

VISIBLY TREE AUTOMATA WITH MEMORY AND CONSTRAINTS *

HUBERT COMON-LUNDH^a, FLORENT JACQUEMARD^b, AND NICOLAS PERRIN^c

^a LSV, CNRS/ENS Cachan

e-mail address: h.comon-lundh@aist.go.jp

^b INRIA Saclay & LSV (CNRS/ENS Cachan)

e-mail address: florent.jacquemard@inria.fr

^c ENS Lyon

e-mail address: nicolas.perrin@ens-lyon.fr

ABSTRACT. Tree automata with one memory have been introduced in 2001. They generalize both pushdown (word) automata and the tree automata with constraints of equality between brothers of Bogaert and Tison. Though it has a decidable emptiness problem, the main weakness of this model is its lack of good closure properties.

We propose a generalization of the visibly pushdown automata of Alur and Madhusudan to a family of tree recognizers which carry along their (bottom-up) computation an auxiliary unbounded memory with a tree structure (instead of a symbol stack). In other words, these recognizers, called Visibly Tree Automata with Memory (VTAM) define a subclass of tree automata with one memory enjoying Boolean closure properties. We show in particular that they can be determinized and the problems like emptiness, membership, inclusion and universality are decidable for VTAM. Moreover, we propose several extensions of VTAM whose transitions may be constrained by different kinds of tests between memories and also constraints à la Bogaert and Tison. We show that some of these classes of constrained VTAM keep the good closure and decidability properties, and we demonstrate their expressiveness with relevant examples of tree languages.

INTRODUCTION

The control flow of programs with calls to functions can be abstracted as pushdown systems. This allows to reduce some program verification problems to problems (e.g. model-checking) on pushdown automata. When it comes to functional languages with *continuation passing style*, the stack must contain information on continuations and has the structure of a dag (for jumps). Similarly, in the context of asynchronous concurrent programming languages, for two concurrent threads the ordering of return is not determined (synchronized)

1998 ACM Subject Classification: F.1.1; F.1.2; I.2.2; I.2.3.

Key words and phrases: Tree automata, Pushdown Automata, Alternating automata, Symbolic constraints, First-order theorem proving.

* An extended abstract containing some of the results presented in this paper has appeared in the proceeding of FOSSACS'07.

and these threads can not be stacked. In these cases, the control flow is better modeled as a tree structure rather than a stack. That is why we are interested in tree automata with one memory, which generalize the pushdown (tree) automata, replacing the a stack with a tree. Here, a “memory” has to be understood as a storage device, whose structure is a tree. For instance, two memories would correspond to two storage devices whose access would be independent.

The *tree automata with one memory* introduced in [7] compute bottom-up on a tree, with an auxiliary memory carrying a tree, as in former works such as [14]. Along a computation, at any node of the tree, the memory is updated incrementally from the memory reached at the sons of the node. This update may consist in building a new tree from the memories at the sons (this generalizes a push) or retrieving a subtree of one of the memories at the sons (this generalizes a pop). In addition, such automata may perform equality tests: a transition may be constrained to be performed, only when the memories reached at some of the sons are identical. In this way, tree automata with one memory also generalize certain cases of tree automata with equality and disequality tests between brothers [4].

Automata with one memory have been introduced in the context of the verification of security protocols, where the messages exchanged are represented as trees. In the context of (functional or concurrent) programs, the creation of a thread, or a **callcc**, corresponds to a push, the termination of a thread or a **callcc** corresponds to a pop. The emptiness problem for such automata is in EXPTIME (note that for the extension with a second memory the emptiness problem becomes undecidable). However, the class of tree languages defined by such automata is neither closed by intersection nor by complement. This is not surprising as they are strictly more general than context free languages.

On the other hand, Alur and Madhusudan have introduced the notion of visibility for pushdown automata [2], which is a relevant restriction in the context of control flow analysis. With this restriction, determinization is possible and actually the class of languages is closed under Boolean operations.

In this paper, we propose the new formalism of Visibly Tree Automata with Memory (VTAM). On one hand, it extends visibly pushdown languages to the recognition of trees, and with a tree structure instead of a stack, following former approaches [14, 21, 10]. On the other hand, VTAM restrict tree automata with one memory, imposing a visibility condition on the transitions: each symbol is assigned a given type of action. When reading a symbol, the automaton can only perform the assigned type of action: push or pop.

