Loops and Overloops for Tree Walking Automata

Pierre-Cyrille Héam*, Vincent Hugot**, and Olga Kouchnarenko

LIFC, Université de Franche-Comté & INRIA CASSIS, Besançon, France
{pcheam,vhugot,okouchnarenko}@lifc.univ-fcomte.fr

Abstract. Tree Walking Automata (TWA) have lately received renewed interest thanks to their tight connection to XML. This paper introduces the notion of tree overloops, which is closely related to tree loops, and investigates the use of both for the following common operations on TWA: testing membership, transformation into a Bottom-Up Tree Automaton (BUTA), and testing emptiness. Notably, we argue that transformation into a BUTA is slightly less straightforward than was assumed, show that using overloops yields much smaller BUTA in the deterministic case, and provide a polynomial over-approximation of this construction which detects emptiness with surprising accuracy against randomly generated TWA.

Keywords: Tree Walking Automata, loops, overloops, membership, emptiness, approximation

1 Introduction

Tree Walking Automata (TWA for short) are a well-established sequential model for recognising tree languages which was introduced in 1969 by Aho and Ullman [1]. While they originally received far less attention than the better known branching model of tree automata, they have been steadily gathering interest in the last few years. Notably, important questions which had remained open for decades have recently been closed. This renewed interest is owed in great part to the ever-growing popularity of XML, with which they and their variants are tightly connected, in particular through Core XPath [6] and streaming [13].

In this context, it becomes helpful to have reasonably efficient algorithms for essential operations on TWA such as deciding membership and emptiness, as well as transformation into a BUTA. Until now, research has been mainly focused on closing fundamental open problems concerning the expressiveness of TWA [5,2,4]. While algorithms for the above operations are known, they appear in print mostly as proof sketches, and there has been no focus on finding tighter complexity bounds. In contrast, this paper provides explicit algorithms for these tasks and deals with complexity issues. The common thread of our contributions

* This author is supported by the project ANR 2010 BLAN 0202 02 FREC.
** This author is supported by the French DGA (Direction Générale de l'Armement).

is the notion of *tree loop*, which is pervasive to the algorithms we give. This notion may be related to Knuth's construction for testing circularity of attribute grammars [11]. The contributions are organised as follows:

- ◇ Section 2 gives a thorough introduction to tree loops – which are more or less folklore – and introduces a new notion of *tree overloop*. Simple algorithms for testing membership follow naturally from this work. To the best of our knowledge, no such algorithm exists in the literature.
- ◇ Section 3 treats the transformation from TWA to BUTA, based on the proof sketches in [3] and [12, p143]. Two variants are given: one using loops and one using overloops. The latter yields slightly smaller automata in general. Then we show that, in the deterministic case, the overloops-based construction admits a much smaller upper bound on the number of generated states.
- ◇ The emptiness problem is known to be ExpTime-complete for TWA, and is traditionally tested by first transforming the TWA into a BUTA. Section 4 provides a polynomial algorithm which computes an "over-approximation" of this BUTA, and thus can – with luck – decide emptiness positively. This approach is tested against randomly generated TWA, and turns out to be astonishingly accurate. Should it prove inefficient against some families of TWA, then the approximation can be refined as much as needed.

Notations. Let $R \subseteq Q^2$ be a binary relation on a set Q; we denote by R^+ and R^* its transitive and reflexive-transitive closure, respectively. The notation $[\![n, m]\!]$ denotes the integer interval $[n, m] \cap \mathbb{Z}$.

We denote by $\mathbb{N}^*$ the set of words over $\mathbb{N}$; if $v, w \in \mathbb{N}^*$, then $v.w$ stands for the concatenation of the words v and w. A *ranked alphabet* is a finite set of symbols, equipped with an arity function $\mathit{arity} : \Sigma \to \mathbb{N}$. The subset of symbols of Σ with arity k is denoted by Σ_k. The set $\mathcal{T}(\Sigma)$ of trees over Σ is defined inductively as the smallest set such that $\Sigma_0 \subseteq \mathcal{T}(\Sigma)$ and, if $k \geqslant 1$, $f \in \Sigma_k$ and $u_1, \ldots, u_k \in \mathcal{T}(\Sigma)$, then $f(u_1, \ldots, u_k) \in \mathcal{T}(\Sigma)$. If $t \in \mathcal{T}(\Sigma)$ is a tree, then the set of *positions* (or *nodes*) $\mathcal{P}os(t) \subseteq \mathbb{N}^*$ is defined inductively by $\mathcal{P}os(t) = \{\varepsilon\}$ if t is a constant – that is to say, $t \in \Sigma_0$ – and $\mathcal{P}os\big(f(u_1, \ldots, u_n)\big) = \{\varepsilon\} \cup \{\, k.\alpha_k \mid k \in [\![0, n-1]\!] \text{ and } \alpha_k \in \mathcal{P}os(u_{k+1}) \,\}$ otherwise, where n is the arity of f. We see a tree t as a function $t : \mathcal{P}os(t) \to \Sigma$ which maps a position to the symbol at that position in t. In this paper we consider only binary trees, that is to say we assume that $k \notin \{\, 0, 2 \,\} \implies \Sigma_k = \varnothing$. Positions are equipped with a non-strict (resp. strict) partial order $\trianglelefteq$ (resp. $\triangleleft$), such that $\alpha \trianglelefteq \beta$ iff β is a prefix of α (resp. $\alpha \trianglelefteq \beta$ and $\alpha \neq \beta$). The *size* of a tree t is denoted by $\|t\|$ and defined by $\|t\| = |\mathcal{P}os(t)|$.

The *parent function* $\mathfrak{p}(\cdot) : \mathcal{P}os(t) \setminus \{\varepsilon\} \to \mathcal{P}os(t)$ maps any (non-root) *child* node $\alpha.k$ (where $k \in \{\, 0, 1 \,\}$) to its *father* α. We denote by $t|_\alpha$ the subtree of t under α. The reader is assumed to be well-acquainted with the bottom-up variety of branching tree automata (see for instance [7]). A Tree-Walking Automaton (TWA) is a tuple $\mathcal{A} = \langle \Sigma, Q, I, F, \Delta \rangle$ where Q is a finite set of states, Σ a ranked alphabet, $I \subseteq Q$ is the set of initial states, $F \subseteq Q$ the subset of final – or

accepting – states, and

$$\Delta \subseteq \Sigma \times Q \times \underbrace{\{\star, \mathbf{0}, \mathbf{1}\}}_{\mathbb{T}\,:\,\text{types}} \times \underbrace{\{\uparrow, \circlearrowleft, \swarrow, \searrow\}}_{\mathbb{M}\,:\,\text{moves}} \times Q$$

is the set of transitions. In this paper the tuple $\langle \Sigma, Q, I, F, \Delta \rangle$ will be assumed whenever we speak of a TWA $\mathcal{A}$. Each node α of a tree t has a *type* in $\mathbb{T}$, denoted by $\natural\alpha$, such that $\natural\varepsilon = \star$ (root), $\natural(\beta.0) = \mathbf{0}$ (left son), $\natural(\beta.1) = \mathbf{1}$ (right son). As we will seldom deal with the root in practice, we define for short the *sons* $\mathbb{S} = \{\mathbf{0}, \mathbf{1}\} \subset \mathbb{T}$. We will also put in relation types and moves through the function $\chi(\cdot) : \mathbb{S} \to \{\swarrow, \searrow\}$ such that $\chi(\mathbf{0}) = \swarrow$ and $\chi(\mathbf{1}) = \searrow$. For our convenience, we will take the special notation $\langle f, p, \tau \to \mu, q \rangle$ for the tuple $(f, p, \tau, \mu, q) \in \Delta$. Using this notation, some of the parameters can be replaced by sets, with the obvious meaning that we consider the set of all transitions thus described. For instance $\langle \Sigma_2, p, \mathbb{T} \to \circlearrowleft, q \rangle = \{ (\sigma, p, \tau, \circlearrowleft, q) \mid \sigma \in \Sigma_2, \tau \in \mathbb{T} \}$. Note that all the transitions from $\langle \Sigma_0, Q, \mathbb{T} \to \{\swarrow, \searrow\}, Q \rangle \cup \langle \Sigma, Q, \star \to \uparrow, Q \rangle$ are invalid.

A *configuration* of $\mathcal{A}$ on a tree t is a pair $c = (\beta, q) \in \mathcal{P}os(t) \times Q$; it is *initial* if $c \in \{\varepsilon\} \times I$ and *final* (or *accepting*) if $c \in \{\varepsilon\} \times F$. It is a *successor* of a configuration (α, p) if $\langle t(\alpha), p, \natural\alpha \to \mu, q \rangle \in \Delta$, where μ is $\uparrow$ if $\beta = \mathfrak{p}(\alpha)$, $\circlearrowleft$ if $\beta = \alpha$, $\swarrow$ if $\beta = \alpha.0$ and $\searrow$ if $\beta = \alpha.1$. We write $c_1 \twoheadrightarrow_{\mathcal{A}} c_2$ (or simply $c_1 \twoheadrightarrow c_2$ whenever $\mathcal{A}$ is clear from the context) if the configuration c_2 is a successor of c_1. A *run* is a (not necessarily finite) sequence of successive configurations $c_1 \twoheadrightarrow c_2 \twoheadrightarrow \ldots c_n \twoheadrightarrow \ldots$. A run is *accepting* (or *successful*) if it starts with an initial configuration and reaches a final configuration. A tree t is *accepted* or *recognised* by $\mathcal{A}$ if there exists an accepting run of $\mathcal{A}$ on t. The set of all accepted trees is the *language* of $\mathcal{A}$, denoted by $\mathcal{L}\mathrm{ng}(\mathcal{A})$.

Example: Let $\mathcal{X}$ be a TWA such that $\Sigma_0 = \{a, b, c\}$ and $\Sigma_2 = \{f, g, h\}$, $Q = \{q_\ell, q_\mathrm{u}\}$, $I = \{q_\ell\}$, $F = \{q_\mathrm{u}\}$, and $\Delta = \langle a, q_\ell, \{\star, \mathbf{0}\} \to \circlearrowleft, q_\mathrm{u} \rangle \cup \langle \Sigma, q_\mathrm{u}, \mathbf{0} \to \uparrow, q_\mathrm{u} \rangle \cup \langle \Sigma_2, q_\ell, \{\star, \mathbf{0}\} \to \swarrow, q_\ell \rangle$. Then $\mathcal{X}$ accepts exactly all trees whose left-most leaf is labelled by a. We shall use this (trivial) example throughout the paper.

