all relations of the form $\preceq_x$, $\preceq^r_x$, $\preceq_d$ and $\preceq^r_d$ are well-founded. Therefore the Hypothesis 1 is true for the working domain $D_T = \langle \mathcal{T}(\mathcal{C}), \{\preceq_x, \preceq^r_x\}_{x \in PVar}, \{\preceq_d, \preceq^r_d\}_{d \in Data} \rangle$, and hence the whole termination analysis framework presented in the section 2 can be employed.

**Lemma 6.** *The relations $\preceq_x$, $\preceq^r_x$, $x \in PVar$ and $\preceq_d$,$\preceq^r_d$, $d \in Data$ are rational.*

The Büchi automaton representing the initial abstraction of the depth-first tree traversal procedure is depicted in Fig. 4. To simplify the figure, we use only the orders $\preceq^r_x$ and $\preceq^r_{mark}$. Thanks to these orders, there is no potential infinite run in the abstraction.

**Extensions** In order to cover larger classes of programs, we extended our framework in two ways. On one hand, we handle data structures more general than trees, using the invariant generation method from [4]. Here we encode graphs as trees with extra edges.

[1] The semantics of the program instructions considered is given in technical report [22].

The basic idea is that each structure has an underlying tree (called a *backbone*), which stays unchanged during the whole computation. The set of extra edges is specified by *pointer descriptors*, which are references to regular expressions to the set of directions in the tree (left, right, left-up, right-up). We check termination using the existing relations $\preceq_x, \preceq_x^r, x \in PVar$ and $\preceq_d, \preceq_d^r, d \in Data$ on the backbone, as well as two new ones $\preceq_{i:s}$ and $\preceq_{i:s}^r$. Intuitively, $t_1 \preceq_{i:s} t_2$ if the set of positions of $t_1$ whose $i$-th descriptor is set to $s$ is a subset of the set of positions of $t_2$ with the same property.

A second extension is allowing tree left- and right-rotations as program statements. 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.

## 4 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 [4] verifier for safety properties (null-pointer dereferences, memory leaks, etc.). We applied our tool to several programs that manipulate:

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

| 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

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 2.5).

## 5 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. 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 other classes of the programs. 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.

*Acknowledgement.* This work was supported by the French project RNTL AVERILES, the Czech Science Foundation (projects 102/07/0322, 201/09P531), and the Czech Ministry of Education by the project MSM 0021630528.

## 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. 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.
3. 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.
4. A. Bouajjani, P. Habermehl, A. Rogalewicz, T. Vojnar. Abstract Regular Tree Model Checking of Complex Dynamic Data Structures. In *Proc. of SAS'06*, *LNCS* 4134. Springer, 2006.
5. M. Bozga, R. Iosif, and Y. Lakhnech. Flat Parametric Counter Automata. In *Proc. of ICALP'06*, volume 4052 of *LNCS*. Springer, 2006.
6. A. R. Bradley, Z. Manna, and H. B. Sipma. Termination of Polynomial Programs. In *Proc. of VMCAI'2005*, volume 3385 of *LNCS*. Springer, 2005.
7. M. Colón and H. Sipma. Synthesis of Linear Ranking Functions. In *Proc of TACAS'01*, volume 2031 of *LNCS*. Springer, 2001.
8. H. Comon, M. Dauchet, R. Gilleron, F. Jacquemard, D. Lugiez, S. Tison, M. Tommasi. Tree Automata Techniques and Applications, 2005. URL: `www.grappa.univ-lille3.fr/tata`.
9. B. Cook, A. Podelski, and A. Rybalchenko. Abstraction Refinement for Termination. In *Proc. of SAS'05*, volume 3672 of *LNCS*, 2005.
10. Byron Cook, Andreas Podelski, and Andrey Rybalchenko. Terminator: Beyond Safety. In *Proc. of CAV 2006*, volume 4144 of *LNCS*. Springer, 2006.
11. C. Courcoubetis, M. Y. Vardi, P. Wolper, and M. Yannakakis. Memory Efficient Algorithms for the Verification of Temporal Properties. In *Proc. of CAV'90*, *LNCS* 531. Springer, 1991.
12. 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*, *LNCS* 4144. Springer, 2006.
13. 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.
14. 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.
15. R. Iosif and A. Rogalewicz. On the Spuriousness Problem for Tree Manipulating Lassos. Technical Report TR-2008-12, Verimag, 2008.
16. S.K. Lahiri and S. Qadeer. Verifying Properties of Well-Founded Linked Lists. In *Proc. of POPL'06*. ACM Press, 2006.
17. 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.
18. A. Podelski and A. Rybalchenko. Transition Invariants. In *Proc. of LICS'04*. IEEE, 2004.
19. A. Rybalchenko. The ARMC tool. URL: `http://www.mpi-inf.mpg.de/~rybal/armc/`.
20. M. Y. Vardi. The Büchi Complementation Saga. In *Proc. of STACS'07*, volume 4393 of *LNCS*. Springer, 2007.
21. C.S. Lee, N.D. Jones, and A.M. Ben-Amram. The Size-Change Principle for Program Termination. In *Proc of POPL'01*, ACM Press, 2001.
22. R. Iosif and A. Rogalewicz. Automata-based Termination Proofs. Technical Report TR-2008-17, Verimag, 2008.