We first show in Section 2 that VTAM can be determinized, using a proof similar to the proof of [2], and do have the good closure properties. The main difficulty here is to understand what is a good notion of visibility for trees, with memories instead of stacks. We also show that the problems of membership and emptiness are decidable in deterministic polynomial time for VTAM.

In a second part of the paper (Section 3), we extended VTAM with constraints. Our constraints here are recognizable relations; a transition can be fired only if the memory contents of the sons of the current node satisfy such a relation. We give then a general theorem, expressing conditions on the relations, which ensure the decidability of emptiness. Such conditions are shown to be necessary on one hand, and, on the other hand, we prove that they are satisfied by some examples, including syntactic equality and disequality tests and structural equality and disequality tests. The case of VTAM with structural equality and disequality tests (this class is denoted $\text{VTAM}_{\neq}^{\equiv}$) is particularly interesting, since the

determinization and closure properties of Section 2 carry over this generalization, which we show in Section 3.4.2. The automata of $\text{VTAM}_{\overline{\neq}}$ also enjoy a good expressive power, as we show in Section 3.7 by presenting some non-trivial examples of languages in this class: well-balanced binary trees, red-black trees, powerlists...

As an intermediate result, we show that, in case of equality tests or structural equality tests, the language of memories that can be reached in a given state is always a regular language. This is a generalization of the well-known result that the set of stack contents in a pushdown automaton is always regular. To prove this, we observe that the memories contents are recognized by a two-way alternating tree automaton with constraints. Then we show, using a saturation strategy, that two-way alternating tree automata with (structural) equality constraints are not more expressive than standard tree automata.

Finally, in Section 4 we propose a class of visibly tree automata, which combines the structural constraints of $\text{VTAM}_{\overline{\neq}}$, testing memory contents, with Bogaert-Tison constraints of [4] (equality and disequality tests between brothers subterms) which operate on the term in input. We show that the tree automata of this class can be determinized, are closed under Boolean operations and have a decidable emptiness problem.

Related Work. Generalizations of pushdown automata to trees (both for input and stack) are proposed in [14, 21, 10]. Our contributions are the generalization of the visibility condition of [2] to such tree automata – our VTAM (without constraints) strictly generalize the VP Languages of [2], and the addition of constraints on the stack contents. The visibly tree automata of [1] use a word stack which is less general than a tree structured memory but the comparison with VTAM is not easy as they are alternating and compute top-down on infinite trees.

Independently, Chabin and Rety have proposed [5] a formalism combining pushdown tree automata of [14] with the concept of visibly pushdown languages. Their automata recognize finite trees using a word stack. They have a decidable emptiness problem and the corresponding tree languages (Visibly Pushdown Tree Languages, VP TL) are closed under Boolean operations. Following remarks of one of these two authors, it appeared that VTAM and VP TL are incomparable, see Section 2.2.

1. PRELIMINARIES

1.1. Term algebra. A *signature* Σ is a finite set of function symbols with arity, denoted by f, g, \dots . We write Σ_n the subset of function symbols of Σ of arity n . Given an infinite set \mathcal{X} of variables, the set of terms built over Σ and \mathcal{X} is denoted $\mathcal{T}(\Sigma, \mathcal{X})$, and the subset of ground terms is denoted $\mathcal{T}(\Sigma)$. The set of variables occurring in a term $t \in \mathcal{T}(\Sigma, \mathcal{X})$ is denoted $\text{vars}(t)$. A *substitution* σ is a mapping from \mathcal{X} to $\mathcal{T}(\Sigma, \mathcal{X})$ such that $\{x \mid \sigma(x) \neq x\}$, the *support* of σ , is finite. The application of a substitution σ to a term t is written $t\sigma$. It is the homomorphic extension of σ to $\mathcal{T}(\Sigma, \mathcal{X})$. The *positions* $\text{Pos}(t)$ in a term t are sequences of positive integers (Λ , the empty sequence, is the root position). A subterm of t at position p is written $t|_p$, and the replacement in t of the subterm at position p by u denoted $t[u]_p$.