2 Loops, Overloops and the Membership Problem

The notion of *loop* turned out to be very useful to deal with TWA. Informally, loops arise naturally as a generalisation of the definition of an accepting run, where the automaton enters the root in a given initial state p_in, moves along the tree, and then comes back to the root in a certain final state p_out. In practice, the details of the moves which form the loop itself are largely irrelevant and are discarded: the most useful information is the pair of states $(p_\mathrm{in}, p_\mathrm{out})$.

Definition 1 (Tree Loops). Let $\mathcal{A}$ be a TWA, t a tree and $\alpha \in \mathcal{P}os(t)$. A pair of states $(p, q) \in Q^2$ is a *loop* of $\mathcal{A}$ on the subtree $t|_\alpha$ if there exist $n \geqslant 0$ and a run $(\alpha, p), (\beta_1, s_1), \ldots, (\beta_n, s_n), (\alpha, q)$ such that for all $k \in [\![1, n]\!]$, $\beta_k \trianglelefteq \alpha$. Such a run is a *looping run*, and we say that it *forms* the loop (p, q).

Example: The looping run $(0, q_\ell), (0.0, q_\ell), (0.0, q_\mathrm{u}), (0, q_\mathrm{u})$ of $\mathcal{X}$ on the subtree $g(f(a, b), c)|_0 = f(a, b)$ forms the loop (q_ℓ, q_u).

Data: A TWA $\mathcal{A} = \langle \Sigma, Q, I, F, \Delta \rangle$
Result: A BUTA $\mathcal{B}$ such that $\mathcal{L}\mathrm{ng}(\mathcal{B}) = \mathcal{L}\mathrm{ng}(\mathcal{A})$

initialise States and Rules to $\varnothing$
foreach $a \in \Sigma_0, \tau \in \mathbb{T}$ **do**
A let $P = (a, \tau, \mathcal{H}_a^{\tau *})$; add $a \to P$ to Rules and P to States
repeat
  foreach $f \in \Sigma_2, \tau \in \mathbb{T}$ **do**
B add every $f(P_0, P_1) \to P$ to Rules and P to States
  where $P_0, P_1 \in$ States such that $P_0 = (\sigma_0, \mathbf{0}, S_0)$ and $P_1 = (\sigma_1, \mathbf{1}, S_1)$
  and $P = (f, \tau, (\mathcal{H}_f^{\tau} \cup S)^*)$, with
  $$S = \left\{ (p,q) \;\middle|\; \exists \theta \in \mathbb{S}, (p_\theta, q_\theta) \in S_\theta : \middle| \begin{array}{l} \langle f, p, \tau \to \chi(\theta), p_\theta \rangle \in \Delta \text{ and} \\ \langle \sigma_\theta, q_\theta, \theta \to \uparrow, q \rangle \in \Delta \end{array} \right\}$$
until Rules *remains unchanged*
return $\mathcal{B} = \langle \Sigma, \text{States}, \{ (\sigma, \star, L) \in \text{States} \mid L \cap (I \times F) \neq \varnothing \}, \text{Rules} \rangle$

Algorithm 1: Tranformation into BUTA, with loops

Data: An escaped TWA $\mathcal{A} = \langle \Sigma, Q, I, F, \Delta \rangle$ (see Def. 13)
Result: A BUTA $\mathcal{B}$ such that $\mathcal{L}\mathrm{ng}(\mathcal{B}) = \mathcal{L}\mathrm{ng}(\mathcal{A})$

initialise States and Rules to $\varnothing$
foreach $a \in \Sigma_0, \tau \in \mathbb{T}$ **do**
C let $P = (\tau, \mathcal{U}_a^{\tau}[\mathcal{H}_a^{\tau *}])$; add $a \to P$ to Rules and P to States
repeat
  foreach $f \in \Sigma_2, \tau \in \mathbb{T}$ **do**
D add every $f(P_0, P_1) \to P$ to Rules and P to States
  where $P_0, P_1 \in$ States such that $P_0 = (\mathbf{0}, S_0)$ and $P_1 = (\mathbf{1}, S_1)$ and
  $P = (\tau, \mathcal{U}_f^{\tau}\left[(\mathcal{H}_f^{\tau} \cup S)^*\right])$, with
  $$S = \left\{ (p, q_\theta) \;\middle|\; \exists \theta \in \mathbb{S}, p_\theta \in Q : \middle| \begin{array}{l} \langle f, p, \tau \to \chi(\theta), p_\theta \rangle \in \Delta \\ \text{and } (p_\theta, q_\theta) \in S_\theta \end{array} \right\}$$
until Rules *remains unchanged*
return $\mathcal{B} = \langle \Sigma, \text{States}, \{ (\star, O) \in \text{States} \mid O \cap (I \times \{\checkmark\}) \neq \varnothing \}, \text{Rules} \rangle$

Algorithm 2: Tranformation into BUTA, with overloops

Data: An escaped TWA $\mathcal{A} = \langle \Sigma, Q, I, F, \Delta \rangle$ (see Def. 13)
Result: Empty (only if $\mathcal{L}\mathrm{ng}(\mathcal{A}) = \varnothing$) or Unknown

initialise $\mathcal{L}_0$, $\mathcal{L}_1$, $\mathcal{L}_\star$ to $\varnothing$; **foreach** $a \in \Sigma_0, \tau \in \mathbb{T}$ **do** $\mathcal{L}_\tau \leftarrow \mathcal{L}_\tau \cup \mathcal{U}_a^{\tau}[\mathcal{H}_a^{\tau *}]$
repeat
  foreach $f \in \Sigma_2, \tau \in \mathbb{T}$ **do** $\mathcal{L}_\tau \leftarrow \mathcal{L}_\tau \cup \mathcal{U}_f^{\tau}\left[(\mathcal{H}_f^{\tau} \cup S)^*\right]$
  where $S = \left\{ (p, q_\theta) \;\middle|\; \exists \theta \in \mathbb{S}, p_\theta \in Q : \middle| \begin{array}{l} \langle f, p, \tau \to \chi(\theta), p_\theta \rangle \in \Delta \\ \text{and } (p_\theta, q_\theta) \in \mathcal{L}_\theta \end{array} \right\}$
until $\mathcal{L}_0, \mathcal{L}_1, \mathcal{L}_\star$ *remain unchanged*
return Empty *if* $\mathcal{L}_\star \cap (I \times \{\checkmark\}) = \varnothing$, *else* Unknown

Algorithm 3: Approximation for emptiness, with overloops

Notice that loops are not only defined on whole trees, but on subtrees as well with the restriction that the automaton cannot leave the subtree during the looping run. It is in fact this restriction which grants loops their usefulness. TWA, unlike their branching cousins, whose runs are defined inductively, do not naturally lend themselves to inductive reasoning; and yet, thanks to the above restriction, loops are easily computed by induction. Thus loops and their variants can be thought of as convenient devices which hide the sequential, stateful aspect of TWA runs beneath a much more "user-friendly" layer of induction.

In the next few paragraphs we compute the loops of a TWA $\mathcal{A}$ on a subtree $t|_\alpha$.

Definition 2 (Kinds of Loops)**.** Clearly for all $p \in Q$, (p, p) is a loop; we call them *trivial loops*. A looping run of $\mathcal{A}$ on $t|_\alpha$ is *simple* if it reaches α exactly twice. It is *non-trivial* if it reaches α at least twice. A loop is *simple* (resp. *non-trivial*) if there exists a *simple* (resp. *non-trivial*) looping run forming it.

Example: The loop (q_ℓ, q_u) in the above example is simple, because $(0, q_\ell), (0.0, q_\ell)$, $(0.0, q_\mathrm{u}), (0, q_\mathrm{u})$ only reaches $\alpha = 0$ twice, on the first and last configuration. The TWA $\mathcal{X}$ forms only trivial and simple loops, but suppose that we alter it so that it also checks that the *right-most* leaf is a. During an accepting run it would go down and left, back up to the root, down and right, and back up to the root again, in a final state. Thus all accepting runs would be non-trivial and non-simple, reaching the root exactly three times.

Fortunately, we only ever need to compute simple loops, as we can deduce the rest from them thanks to the following lemma:

Lemma 3 (*Loop Decomposition*)**.** *If $S \subseteq Q^2$ is the set of all simple loops of $\mathcal{A}$ on a given subtree $u = t|_\alpha$, then S^* is the set of all loops of $\mathcal{A}$ on u.*

Proof. Every looping run is either trivial or non-trivial. All trivial loops are in S^* by reflexive closure. Furthermore, every non-trivial looping run can easily be decomposed into one or more simple runs. Indeed, any non-trivial looping run ℓ has the following general form, where $\beta_i^k \lhd \alpha$ for all k, i, and the notation $[x_k]^{k \in [\![1,m]\!]}$ designates the run obtained by concatenating the runs $x_1, \ldots, x_m$:

$$\ell = (\alpha, p^0), \left[(\beta_1^k, s_1^k), \ldots, (\beta_{n_k}^k, s_{n_k}^k), (\alpha, p^k)\right]^{k \in [\![1,m]\!]} \quad .$$

This can be seen as the composition of m simple looping runs ℓ_k, for $k \in [\![1, m]\!]$, where $\ell_k = (\alpha, p^{k-1}), (\beta_1^k, s_1^k), \ldots, (\beta_{n_k}^k, s_{n_k}^k), (\alpha, p^k)$. Let us compute the loops formed by the looping run ℓ: for every $k, l \in [\![1, m]\!]$, $k \leqslant l$, we can build a looping run $\ell_k, \ell_{k+1}, \ldots, \ell_l$, and it follows that (p^{k-1}, p^l) is a loop. Since only the states p^k appear at position α, ℓ forms no other loops. But we have

$$\left\{ (p^{k-1}, p^k) \;\middle|\; k \in [\![1, m]\!] \right\}^+ = \left\{ (p^{k-1}, p^l) \;\middle|\; k, l \in [\![1, m]\!] : k \leqslant l \right\} \quad .$$

Note that each loop (p^{k-1}, p^k) is formed by ℓ_k. Therefore the loops formed by the non-trivial looping run ℓ are the transitive closure of the loops formed by the simple looping runs of which it is composed. □

Let us denote $\mho^\tau(u)$ the set of all loops of $\mathcal{A}$ on a subtree u, where τ is the type of the root of u. Concretely, if u is the whole tree, then $\tau = \star$ and, more generally, if u is a subtree, say, $u = t|_\alpha$, then $\tau = \natural\alpha$. Note that thanks to the above-mentioned restriction in the definition of loops, the type of the subtree's root is the only information which is actually needed from the context.

Let $a \in \Sigma_0$ be a leaf of type τ. We compute the loops on a. By definition of a looping run, $\mathcal{A}$ cannot move up; nor can it move down since leaves have no children. So the only transitions which can be activated are $\circlearrowleft$-transitions. As we are only interested in *simple* loops, we can only activate one of these transitions *once*, thus creating runs of the form $(\alpha, p) \twoheadrightarrow (\alpha, q)$, and the corresponding loops (p, q). Let us have a general notation for this:

Definition 4 (Simple Here-Loops). $\mathcal{H}^\tau_\sigma \stackrel{\text{def}}{=} \{\, (p,q) \mid \langle \sigma, p, \tau \to \circlearrowleft, q\rangle \in \Delta \,\}$.

Thus the simple loops on a are $\mathcal{H}^\tau_a$. By Lemma 3 we have $\mho^\tau(a) = (\mathcal{H}^\tau_a)^*$. We now deal with inner nodes. Let $f \in \Sigma_2$, and $u = f(u_{\boldsymbol{0}}, u_{\boldsymbol{1}})$; again, τ denotes the type of the root of u. Clearly the elements of $\mathcal{H}^\tau_f$ are loops on u, as above, but this time $\mathcal{A}$ can move down as well. It cannot move up on the first move (that would mean leaving the subtree), but it will obviously *need* to move up to rejoin the root if it ever moves down. To clarify all that, let us reason on what the first move of a simple looping run can be. It cannot be $\uparrow$ and all simple loops whose first move is $\circlearrowleft$ are already computed in $\mathcal{H}^\tau_f$. Say the first move is $\swarrow$: then the run can do whatever it wants in the left subtree $u_{\boldsymbol{0}}$, after which it has to move back up to the root to complete the loop. Again, we only consider *simple* loops, so no move can be made past this point, as the root has been reached twice already. Thus the general form of such a run is $(\varepsilon, p), (0, p_{\boldsymbol{0}}), (\beta_1, s_1), \ldots, (\beta_n, s_n), (0, q_{\boldsymbol{0}}), (\varepsilon, q)$, with all $\beta_k \trianglelefteq 0$. But by definition, this means that $(p_{\boldsymbol{0}}, q_{\boldsymbol{0}})$ is a loop on $u_{\boldsymbol{0}}$, ie. $(p_{\boldsymbol{0}}, q_{\boldsymbol{0}}) \in \mho^{\boldsymbol{0}}(u_{\boldsymbol{0}})$. Needless to say, the same applies (with $\boldsymbol{1}$ instead of $\boldsymbol{0}$) if the first move is $\searrow$. It follows that to determine whether (p, q) forms a simple loop on u, we need only check three things: 1. $\mathcal{A}$ can move down (left or right) from state p into a state $p_{\boldsymbol{0}}$, 2. there is a loop $(p_{\boldsymbol{0}}, q_{\boldsymbol{0}})$ on this subtree and 3. in state $q_{\boldsymbol{0}}$, $\mathcal{A}$ can move up from this subtree and into the state q. Formally:

$$\mho^\tau(u) = \left(\mathcal{H}^\tau_f \cup \left\{ (p,q) \;\middle|\; \begin{array}{c} \exists \theta \in \mathbb{S} : \\ \exists (p_\theta, q_\theta) \in \mho^\theta(u_\theta) \end{array} \text{ st. } \begin{array}{l} \langle f, p, \tau \to \chi(\theta), p_\theta\rangle \in \Delta \\ \langle u_\theta(\varepsilon), q_\theta, \theta \to \uparrow, q\rangle \in \Delta \end{array} \right\} \right)^* .$$

Theorem 5 (*Loops*). *Let $\mathcal{A}$ be a TWA and $t \in \mathcal{T}(\Sigma)$. Then for all $\alpha \in \mathcal{P}os(t)$, $\mho^{\natural\alpha}(t|_\alpha)$, as defined above, is the set of all loops of $\mathcal{A}$ on $t|_\alpha$.*

Example: For the TWA $\mathcal{X}$, $\mho^{\boldsymbol{0}}(a) = \{\, (q_\ell, q_u) \,\}^* = \{\, (q_\ell, q_\ell), (q_u, q_u), (q_\ell, q_u) \,\}$, and $\mho^\star(f(a,b)) = (\varnothing \cup \{(q_\ell, q_u)\})^*$ (no simple here-loop, and one loop built on the left child). On the other hand, $\mho^\star(f(b,a)) = \varnothing^*$, because $\mho^{\boldsymbol{1}}(a) = \mho^{\boldsymbol{0}}(b) = \varnothing^*$.

Note that a reasonably efficient algorithm for testing membership is straightforwardly derived from the above computation of loops:

Corollary 6 (*TWA Membership*). *Let $\mathcal{A}$ be a TWA and $t \in \mathcal{T}(\Sigma)$. Then we have $t \in \mathcal{L}ng(\mathcal{A})$ if and only if $\mho^\star(t) \cap (I \times F) \neq \varnothing$.*

Corollary 7. *The complexity of TWA membership is* $O\left(\|t\| \cdot \left(|Q|^3 + |\Delta|\right)\right)$.

We now introduce a new notion related to tree loops: *tree overloops*.

Definition 8 (Over-Root, Extended Positions and Transitions)**.** The *extended positions* $\overline{\mathcal{P}os}(t)$ of a tree $t \in \mathcal{T}(\Sigma)$ are the set $\mathcal{P}os(t) \cup \{\overline{\varepsilon}\}$, where $\overline{\varepsilon}$ is called the *overroot*. The parent function $\mathfrak{p}(\cdot)$ is extended over $\overline{\mathcal{P}os}(t)$ into the *extended parent function* $\overline{\mathfrak{p}}(\cdot)$, such that $\overline{\mathfrak{p}}(\varepsilon) = \overline{\varepsilon}$ and $\varepsilon \lhd \overline{\varepsilon}$. The notion of configuration is extended as well, so that the transitions of $\langle \Sigma, Q, \star \to \uparrow, Q\rangle$ become valid. Their application yields configurations of the form $(\overline{\varepsilon}, q)$.

Definition 9 (Tree Over-Loops)**.** Let $\mathcal{A}$ be a TWA and t a tree. A pair of states $(p, q) \in Q^2$ forms an *overloop* of $\mathcal{A}$ on $t|_\alpha$ if there exists a run $(\alpha, p), (\beta_1, s_1), \ldots, (\beta_n, s_n), (\overline{\mathfrak{p}}(\alpha), q)$ such that for all $k \in [\![1, n]\!], \beta_k \trianglelefteq \alpha$.

A way to compute overloops is to compute loops, then check for $\uparrow$-transitions:

Definition 10 (Up-Closure)**.** Let $L \subseteq Q^2, \tau \in \mathbb{T}$ and $\sigma \in \Sigma$:

$$\mathcal{U}^\tau_\sigma[L] \stackrel{\text{def}}{=} \left\{ (p, q) \in Q^2 \;\middle|\; \exists p' \in Q : (p, p') \in L \text{ and } \langle \sigma, p', \tau \to \uparrow, q\rangle \in \Delta \right\} .$$

Lemma 11 (*Up-Closure*)**.** *Let $\mathcal{A}$ be a TWA. If L is the set of all loops of $\mathcal{A}$ on a subtree $u = t|_\alpha$, then $\mathcal{U}^{\natural\alpha}_{t(\alpha)}[L]$ is the set of all overloops of $\mathcal{A}$ on u.*

Similarly to loops, we denote $\mathfrak{O}^\tau(u)$ the set of all overloops of $\mathcal{A}$ on a subtree u, where τ is the type of the root of u. By Lem. 11 we have $\mathfrak{O}^\tau(u) = \mathcal{U}^\tau_{u(\varepsilon)}[\mho^\tau(u)]$, and in the case of leaves this yields $\mathfrak{O}^\tau(a) = \mathcal{U}^\tau_a[(\mathcal{H}^\tau_a)^*]$. However, in the case of inner nodes (say $u = f(u_{\boldsymbol{0}}, u_{\boldsymbol{1}})$), in order to have an inductive computation of overloops instead of one based on loops, we need to compute the overloops of the father, knowing the overloops of the children. The simplest way is to compute the loops of the father and take the up-closure. We only need to check whether 1. the automaton can go down and left (resp. right) from p to a state $p_{\boldsymbol{0}}$ and 2. there is a left (resp. right) overloop $(p_{\boldsymbol{0}}, q_{\boldsymbol{0}})$: this forms a loop $(p, q_{\boldsymbol{0}})$. Formally:

$$\mathfrak{O}^\tau(u) = \mathcal{U}^\tau_f\left[\left(\mathcal{H}^\tau_f \cup \left\{ (p, q_\theta) \;\middle|\; \begin{array}{l} \exists \theta \in \mathbb{S} : \\ \exists p_\theta \in Q \end{array} \text{ st. } \begin{array}{l} \langle f, p, \tau \to \chi(\theta), p_\theta\rangle \in \Delta \\ \text{and } (p_\theta, q_\theta) \in \mathfrak{O}^\theta(u_\theta) \end{array} \right\}\right)^*\right] .$$

Theorem 12 (*Overloops*)**.** *Let $\mathcal{A}$ be a TWA and $t \in \mathcal{T}(\Sigma)$. Then for all $\alpha \in \mathcal{P}os(t)$, $\mathfrak{O}^{\natural\alpha}(t|_\alpha)$, as defined above, is the set of all overloops of $\mathcal{A}$ on $t|_\alpha$.*

Example: For the TWA $\mathcal{X}$, $\mathfrak{O}^{\boldsymbol{0}}(a) = \mathcal{U}^{\boldsymbol{0}}_a\left[\mho^{\boldsymbol{0}}(a)\right] = \{(q_{\mathrm{u}}, q_{\mathrm{u}}), (q_\ell, q_{\mathrm{u}})\}$. However $\mho^\star(f(a, b))$ is the empty set. Thus a small adjustment is needed to test membership using overloops, as standard TWA – such as $\mathcal{X}$ – never admit any overloop at the root of a tree, for lack of $\uparrow$-transitions.