1.2. Rewriting. We assume standard definitions and notations for term rewriting [11]. A *term rewriting system* (TRS) over a signature Σ is a finite set of rewrite rules $\ell \rightarrow r$, where $\ell \in \mathcal{T}(\Sigma, \mathcal{X})$ and $r \in \mathcal{T}(\Sigma, \text{vars}(\ell))$. A term $t \in \mathcal{T}(\Sigma, \mathcal{X})$ rewrites to s by a TRS \mathcal{R} (denoted $t \rightarrow_{\mathcal{R}} s$) if there is a rewrite rule $\ell \rightarrow r \in \mathcal{R}$, a position p of t and a substitution σ such that $t|_p = \ell\sigma$ and $s = t[r\sigma]_p$. The transitive and reflexive closure of $\rightarrow_{\mathcal{R}}$ is denoted $\xrightarrow{*}_{\mathcal{R}}$.

1.3. Tree Automata. Following definitions and notation of [8], we consider tree automata which compute bottom-up (from leaves to root) on (finite) ground terms in $\mathcal{T}(\Sigma)$. At each stage of computation on a tree t , a tree automaton reads the function symbol f at the current position p in t and updates its current state, according to f and to the respective states reached at the positions immediately under p in t . Formally, a bottom-up *tree automaton* (TA) \mathcal{A} on a signature Σ is a tuple (Q, Q_f, Δ) where Σ is the computation signature, Q is a finite set of nullary state symbols, disjoint from Σ , $Q_f \subseteq Q$ is the subset of final states and Δ is a set of rewrite rules of the form: $f(q_1, \dots, q_n) \rightarrow q$, where $f \in \Sigma$ and $q_1, \dots, q_n \in Q$. A term t is *accepted* (we may also write *recognized*) by \mathcal{A} in state q iff $t \xrightarrow{\Delta^*} q$, and the *language* $L(\mathcal{A}, q)$ of \mathcal{A} in state q is the set of ground terms accepted in q . The language $L(\mathcal{A})$ of \mathcal{A} is $\bigcup_{q \in Q_f} L(\mathcal{A}, q)$ and a set of ground terms is called *regular* if it is the language of a TA.

2. VISIBLY TREE AUTOMATA WITH MEMORY

We propose in this section a subclass of the tree automata with one memory [7] which is stable under Boolean operations and has decidable emptiness and membership problems.

2.1. Definition of VTAM. Tree automata have been extended [14, 21, 10, 7] to carry an unbounded information along the states in computations. In [7], this information is stored in a tree structure and is called *memory*. We keep this terminology here, and call our recognizers *tree automata with memory* (TAM). For consistency with the above formalisms, the memory contents will be ground terms over a *memory signature* Γ .

Like for TA we consider bottom-up computations of TAM in trees; at each stage of computation on a tree t , a TAM, like a TA, reads the function symbol at the current position p in t and updates its current state, according to the states reached immediately under p . Moreover, a configuration of TAM contains not only a state but also a memory, which is a tree. The current memory is updated according to the respective contents of memories reached in the nodes immediately under p in t .

As above, we use term rewrite systems in order to define the transitions allowed in a TAM. For this purpose, we add an argument to state symbols, which will contain the memory. Hence, a configuration of TAM in state q and whose memory content is the ground term $m \in \mathcal{T}(\Gamma)$, is represented by the term $q(m)$. We propose below a very general definition of TAM. It is similar to the one of [7], except that we have here general patterns m_1, \dots, m_n, m , while these patterns are restricted in [7], for instance avoiding memory duplications. Since we aim at providing closure and decision properties, we will also impose (other) restrictions later on.

Definition 2.1. A bottom-up *tree automaton with memory* (TAM) on a signature Σ is a tuple (Γ, Q, Q_f, Δ) where Γ is a memory signature, Q is a finite set of unary state symbols, disjoint from $\Sigma \cup \Gamma$, $Q_f \subseteq Q$ is the subset of final states and Δ is a set of rewrite rules of the

form $f(q_1(m_1), \dots, q_n(m_n)) \rightarrow q(m)$ where $f \in \Sigma_n$, $q_1, \dots, q_n, q \in Q$ and $m_1, \dots, m_n, m \in \mathcal{T}(\Gamma, \mathcal{X})$.