Definition 13 (Overfinal State & Escaped TWA)**.** Let $\mathcal{A} = \langle \Sigma, Q, I, F, \Delta\rangle$ be a TWA; it can be transformed into an *escaped TWA*

$$\mathcal{A}' = \left\langle\ \Sigma,\ Q \uplus \{\checkmark\},\ I, F,\ \Delta \uplus \langle \Sigma, F, \star \to \uparrow, \checkmark\rangle\ \right\rangle ,$$

where $\checkmark \notin Q$ is a fresh state, called *overfinal state*. [Clearly $\mathcal{L}\mathrm{ng}(\mathcal{A}) = \mathcal{L}\mathrm{ng}(\mathcal{A}')$.]

Example: Once $\mathcal{X}$ is escaped, we have $\mathfrak{V}^\star(f(a,b)) = \{\,(q_\mathrm{u},\checkmark),(q_\ell,\checkmark)\,\}$.

Corollary 14 (*TWA Membership Redux*)**.** *Let $\mathcal{A}$ be an escaped TWA and $t \in \mathcal{T}(\Sigma)$. Then $t \in \mathcal{L}ng(\mathcal{A})$ if and only if $\mathfrak{V}^\star(t) \cap (I \times \{\checkmark\}) \neq \varnothing$.*

3 Transforming TWA into equivalent BUTA

It is well-known that every TWA is equivalent to a BUTA; a more general version of this result has been proven in [8] – using game-theoretic arguments – and the main idea of a loop-based transformation from TWA into BUTA is outlined in [3] and [12, p143]. In this section we present two versions of it: the classical, loop-based one (Algo. $1_{[\mathrm{p4}]}$) and an overloop-based variant (Algo. $2_{[\mathrm{p4}]}$). We go on to show that, in the case of deterministic TWA, the overloop-based construction results in much smaller equivalent BUTA than the classical one.

3.1 Two Variants: Loops and Overloops

Lemma 15 (*Loop-Based Algorithm*)**.** *Let $\mathcal{A}$ be a TWA, $\mathcal{B}$ its equivalent BUTA by Algorithm 1, $t \in \mathcal{T}(\Sigma)$ and a subtree $u = t|_\alpha$. Then for every type $\tau \in \mathbb{T}$, there is one unique run ρ of $\mathcal{B}$ on u such that $\rho(\varepsilon) = (u(\varepsilon), \tau, L)$. Furthermore, L is the set of all loops of $\mathcal{A}$ on u, provided that $\natural\alpha = \tau$.*

Proof. Both claims are shown by structural induction on u. *First claim*: If $u = a \in \Sigma_0$, then by line A in Algorithm 1, $\rho(\varepsilon) = P = (a,\tau,L) = (u(\varepsilon),\tau,L)$. It is unique, as only one transition $a \to P$ is generated for each couple a,τ. If $u = f(u_{\boldsymbol{0}}, u_{\boldsymbol{1}}), f \in \Sigma_2$, then by induction hypothesis there exists one run $\rho_{\boldsymbol{0}}$ on $u_{\boldsymbol{0}}$ such that $\rho_{\boldsymbol{0}}(\varepsilon) = P_{\boldsymbol{0}} = (u_{\boldsymbol{0}}(\varepsilon), \boldsymbol{0}, S_{\boldsymbol{0}})$, and one run $\rho_{\boldsymbol{1}}$ on $u_{\boldsymbol{1}}$ such that $\rho_{\boldsymbol{1}}(\varepsilon) = P_{\boldsymbol{1}} = (u_{\boldsymbol{1}}(\varepsilon), \boldsymbol{1}, S_{\boldsymbol{1}})$. Thus by line B in Algo. 1 we use the rule $f(P_{\boldsymbol{0}}, P_{\boldsymbol{1}}) \to P$ to build the run ρ such that $\rho(\varepsilon) = P = (f,\tau,L) = (u(\varepsilon),\tau,L)$, $\rho|_{\boldsymbol{0}} = \rho_{\boldsymbol{0}}$ and $\rho|_{\boldsymbol{1}} = \rho_{\boldsymbol{1}}$. Since $\rho_{\boldsymbol{0}}$ and $\rho_{\boldsymbol{1}}$ are unique, so is ρ. *Second claim:* If $u = a \in \Sigma_0$, then $\rho(\varepsilon) = (a,\tau,\mathcal{H}_a^{\tau\,*})$, and by Theorem 5 we have $\mathcal{H}_a^{\tau\,*} = \mho^\tau(a)$. If $u = f(u_{\boldsymbol{0}}, u_{\boldsymbol{1}})$, then $\rho(\varepsilon) = (f,\tau,(\mathcal{H}_f^\tau \cup S)^*)$ and by induction hypothesis $S_\theta = \mho^\theta(u_\theta)$ and $\sigma_\theta = u_\theta(\varepsilon)$, for all $\theta \in \mathbb{S}$. Thus by Theorem 5, $(\mathcal{H}_f^\tau \cup S)^* = \mho^\tau(u)$. □

Theorem 16. *Algorithm 1 is correct; that is, $\mathcal{L}ng(\mathcal{A}) = \mathcal{L}ng(\mathcal{B})$.*

Proof. If $t \in \mathcal{L}\mathrm{ng}(\mathcal{A})$, then there is a loop $(q_\mathrm{i}, q_\mathrm{f}) \in I \times F$ of $\mathcal{A}$ on t. Therefore there is a run ρ of $\mathcal{B}$ on t such that $\rho(\varepsilon) = (t(\varepsilon), \star, L)$, with $(q_\mathrm{i}, q_\mathrm{f}) \in L$. Thus $\rho(\varepsilon)$ is a final state and $t \in \mathcal{L}\mathrm{ng}(\mathcal{B})$. Conversely, if $t \in \mathcal{L}\mathrm{ng}(\mathcal{B})$ then there is an accepting run ρ of $\mathcal{B}$ on t, that is to say such that $\rho(\varepsilon) = (t(\varepsilon), \star, \mho^\star(t))$ and there exists $(q_\mathrm{i}, q_\mathrm{f}) \in (I \times F) \cap \mho^\star(t)$. Thus by Cor. 6 we have $t \in \mathcal{L}\mathrm{ng}(\mathcal{A})$. □

Two short but important remarks are in order. First: it might seem strange that our states are in $\Sigma \times \mathbb{T} \times 2^{Q^2}$, and not more simply in $\mathbb{T} \times 2^{Q^2}$, as suggested in [12]. In [3] a similar construction – albeit deterministic, see the second remark – is proposed, which does not include Σ either. However, it is not clear how loops could be considered independently from the root symbol of the subtree that bears them.

Consider for instance $a, b \in \Sigma_0$ with only the transitions $\langle \{ a, b \}, p, \tau \to \circlearrowleft, q \rangle$ and $\langle b, q, \tau \to \uparrow, s' \rangle \in \Delta$. Then the loops on a and b are exactly the same – $\{(p, q)\}^*$ – and yet, from their father's point of view, they behave very differently. If $\mathcal{A}$ can go down from a state s to p, it can form a loop (s, s') if the child is b, but not if it is a. In contrast to the loop-based construction, the overloop-based algorithm (Algo. 2) suppresses this problem completely.

Second: the observation made in Lemma 15 that the run of $\mathcal{B}$ is unique, given a subtree and a type, makes it easy to adapt the algorithm to yield a deterministic BUTA. Indeed, every tree in $\mathcal{T}(\Sigma)$ is non-deterministically evaluated by $\mathcal{B}$ into exactly three possible states (one per type); the correct one is chosen according to the context during the run. Recall that rules $f(P_0, P_1) \to P$ are built such that the "type" component of P_θ is θ, and final states bear the root type $\star$. Hence, it suffices to group those three possible states into one element of $\Sigma \times (2^{Q^2})^{|\mathbb{T}|}$ to achieve determinism, which brings us back to the states suggested in [3].

Lemma 17 (*Overloop-Based Algorithm*)**.** *Let $\mathcal{A}$ be a TWA, $\mathcal{B}$ its equivalent BUTA by Algorithm 2, $t \in \mathcal{T}(\Sigma)$ and a subtree $u = t|_\alpha$. Then for all $\tau \in \mathbb{T}$, there is one unique run ρ of $\mathcal{B}$ on u such that $\rho(\varepsilon) = (\tau, O)$. Furthermore, O is the set of all overloops of $\mathcal{A}$ on u, provided that $\natural\alpha = \tau$.*

Theorem 18. *Algorithm 2 is correct; that is, $\mathcal{L}ng(\mathcal{A}) = \mathcal{L}ng(\mathcal{B})$.*

Note that this construction can be adapted to yield deterministic BUTA in exactly the same way as for Algo. 1.

3.2 Overloops and the Deterministic Case

Definition 19 (Deterministic TWA)**.** A TWA $\mathcal{A} = \langle \Sigma, Q, I, F, \Delta \rangle$ is *deterministic* (ie. a *DTWA*) if[(a)] for all $\sigma \in \Sigma, p \in Q, \tau \in \mathbb{T}$, $|\langle \sigma, p, \tau \to \mathbb{M}, Q \rangle \cap \Delta| \leqslant 1$.

Definition 20 (Functional Relation)**.** A relation $R \subseteq Q^2$ is *functional* (or *right-unique*, or *a partial function*) if, for all $p, q, q' \in Q$, pRq and $pRq' \implies q = q'$.

Remark 21. There are $2^{|Q|^2}$ binary relations on Q, of which $|Q + 1|^{|Q|}$ are partial functions, of which $|Q|^{|Q|}$ are total functions.

Remark 22. If a relation R is functional, then so is R^k, for any $k \in \mathbb{N}$.

By construction, a BUTA built by Algo. 1 (loop-based) has at most $|\Sigma| \cdot |\mathbb{T}| \cdot 2^{|Q|^2}$ states, while one built by Algo. 2 (overloop-based) has at most $|\mathbb{T}| \cdot 2^{|Q|^2}$. We will see in this section that, in the deterministic case, this upper bound is in fact much lower for the overloop-based algorithm than for the traditional loop-based one. More specifically, we will show that the following holds:

Theorem 23 (*Deterministic Upper-Bound*)**.** *Let $\mathcal{A}$ be a deterministic TWA and $\mathcal{B}$ its equivalent BUTA built by application of Algorithm 2. Then $\mathcal{B}$ has at most $|\mathbb{T}| \cdot 2^{|Q| \log_2(|Q|+1)}$ states.*

(a) In this paper we do not need the usual, stronger definition, where I is a singleton.