The rules of Δ are also called *transition rules*. A term t is *accepted* by \mathcal{A} in state $q \in Q$ and with memory $m \in \mathcal{T}(\Gamma)$ iff $t \xrightarrow{\Delta^*} q(m)$, and the *language* $L(\mathcal{A}, q)$ and *memory language* $M(\mathcal{A}, q)$ of \mathcal{A} in state q are respectively defined by:

$$\begin{aligned} L(\mathcal{A}, q) &= \{t \mid \exists m \in \mathcal{T}(\Gamma), t \xrightarrow{\Delta^*} q(m)\} \\ M(\mathcal{A}, q) &= \{m \mid \exists t \in \mathcal{T}(\Sigma), t \xrightarrow{\Delta^*} q(m)\}. \end{aligned}$$

The language of \mathcal{A} is the union of languages of \mathcal{A} in its final states, denoted: $L(\mathcal{A}) = \bigcup_{q \in Q_f} L(\mathcal{A}, q)$.

Visibility Condition. The above formalism is of course far too expressive. As there are no restrictions on the operation performed on memory by the rewrite rules, one can easily encode a Turing machine as a TAM. We shall now define a decidable restriction called *visibly tree automata with memory* (VTAM).

First, we consider only three main families (later divided into the subcategories defined in Figure 1) of operations on memory. We assume below a computation step at some position p of a term, where memories m_1, \dots, m_n have been reached at the positions immediately below p :

PUSH: the new current memory m is built with a symbol $h \in \Gamma_n$ *pushed* on the top of memories m_1, \dots, m_n : $f(q_1(m_1), \dots, q_n(m_n)) \rightarrow q(h(m_1, \dots, m_n))$. According to the terminology of [2], this corresponds to a *call* move in a program represented by an automaton.

POP: the new current memory is a subterm of one of the memories reached so far: $f(\dots, q_i(h(m'_1, \dots, m'_k)), \dots) \rightarrow q(m'_j)$. The top symbol h of m_i is also read. This corresponds to a function's *return* in a program.

We have here to split POP operations into four categories, depending on whether we pop on the memory at the left son or on the memory at the right son and on whether we get the left son of that memory or its right son.

INT (internal): the new current memory is one of the memories reached:

$$f(q_1(m_1), \dots, q_n(m_n)) \rightarrow q(m_i)$$

This corresponds to an internal operation (neither call nor return) in a function of a program.

Again, we need to split INT operations into three categories: one for constant symbols and two rules for binary symbols, depending on which of the two sons memories we keep.

Next, we adhere to the *visibility* condition of [2]. The idea behind this restriction, which was already in [16], is that the symbol read by an automaton (in a term in our case and [1], in a word in the case of [2]) corresponds to an instruction of a program, and hence belongs to one of the three above families (call, return or internal). Indeed, the effect of the execution of a given instruction on the current program state (a stack for [2] or a tree in our case) will always be in the same family. In other words, in this context, the family of the memory operations performed by a transition is completely determined by the function symbol read.

Let us assume from now on for the sake of simplicity the following restriction on the arity of symbols:

PUSH	a		$\rightarrow q(c)$	$a \in \Sigma_{\text{PUSH}}$
PUSH	$f(q_1(y_1),$	$q_2(y_2))$	$\rightarrow q(h(y_1, y_2))$	$f \in \Sigma_{\text{PUSH}}$
POP ₁₁	$f(q_1(h(y_{11}, y_{12})),$	$q_2(y_2))$	$\rightarrow q(y_{11})$	$f \in \Sigma_{\text{POP}_{11}}$
	$f(q_1(\perp),$	$q_2(y_2))$	$\rightarrow q(\perp)$	
POP ₁₂	$f(q_1(h(y_{11}, y_{12})),$	$q_2(y_2))$	$\rightarrow q(y_{12})$	$f \in \Sigma_{\text{POP}_{12}}$
	$f(q_1(\perp),$	$q_2(y_2))$	$\rightarrow q(\perp)$	
POP ₂₁	$f(q_1(y_1),$	$q_2(h(y_{21}, y_{22})))$	$\rightarrow q(y_{21})$	$f \in \Sigma_{\text{POP}_{21}}$
	$f(q_1(y_1),$	$q_2(\perp))$	$\rightarrow q(\perp)$	
POP ₂₂	$f(q_1(y_1),$	$q_2(h(y_{21}, y_{22})))$	$\rightarrow q(y_{22})$	$f \in \Sigma_{\text{POP}_{22}}$
	$f(q_1(y_1),$	$q_2(\perp))$	$\rightarrow q(\perp)$	
INT ₀	a		$\rightarrow q(\perp)$	$a \in \Sigma_{\text{INT}_0}$
INT ₁	$f(q_1(y_1),$	$q_2(y_2))$	$\rightarrow q(y_1)$	$f \in \Sigma_{\text{INT}_1}$
INT ₂	$f(q_1(y_1),$	$q_2(y_2))$	$\rightarrow q(y_2)$	$f \in \Sigma_{\text{INT}_2}$

where $q_1, q_2, q \in Q$, y_1, y_2 are distinct variables of \mathcal{X} , $c \in \Gamma_2$, $h \in \Gamma_2$.