The idea is that every state which we build corresponds exactly to the set L of all loops (resp. overloops) of the automaton $\mathcal{A}$ on a certain subtree u. Since $L \subseteq Q^2$, we can see it as a binary relation on the states. The intuition here is that, if $\mathcal{A}$ is deterministic, and enters the root of u in one given state p, then there "should be" only one possible outcome. More formally:

Lemma 24. *If $\mathcal{A}$ is a deterministic TWA, then $\twoheadrightarrow_{\mathcal{A}}$ is functional.*

Proof. In a given configuration (α, p), over a tree t, $|\langle t(\alpha), p, \natural\alpha \to \mathbb{M}, Q\rangle \cap \Delta| \leqslant 1$. Therefore, (α, p) has at most one successor. □

However, in the case of loops, this does not suffice to make L functional because, determinism notwithstanding, a single (non-trivial) loop may reach the root several times, and in different states, before exiting the subtree. Thus there is nothing to prevent us from having both pLq and pLq', for $q \neq q'$; we show next that in that case, one of these loops is simply an extension of the other.

Lemma 25 (*Hidden Loops*)**.** *Let $p, q, q' \in Q$, $q \neq q'$ such that (p, q) and (p, q') are loops of the TWA $\mathcal{A}$ on a given subtree $t|_\alpha$. Then if $\mathcal{A}$ is deterministic, either (q, q') or (q', q) must be a loop of $\mathcal{A}$ on $t|_\alpha$.*

Proof. By Definition 1, there exist two runs $c_0, \ldots, c_n$ and $d_0, \ldots, d_m$ such that $c_0 = d_0 = (\alpha, p)$, $c_n = (\alpha, q)$ and $d_m = (\alpha, q')$. If $n = m$ then $c_0 \twoheadrightarrow^n c_n$ and $c_0 \twoheadrightarrow^n d_n$ and by Lemma 24 and Remark 22, it follows that $c_n = d_m$. But this contradicts $q \neq q'$, so we must have $n \neq m$. Say that $n < m$. Then $c_n = d_n$, and $(\alpha, q) = d_n, \ldots, d_m = (\alpha, q')$ forms a run. Therefore (q, q') is a loop. Similarly, if $n > m$, then by the same arguments (q', q) is a loop. □

Contrariwise, two overloops cannot be combined to form another overloop on the same subtree, which satisfies the above intuition of a "single outcome":

Lemma 26. *Let $p, q, q' \in Q$, such that (p, q) and (p, q') are overloops of the TWA $\mathcal{A}$ on a given subtree $t|_\alpha$. Then if $\mathcal{A}$ is deterministic, $q = q'$.*

Proof. By Def. 9, there exist $s, s' \in Q$ such that $(\alpha, p), \ldots, (\alpha, s), (\overline{\mathfrak{p}}(\alpha), q)$ and $(\alpha, p), \ldots, (\alpha, s'), (\overline{\mathfrak{p}}(\alpha), q')$ are runs; thus (p, s) and (p, s') are loops. If $s \neq s'$, then by Lem. 25, say, (s, s'), is a loop. So there exist $s_1, \ldots, s_n \in Q, \beta_1 \trianglelefteq \alpha$ $, \ldots, \beta_n \trianglelefteq \alpha$ such that $(\alpha, s), (\beta_1, s_1), \ldots, (\beta_n, s_n), (\alpha, s')$ is a run. Thus we have in particular $(\alpha, s) \twoheadrightarrow (\overline{\mathfrak{p}}(\alpha), q)$ and $(\alpha, s) \twoheadrightarrow (\beta_1, s_1)$. It follows that $\overline{\mathfrak{p}}(\alpha) = \beta_1 \trianglelefteq \alpha$, which is contradictory. Hence $s = s'$. We have both $(\alpha, s) \twoheadrightarrow (\overline{\mathfrak{p}}(\alpha), q)$ and $(\alpha, s) \twoheadrightarrow (\overline{\mathfrak{p}}(\alpha), q')$. Since $\twoheadrightarrow$ is functional (Lem. 24), we have finally $q = q'$. □

With this, we can conclude the proof of Theorem 23.

Proof of Theorem 23. By construction, for every state $P = (\tau, L)$ generated for $\mathcal{B}$ by Algorithm 2, there exists at least a subtree t such that L is the set of overloops of $\mathcal{A}$ on t. Thus, by Lemma 26, L is functional. Therefore, by Remark 21, there are at most $|\mathbb{T}| \cdot |Q + 1|^{|Q|}$ states (or, equivalently, $|\mathbb{T}| \cdot 2^{|Q| \log_2(|Q|+1)}$). □

4 The Emptiness Problem & Experimental Results

Polynomial Over-Approximation for the Emptiness Problem. Testing emptiness of a TWA $\mathcal{A}$ is an ExpTime-complete problem [3]. This is rather unfortunate, as there are practical questions – such as satisfiability of some XPath fragments – which reduce to the emptiness of the language of a TWA. We present in this section a crude but fairly accurate and very expeditious overloops-based algorithm capable of detecting emptiness in a number of cases. Algorithm $3_{[p4]}$ is a variant of Algorithm 2 with the following properties:

Lemma 27 (*Overloops Over-Approximation*)**.** *Let $\mathcal{A}$ be a TWA, then when the execution of Algorithm 3 ends, for any $\tau \in \mathbb{T}$, $\mathcal{L}_\tau \supseteq \bigcup_{t \in \mathcal{T}(\Sigma)} \mathfrak{V}^\tau(t)$.*

Theorem 28. *Algorithm 3 is correct; that is, it yields* Empty *only if $\mathcal{L}ng(\mathcal{A}) = \varnothing$.*

Corollary 29 (*Complexity of the Approximation*)**.** *The execution of Algorithm 3 is done in a time polynomial in the size of $\mathcal{A}$ – more precisely: $O(|\Sigma| \cdot |\mathbb{T}|^2 \cdot |Q|^4 \cdot |\Delta|)$.*

Note that Algorithm 3 can easily be made just as coarse or as fine as the need dictates. At the coarse end of that gamut we have a variant of Algorithm 3 which forgoes type information, thus hoarding up all overloops in a single set $\mathcal{L}$ instead of three, and at the fine end we find something equivalent to Algorithm 2.

Experimental Results. Approximation. The approximation has yielded astonishingly good results with randomly generated TWA: out of the – roughly – ten thousands of automata of various sizes ($2 \leqslant |Q| \leqslant 20$) on which it was tested, 75% of which had empty languages, only two of them yielded Unknown instead of Empty. Those results are – unfortunately – probably much better than what can be expected in practice, as our generation scheme is, for now, very simplistic. It is therefore likely that the generated instances are in some sense trivial wrt. emptiness. Two approaches which we plan on taking to obtain more meaningful results are a study similar to that of [9] to identify interesting instances, and the use of statistically-exploitable generation schemes as in [10]. General Results. Comparing the output of Algos. 1 & 2, we noted that the latter generates smaller automata – the cardinality of each state being ignored – by a factor two or more, depending on the size of the input TWA. The same caveat as above applies concerning the random TWA. Demonstration Software. Readers interested in experimenting with this paper's algorithms will find online [(b)] a proof of concept (binaries and `OCaml` source code), as well as instructions for use.

5 Conclusion

In this paper we have introduced tree overloops, and applied both loops and overloops to common operations on TWA: deciding membership, transforming a TWA into a BUTA, and inexpensively testing emptiness. We have shown that the

[(b)] On `http://lifc.univ-fcomte.fr/~vhugot/TWA`.

use of overloops simplifies transformation into BUTA, and substantially lowers the upper bound in the deterministic case. We intend to pursue this further by using overloops to characterise useful classes of TWA and perform significant simplifications on the automata, hopefully leading to applications to XPath.

Acknowledgements. The authors would like to thank the members of the INRIA ARC ACCESS for interesting discussions on this topic. Our thanks go as well to the anonymous reviewer who provided a tighter complexity bound for Cor. 7, and whose careful proofreading improved the readability of this paper.

References

1. Aho, A., Ullman, J.: Translations on a context free grammar. Information and Control 19(5), 439–475 (1969)
2. Bojańczyk, M.: 1-bounded TWA cannot be determinized. FSTTCS'03, LNCS 2914, 62–73 (2003)
3. Bojańczyk, M.: Tree-Walking Automata. LATA'08 (tutorial), LNCS 5196 (2008), http://www.mimuw.edu.pl/~bojan/papers/twasurvey.pdf
4. Bojańczyk, M., Colcombet, T.: Tree-walking automata cannot be determinized. Theoretical Computer Science 350(2-3), 164–173 (2006)
5. Bojańczyk, M., Colcombet, T.: Tree-walking automata do not recognize all regular languages. pp. 234–243. STOC '05, ACM, New York, NY, USA (2005)
6. ten Cate, B., Segoufin, L.: Transitive closure logic, nested tree walking automata, and XPath. J. ACM 57(3), 251–260 (2010)
7. Comon, H., Dauchet, M., Gilleron, R., Löding, C., Jacquemard, F., Lugiez, D., Tison, S., Tommasi, M.: Tree automata techniques and applications (2007)
8. Cosmadakis, S., Gaifman, H., Kanellakis, P., Vardi, M.: Decidable optimization problems for database logic programs. pp. 477–490. STOC '88, ACM (1988)
9. Heam, P., Hugot, V., Kouchnarenko, O.: Random Generation of Positive TAGEDs wrt. the Emptiness Problem. Tech. Rep. RR-7441, INRIA (11 2010)
10. Héam, P.C., Nicaud, C., Schmitz, S.: Random Generation of Deterministic Tree (Walking) Automata. In: CIAA'09, LNCS, vol. 5642, pp. 115–124 (2009)
11. Knuth, D.: Semantics of context-free languages: Correction. Theory of Computing Systems 5(2), 95–96 (1971)
12. Samuelides, M.: Automates d'arbres à jetons. Ph.D. thesis, Université Paris-Diderot - Paris VII (12 2007), http://tel.archives-ouvertes.fr/tel-00255024/en/
13. Segoufin, L., Vianu, V.: Validating Streaming XML Documents. In: Popa, L. (ed.) PODS. pp. 53–64. ACM (2002)

Appendix: Proofs

Corollary 6:

Proof. There is a loop $(q_\mathrm{i}, q_\mathrm{f}) \in I \times F$ of $\mathcal{A}$ on t iff there is a run of the form $(\varepsilon, q_\mathrm{i}), \dots, (\varepsilon, q_\mathrm{f})$. The first configuration is initial, and the last final. Therefore it is an accepting run, and $t \in \mathcal{L}\mathrm{ng}\,(\mathcal{A})$. □

Corollary 7:

Proof. The computation is done inductively on the structure of t; thus there are $\|t\|$ main operations. Each of those operations includes the computation of the here-loops, which is $O(|\Delta|)$, and for inner nodes, the computation of down-loops, which is $O(|Q|^2 \cdot |\Delta|)$. Lastly, the transitive closure is computed: this is a polynomial operation. For instance using the Roy–Floyd–Warshall algorithm this adds a $\Theta(Q^3)$. All in all, the whole computation is a $O\left(\|t\| \cdot (|Q|^3 + |Q|^2 \cdot |\Delta|)\right)$. □

Lemma 11:

Proof. Immediate from Def. 9, as we have necessarily $\beta_n = \alpha$. Thus any overloop is a loop followed by a move up, and conversely. □

Corollary 14:

Proof. The couple $(q_\mathrm{i}, \checkmark) \in I \times \{\checkmark\}$ is an overloop iff there is a run $(\varepsilon, q_\mathrm{i}), \dots, (\varepsilon, q_\mathrm{f}), (\overline{\varepsilon}, \checkmark)$. By Def. 13, we must have $q_\mathrm{f} \in F$; therefore $t \in \mathcal{L}\mathrm{ng}\,(\mathcal{A})$. □

Theorem 16:

Proof. If $t \in \mathcal{L}\mathrm{ng}\,(\mathcal{A})$, then there is a loop $(q_\mathrm{i}, q_\mathrm{f}) \in I \times F$ of $\mathcal{A}$ on t. Therefore there is a run ρ of $\mathcal{B}$ on t such that $\rho(\varepsilon) = (t(\varepsilon), \star, L)$, with $(q_\mathrm{i}, q_\mathrm{f}) \in L$. Thus $\rho(\varepsilon)$ is a final state and $t \in \mathcal{L}\mathrm{ng}\,(\mathcal{B})$. Conversely, if $t \in \mathcal{L}\mathrm{ng}\,(\mathcal{B})$ then there is an accepting run ρ of $\mathcal{B}$ on t, that is to say such that $\rho(\varepsilon) = (t(\varepsilon), \star, \mho^\star(t))$ and there exists $(q_\mathrm{i}, q_\mathrm{f}) \in (I \times F) \cap \mho^\star(t)$. Thus by Cor. 6 we have $t \in \mathcal{L}\mathrm{ng}\,(\mathcal{A})$. □

Lemma 17:

Proof. See proof of Lemma 15. The only change is that this time, we build the loops, then deduce the overloops from them (Lem. 11). For leaves, the loops are built line C exactly as they are in Algo. 1, line A. For inner nodes and singular loops whose first move is down, the computation on line D is slightly different from that of B, – and in fact easier – as we know the overloops of the children instead of their loops. □

Theorem 18:

Proof. By construction $(i, \checkmark) \in I \times \{\checkmark\}$ is an overloop iff there exists $f \in F$ such that (i, f) is a loop. Same proof as Theorem 16. □

Lemma 27:

Proof. This result is fairly clear when comparing Algorithms 2 and 3. Let us consider a tree t and a subtree $u = t|_\alpha$, with $\tau = \natural\alpha$. We show that $\mathfrak{O}^\tau(u) \subseteq \mathcal{L}_\tau$. *Base case:* $u = a \in \Sigma_0$. Then by the first line of Algo. 3, we have $\mathfrak{O}^\tau(u) = \mathfrak{O}^\tau(a) = \mathcal{U}_a^\tau[\mathcal{H}_a^{\tau *}] \subseteq \mathcal{L}_\tau$. *Inductive case:* If $u = f(u_{\boldsymbol{0}}, u_{\boldsymbol{1}}), f \in \Sigma_2$, then by induction hypothesis we have $\mathfrak{O}^{\boldsymbol{0}}(u_{\boldsymbol{0}}) \subseteq \mathcal{L}_{\boldsymbol{0}}$ and $\mathfrak{O}^{\boldsymbol{1}}(u_{\boldsymbol{1}}) \subseteq \mathcal{L}_{\boldsymbol{1}}$. The expression computed in the main loop is almost the same as that of Thm. 12 for $\mathfrak{O}^\tau(u)$, the only difference being that $\mathcal{L}_\theta$ is used instead of $\mathfrak{O}^\theta(u_\theta)$. Since we have $\mathfrak{O}^\theta(u_\theta) \subseteq \mathcal{L}_\theta$ for all $\theta \in \mathbb{S}$, the expression in Algo. 3 computes *at least* all overloops of $\mathfrak{O}^\tau(u)$ – and adds them to $\mathcal{L}_\tau$. Thus $\mathfrak{O}^\tau(u) \subseteq \mathcal{L}_\tau$. □

Theorem 28:

Proof. Suppose that Algo. 3 yields Empty. By definition, this is the case if and only if $\mathcal{L}_\star \cap (I \times \{\checkmark\}) = \varnothing$. By Lemma 27, we have $\bigcup_{t \in \mathcal{T}(\Sigma)} \mathfrak{O}^\tau(t) \subseteq \mathcal{L}_\tau$ for all types τ, and it follows that, in particular $(\bigcup_{t \in \mathcal{T}(\Sigma)} \mathfrak{O}^\star(t)) \cap (I \times \{\checkmark\}) = \varnothing$. This can be equivalently rephrased as $\forall t \in \mathcal{T}(\Sigma)$, $\mathfrak{O}^\star(t) \cap (I \times \{\checkmark\}) = \varnothing$. By Corollary 14, this translates into: for all $t \in \mathcal{T}(\Sigma), t \notin \mathcal{L}\mathrm{ng}(\mathcal{A})$, that is to say, $\mathcal{L}\mathrm{ng}(\mathcal{A}) = \varnothing$. □

Corollary 29:

Proof. For all types τ, all operations in Algo. 3 which alter $\mathcal{L}_\tau$ add elements to it. The first loop executes a fixed number of times $(|\Sigma_0| \times |\mathbb{T}|)$. The main loop contains only an inner loop which executes a fixed number of times as well $(|\Sigma_2| \times |\mathbb{T}|)$, and the main loop itself executes until no element is added to $\mathcal{L}_{\boldsymbol{0}}$, $\mathcal{L}_{\boldsymbol{1}}$ or $\mathcal{L}_\star$ during the iteration. Since an iteration can only add elements, and each iteration adds at least one, there can be at most

$$\sum_{\tau \in \mathbb{T}} |\mathcal{L}_\tau| = \sum_{\tau \in \mathbb{T}} |Q|^2 = |\mathbb{T}| \times |Q|^2$$

iterations of the main loop. Each iteration of both the first loop and the main inner loop computes a set of overloops, based on two sets of previously-computed (potential) overloops. This operation executes in a time which is a $O(|Q|^2 \cdot |\Delta|)$, and it is executed in total $|\Sigma_0| \cdot |\mathbb{T}| + |\mathbb{T}| \cdot |Q|^2 \cdot (|\Sigma_2| \cdot |\mathbb{T}|)$ times. Overall, the number of times it is executed is a $O(|\Sigma| \cdot |\mathbb{T}|^2 \cdot |Q|^2)$. Globally, the execution time of Algo. 3 is a $O(|\Sigma| \cdot |\mathbb{T}|^2 \cdot |Q|^4 \cdot |\Delta|)$. This is of course a *very* loose bound. □

Appendix: Extended Example

This appendix contains the output of our demonstration program [(c)], with our running example $\mathcal{X}$ as the input TWA, and $g(f(a,b),c)$ as the input tree.

Among other things, the tool displays the results of the transformation into BUTA by Algorithms 1 and 2. Notice that, even for a TWA as trivial as $\mathcal{X}$, the resulting loop-based BUTA $\mathcal{B}_1$ is huge, while the overloop-based BUTA $\mathcal{B}_2$ is comparatively quite small. Numerically, $\|\mathcal{B}_1\| = 1986$ and $\|\mathcal{B}_2\| = 95$, where the size of a BUTA $\langle \Sigma, Q, F, \Delta \rangle$ is defined – in the usual way [7] – as:

$$\| \langle \Sigma, Q, F, \Delta \rangle \| \stackrel{\text{def}}{=} |Q| + \sum_{f(p_1,\ldots,p_n)\to q \in \Delta} \big(\mathit{arity}(f) + 2 \big) \quad .$$

Note that these are the "raw" results of the algorithms; in practice, it is therefore advisable to trim the BUTA, removing useless rules and states. We used the `cleanup` method described in [9], which is stronger than the standard reduction algorithm, even in the absence of global (`TAGED`) constraints. Of course, standard reduction alone would have no effect whatsoever, because Algorithms 1 and 2 yield reduced BUTA by construction. The automata after cleanup, $\mathcal{B}'_1$ and $\mathcal{B}'_2$, were such that $\|\mathcal{B}'_1\| = 1617$ and $\|\mathcal{B}'_2\| = 78$. So we have

$$\frac{\|\mathcal{B}_1\|}{\|\mathcal{B}_2\|} \approx 20.9 \quad \text{and} \quad \frac{\|\mathcal{B}'_1\|}{\|\mathcal{B}'_2\|} \approx 20.7 \quad \text{and} \quad \frac{\|\mathcal{B}_1\|}{\|\mathcal{B}'_1\|} \approx \frac{\|\mathcal{B}_2\|}{\|\mathcal{B}'_2\|} \approx 1.2 \quad .$$

Thus attempts to reduce the resulting BUTA "after the fact" are much less useful than the switch from loops to overloops, and cannot reclaim the extra space taken by the loop-based construction. Of course, this is just an empirical observation on a single, trivial example, but we have observed the same trend with many other TWA, whether hand-written or randomly generated [(d)], and we believe that it is worth mentioning in this appendix.

More will be known about the size differences between the two approaches once tests are done using random TWA generated according to a statistically-exploitable distribution, such as that described in [10].

(c) The reader will find detailed explanations regarding the program's inputs and outputs on the dedicated web page.

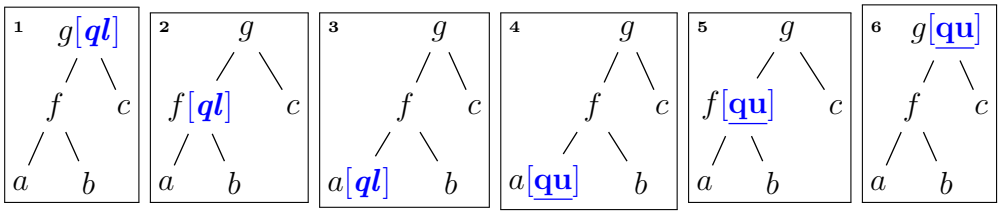
(d) We generated about twenty thousands of them during our tests

TWA Output Proof of Concept;
contact Vincent Hugot (vhugot@lifc.univ-fcomte.fr)
Generated on *2011-04-19 13:16:11 (GMT)*

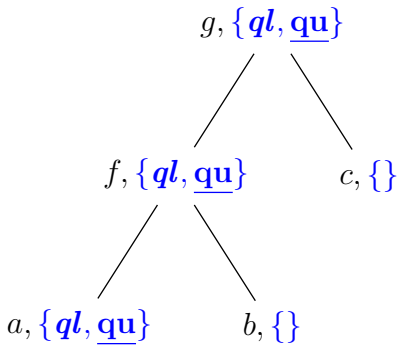
```
TWA [] = {
  alphab = #6{a/0, b/0, c/0, f/2, g/2, h/2}
  states = #2{ql, qu}
  inits = #1{ql}
  finals = #1{qu}
  rules = #14{(a ql @) -> (* qu), (a ql <) -> (* qu), (a
      qu <) -> (! qu), (b qu <) -> (! qu), (c qu <) ->
     (! qu), (f ql @) -> (< ql), (f ql <) -> (< ql), (f
     qu <) -> (! qu), (g ql @) -> (< ql), (g ql <) -> (<
      ql), (g qu <) -> (! qu), (h ql @) -> (< ql), (h ql
      <) -> (< ql), (h qu <) -> (! qu)}
}
```

1 Deterministic run

ACCEPTING
$(\boldsymbol{ql}, \varepsilon) \to (\boldsymbol{ql}, 0) \to (\boldsymbol{ql}, 0.0) \to (\underline{\mathbf{qu}}, 0.0) \to (\underline{\mathbf{qu}}, 0) \to (\underline{\mathbf{qu}}, \varepsilon)$



2 Nondeterministic power run



3 Transformations into BUTA

Note: Post-processing BUTA Cleanup is deactivated.

3.1 Classical loop-based transformation

```
States/Loops correspondance table
0 --> [a,@,#3{(ql,ql), (ql,qu), (qu,qu)}]
1 --> [a,<,#3{(ql,ql), (ql,qu), (qu,qu)}]
2 --> [a,>,#2{(ql,ql), (qu,qu)}]
3 --> [b,@,#2{(ql,ql), (qu,qu)}]
4 --> [b,<,#2{(ql,ql), (qu,qu)}]
5 --> [b,>,#2{(ql,ql), (qu,qu)}]
6 --> [c,@,#2{(ql,ql), (qu,qu)}]
7 --> [c,<,#2{(ql,ql), (qu,qu)}]
8 --> [c,>,#2{(ql,ql), (qu,qu)}]
9 --> [f,@,#3{(ql,ql), (ql,qu), (qu,qu)}]
10 --> [f,@,#2{(ql,ql), (qu,qu)}]
11 --> [f,<,#3{(ql,ql), (ql,qu), (qu,qu)}]
12 --> [f,<,#2{(ql,ql), (qu,qu)}]
13 --> [f,>,#2{(ql,ql), (qu,qu)}]
```

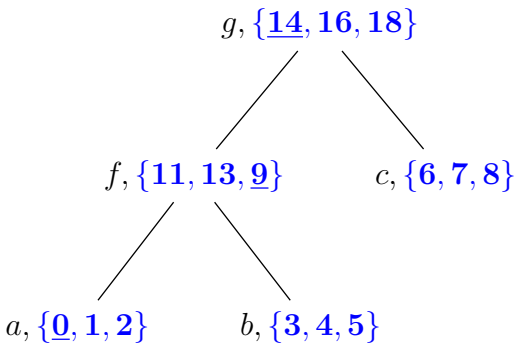
```
14 --> [g,@,#3{(ql,ql), (ql,qu), (qu,qu)}]
15 --> [g,@,#2{(ql,ql), (qu,qu)}]
16 --> [g,<,#3{(ql,ql), (ql,qu), (qu,qu)}]
17 --> [g,<,#2{(ql,ql), (qu,qu)}]
18 --> [g,>,#2{(ql,ql), (qu,qu)}]
19 --> [h,@,#3{(ql,ql), (ql,qu), (qu,qu)}]
20 --> [h,@,#2{(ql,ql), (qu,qu)}]
21 --> [h,<,#3{(ql,ql), (ql,qu), (qu,qu)}]
22 --> [h,<,#2{(ql,ql), (qu,qu)}]
23 --> [h,>,#2{(ql,ql), (qu,qu)}]


TAGED 'fromTWA-Sigma' [1986] = {
  alphab = #6{a/0, b/0, c/0, f/2, g/2, h/2}
  states = #24{0, 1, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19,
      2, 20, 21, 22, 23, 3, 4, 5, 6, 7, 8, 9}
  final = #4{0, 14, 19, 9}
  rules = #495{a()->0, a()->1, a()->2, b()->3, b()->4, b()
     ->5,
c()->6, c()->7, c()->8, f(1, 13)->11, f(1, 13)->13,
f(1, 13)->9, f(1, 18)->11, f(1, 18)->13, f(1, 18)->9,
f(1, 2)->11, f(1, 2)->13, f(1, 2)->9, f(1, 23)->11,
f(1, 23)->13, f(1, 23)->9, f(1, 5)->11, f(1, 5)->13,
f(1, 5)->9, f(1, 8)->11, f(1, 8)->13, f(1, 8)->9,
f(11, 13)->11, f(11, 13)->13, f(11, 13)->9, f(11,
18)->11, f(11, 18)->13, f(11, 18)->9, f(11, 2)->11,
f(11, 2)->13, f(11, 2)->9, f(11, 23)->11, f(11, 23)->13,
f(11, 23)->9, f(11, 5)->11, f(11, 5)->13, f(11, 5)->9,
f(11, 8)->11, f(11, 8)->13, f(11, 8)->9, f(12, 13)->10,
f(12, 13)->12, f(12, 13)->13, f(12, 18)->10, f(12,
18)->12, f(12, 18)->13, f(12, 2)->10, f(12, 2)->12,
f(12, 2)->13, f(12, 23)->10, f(12, 23)->12, f(12,
23)->13, f(12, 5)->10, f(12, 5)->12, f(12, 5)->13,
f(12, 8)->10, f(12, 8)->12, f(12, 8)->13, f(16, 13)->11,
f(16, 13)->13, f(16, 13)->9, f(16, 18)->11, f(16,
18)->13, f(16, 18)->9, f(16, 2)->11, f(16, 2)->13,
f(16, 2)->9, f(16, 23)->11, f(16, 23)->13, f(16, 23)->9,
f(16, 5)->11, f(16, 5)->13, f(16, 5)->9, f(16, 8)->11,
f(16, 8)->13, f(16, 8)->9, f(17, 13)->10, f(17, 13)->12,
f(17, 13)->13, f(17, 18)->10, f(17, 18)->12, f(17,
18)->13, f(17, 2)->10, f(17, 2)->12, f(17, 2)->13,
f(17, 23)->10, f(17, 23)->12, f(17, 23)->13, f(17,
5)->10, f(17, 5)->12, f(17, 5)->13, f(17, 8)->10,
```

```
f(17, 8)->12, f(17, 8)->13, f(21, 13)->11, f(21, 13)->13,
f(21, 13)->9, f(21, 18)->11, f(21, 18)->13, f(21,
18)->9, f(21, 2)->11, f(21, 2)->13, f(21, 2)->9, f(21,
23)->11, f(21, 23)->13, f(21, 23)->9, f(21, 5)->11,
f(21, 5)->13, f(21, 5)->9, f(21, 8)->11, f(21, 8)->13,
f(21, 8)->9, f(22, 13)->10, f(22, 13)->12, f(22, 13)->13,
f(22, 18)->10, f(22, 18)->12, f(22, 18)->13, f(22,
2)->10, f(22, 2)->12, f(22, 2)->13, f(22, 23)->10,
f(22, 23)->12, f(22, 23)->13, f(22, 5)->10, f(22,
5)->12, f(22, 5)->13, f(22, 8)->10, f(22, 8)->12,
f(22, 8)->13, f(4, 13)->10, f(4, 13)->12, f(4, 13)->13,
f(4, 18)->10, f(4, 18)->12, f(4, 18)->13, f(4, 2)->10,
f(4, 2)->12, f(4, 2)->13, f(4, 23)->10, f(4, 23)->12,
f(4, 23)->13, f(4, 5)->10, f(4, 5)->12, f(4, 5)->13,
f(4, 8)->10, f(4, 8)->12, f(4, 8)->13, f(7, 13)->10,
f(7, 13)->12, f(7, 13)->13, f(7, 18)->10, f(7, 18)->12,
f(7, 18)->13, f(7, 2)->10, f(7, 2)->12, f(7, 2)->13,
f(7, 23)->10, f(7, 23)->12, f(7, 23)->13, f(7, 5)->10,
f(7, 5)->12, f(7, 5)->13, f(7, 8)->10, f(7, 8)->12,
f(7, 8)->13, g(1, 13)->14, g(1, 13)->16, g(1, 13)->18,
g(1, 18)->14, g(1, 18)->16, g(1, 18)->18, g(1, 2)->14,
g(1, 2)->16, g(1, 2)->18, g(1, 23)->14, g(1, 23)->16,
g(1, 23)->18, g(1, 5)->14, g(1, 5)->16, g(1, 5)->18,
g(1, 8)->14, g(1, 8)->16, g(1, 8)->18, g(11, 13)->14,
g(11, 13)->16, g(11, 13)->18, g(11, 18)->14, g(11,
18)->16, g(11, 18)->18, g(11, 2)->14, g(11, 2)->16,
g(11, 2)->18, g(11, 23)->14, g(11, 23)->16, g(11,
23)->18, g(11, 5)->14, g(11, 5)->16, g(11, 5)->18,
g(11, 8)->14, g(11, 8)->16, g(11, 8)->18, g(12, 13)->15,
g(12, 13)->17, g(12, 13)->18, g(12, 18)->15, g(12,
18)->17, g(12, 18)->18, g(12, 2)->15, g(12, 2)->17,
g(12, 2)->18, g(12, 23)->15, g(12, 23)->17, g(12,
23)->18, g(12, 5)->15, g(12, 5)->17, g(12, 5)->18,
g(12, 8)->15, g(12, 8)->17, g(12, 8)->18, g(16, 13)->14,
g(16, 13)->16, g(16, 13)->18, g(16, 18)->14, g(16,
18)->16, g(16, 18)->18, g(16, 2)->14, g(16, 2)->16,
g(16, 2)->18, g(16, 23)->14, g(16, 23)->16, g(16,
23)->18, g(16, 5)->14, g(16, 5)->16, g(16, 5)->18,
g(16, 8)->14, g(16, 8)->16, g(16, 8)->18, g(17, 13)->15,
g(17, 13)->17, g(17, 13)->18, g(17, 18)->15, g(17,
18)->17, g(17, 18)->18, g(17, 2)->15, g(17, 2)->17,
g(17, 2)->18, g(17, 23)->15, g(17, 23)->17, g(17,
23)->18, g(17, 5)->15, g(17, 5)->17, g(17, 5)->18,
```

```
g(17, 8)->15, g(17, 8)->17, g(17, 8)->18, g(21, 13)->14,
g(21, 13)->16, g(21, 13)->18, g(21, 18)->14, g(21,
18)->16, g(21, 18)->18, g(21, 2)->14, g(21, 2)->16,
g(21, 2)->18, g(21, 23)->14, g(21, 23)->16, g(21,
23)->18, g(21, 5)->14, g(21, 5)->16, g(21, 5)->18,
g(21, 8)->14, g(21, 8)->16, g(21, 8)->18, g(22, 13)->15,
g(22, 13)->17, g(22, 13)->18, g(22, 18)->15, g(22,
18)->17, g(22, 18)->18, g(22, 2)->15, g(22, 2)->17,
g(22, 2)->18, g(22, 23)->15, g(22, 23)->17, g(22,
23)->18, g(22, 5)->15, g(22, 5)->17, g(22, 5)->18,
g(22, 8)->15, g(22, 8)->17, g(22, 8)->18, g(4, 13)->15,
g(4, 13)->17, g(4, 13)->18, g(4, 18)->15, g(4, 18)->17,
g(4, 18)->18, g(4, 2)->15, g(4, 2)->17, g(4, 2)->18,
g(4, 23)->15, g(4, 23)->17, g(4, 23)->18, g(4, 5)->15,
g(4, 5)->17, g(4, 5)->18, g(4, 8)->15, g(4, 8)->17,
g(4, 8)->18, g(7, 13)->15, g(7, 13)->17, g(7, 13)->18,
g(7, 18)->15, g(7, 18)->17, g(7, 18)->18, g(7, 2)->15,
g(7, 2)->17, g(7, 2)->18, g(7, 23)->15, g(7, 23)->17,
g(7, 23)->18, g(7, 5)->15, g(7, 5)->17, g(7, 5)->18,
g(7, 8)->15, g(7, 8)->17, g(7, 8)->18, h(1, 13)->19,
h(1, 13)->21, h(1, 13)->23, h(1, 18)->19, h(1, 18)->21,
h(1, 18)->23, h(1, 2)->19, h(1, 2)->21, h(1, 2)->23,
h(1, 23)->19, h(1, 23)->21, h(1, 23)->23, h(1, 5)->19,
h(1, 5)->21, h(1, 5)->23, h(1, 8)->19, h(1, 8)->21,
h(1, 8)->23, h(11, 13)->19, h(11, 13)->21, h(11, 13)->23,
h(11, 18)->19, h(11, 18)->21, h(11, 18)->23, h(11,
2)->19, h(11, 2)->21, h(11, 2)->23, h(11, 23)->19,
h(11, 23)->21, h(11, 23)->23, h(11, 5)->19, h(11,
5)->21, h(11, 5)->23, h(11, 8)->19, h(11, 8)->21,
h(11, 8)->23, h(12, 13)->20, h(12, 13)->22, h(12,
13)->23, h(12, 18)->20, h(12, 18)->22, h(12, 18)->23,
h(12, 2)->20, h(12, 2)->22, h(12, 2)->23, h(12, 23)->20,
h(12, 23)->22, h(12, 23)->23, h(12, 5)->20, h(12,
5)->22, h(12, 5)->23, h(12, 8)->20, h(12, 8)->22,
h(12, 8)->23, h(16, 13)->19, h(16, 13)->21, h(16,
13)->23, h(16, 18)->19, h(16, 18)->21, h(16, 18)->23,
h(16, 2)->19, h(16, 2)->21, h(16, 2)->23, h(16, 23)->19,
h(16, 23)->21, h(16, 23)->23, h(16, 5)->19, h(16,
5)->21, h(16, 5)->23, h(16, 8)->19, h(16, 8)->21,
h(16, 8)->23, h(17, 13)->20, h(17, 13)->22, h(17,
13)->23, h(17, 18)->20, h(17, 18)->22, h(17, 18)->23,
h(17, 2)->20, h(17, 2)->22, h(17, 2)->23, h(17, 23)->20,
h(17, 23)->22, h(17, 23)->23, h(17, 5)->20, h(17,
```

```
5)->22, h(17, 5)->23, h(17, 8)->20, h(17, 8)->22,
h(17, 8)->23, h(21, 13)->19, h(21, 13)->21, h(21,
13)->23, h(21, 18)->19, h(21, 18)->21, h(21, 18)->23,
h(21, 2)->19, h(21, 2)->21, h(21, 2)->23, h(21, 23)->19,
h(21, 23)->21, h(21, 23)->23, h(21, 5)->19, h(21,
5)->21, h(21, 5)->23, h(21, 8)->19, h(21, 8)->21,
h(21, 8)->23, h(22, 13)->20, h(22, 13)->22, h(22,
13)->23, h(22, 18)->20, h(22, 18)->22, h(22, 18)->23,
h(22, 2)->20, h(22, 2)->22, h(22, 2)->23, h(22, 23)->20,
h(22, 23)->22, h(22, 23)->23, h(22, 5)->20, h(22,
5)->22, h(22, 5)->23, h(22, 8)->20, h(22, 8)->22,
h(22, 8)->23, h(4, 13)->20, h(4, 13)->22, h(4, 13)->23,
h(4, 18)->20, h(4, 18)->22, h(4, 18)->23, h(4, 2)->20,
h(4, 2)->22, h(4, 2)->23, h(4, 23)->20, h(4, 23)->22,
h(4, 23)->23, h(4, 5)->20, h(4, 5)->22, h(4, 5)->23,
h(4, 8)->20, h(4, 8)->22, h(4, 8)->23, h(7, 13)->20,
h(7, 13)->22, h(7, 13)->23, h(7, 18)->20, h(7, 18)->22,
h(7, 18)->23, h(7, 2)->20, h(7, 2)->22, h(7, 2)->23,
h(7, 23)->20, h(7, 23)->22, h(7, 23)->23, h(7, 5)->20,
h(7, 5)->22, h(7, 5)->23, h(7, 8)->20, h(7, 8)->22,
h(7, 8)->23}
  ==rel = #0{}
  <>rel = #0{}
}
```

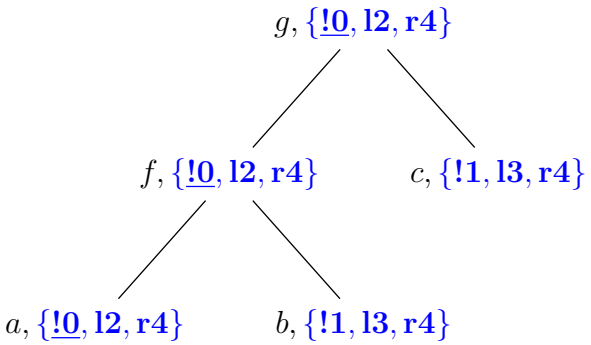


3.2 New overloop-based transformation

```
States/Loops correspondance table
!0 --> [@,#2{(ql,#), (qu,#)}]
!1 --> [@,#1{(qu,#)}]
l2 --> [<,#2{(ql,qu), (qu,qu)}]
l3 --> [<,#1{(qu,qu)}]
```

```
r4 --> [>,#0{}]


TAGED 'fromTWA' [95] = {
  alphab = #6{a/0, b/0, c/0, f/2, g/2, h/2}
  states = #5{!0, !1, l2, l3, r4}
  final = #1{!0}
  rules = #27{a()->!0, a()->l2, a()->r4, b()->!1, b()->l3, b
     ()->r4,
c()->!1, c()->l3, c()->r4, f(l2, r4)->!0, f(l2, r4)->l2,
f(l2, r4)->r4, f(l3, r4)->!1, f(l3, r4)->l3, f(l3,
r4)->r4, g(l2, r4)->!0, g(l2, r4)->l2, g(l2, r4)->r4,
g(l3, r4)->!1, g(l3, r4)->l3, g(l3, r4)->r4, h(l2,
r4)->!0, h(l2, r4)->l2, h(l2, r4)->r4, h(l3, r4)->!1,
h(l3, r4)->l3, h(l3, r4)->r4}
  ==rel = #0{}
  <>rel = #0{}
}
```



4 Membership by overloops

```
Overloops of the TWA on the tree :


#2{(ql,#), (qu,#)}
```

5 Term in language

5.1 Loops

(as string)

a

(as tree representation)

a, **(0)**

5.2 Over-Loops

(as string)
a
(as tree representation)

a, **(!0)**

6 Over-Approximation

```
Found potential overloop @ #2{(ql,#), (qu,#)}

#0{}
#0{}
#0{}
```

Unknown