Figure 1: VTAM transition categories.

All the symbols of Σ and Γ have either arity 0 or 2.

This is not a real restriction, and the results of this paper can be extended straightforwardly to the case of function symbols with other arities. The signature Σ is partitioned in eight subsets:

$$\Sigma = \Sigma_{\text{PUSH}} \uplus \Sigma_{\text{POP}_{11}} \uplus \Sigma_{\text{POP}_{12}} \uplus \Sigma_{\text{POP}_{21}} \uplus \Sigma_{\text{POP}_{22}} \uplus \Sigma_{\text{INT}_0} \uplus \Sigma_{\text{INT}_1} \uplus \Sigma_{\text{INT}_2}$$

The eight corresponding categories of transitions (transitions of the same category perform the same kind of operation on the memory) are defined formally in Figure 1. In this figure, one constant symbol has a particular role:

\perp is a special constant symbol in Γ , used to represent an empty memory.

Note that there are three categories for INT, INT₀ is for constant symbols and INT₁, INT₂ are for binary symbols and differ according to the memory which is kept. Similarly, there are four variants of POP transitions, POP₁₁, ..., POP₂₂. Moreover, each POP rule has a variant, which reads an empty memory (*i.e.* the symbol \perp).

Definition 2.2. A *visibly tree automaton with memory* (or VTAM for short) on Σ is a TAM (Γ, Q, Q_f, Δ) such that every rule of Δ belongs to one of the above categories PUSH, POP₁₁, POP₁₂, POP₂₁, POP₂₂, INT₀, INT₁, INT₂.

2.2. Expressiveness, Comparison. Standard bottom-up tree automata are particular cases of VTAM (simply assume all the symbols of the signature in INT₀ or INT₁).

Now, let us try to explain more precisely the relation with the visibly pushdown languages of [2], when considering finite word languages.

If the stack is empty in any accepting configuration of some finite word pushdown automaton \mathcal{A} , then it is easy to compute a pushdown automaton $\tilde{\mathcal{A}}$, which accepts the reverses (mirror images) of the words accepted by \mathcal{A} . Moreover, if \mathcal{A} is a visibly pushdown automaton, then $\tilde{\mathcal{A}}$ is also a visibly pushdown automaton: it suffices to exchange the push and pop symbols.

For pushdown word languages, there is a well-known lemma showing that the recognition by final state is equivalent to the recognition by empty stack. This equivalence however requires ϵ -transitions to empty the stack when a final state is reached. There are however no ϵ -transitions in visibly pushdown automata. So, if we consider for instance the language of words $w \in \{a, b\}^*$ such that any prefix of w contains more a than b 's, it is recognized by a visibly pushdown automaton. While, if we consider the mirror image (all suffixes contain more a 's than b 's), it is not recognized by a visibly pushdown automaton.

In conclusion, as long as visibility is relevant, the way the automaton is moving is also relevant. This applies of course to trees as well: there is a difference between top-down and bottom-up recognition.

Now, if we encode a word as a tree on a unary alphabet, starting from right to left, VTAM generalize visibly pushdown automata: moving bottom-up in the tree corresponds to moving left-right in the word.

VPTA transitions and VPTL are defined in [5] in the same formalism (rewrite rules) as in Figure 1, except that the rules are oriented in the other direction (top-down computations) and the memory contains a word, i.e. terms built with unary function symbols and one constant (empty stack).

As sketched above, since the automata of [5] work top-down, a language can be recognized by a VTAM (which works bottom-up) and not by a VPTL. As a typical example, consider the trees containing only unary symbols a, b and a constant 0 and such that all subterms contain more a 's than b 's.

But the converse is also true: there are similarly languages that are recognized by VPTA and not by VTAM (and there, constraints cannot help!)

Now, if we consider a slight modification of VPTA, in which the automata work bottom-up (simply change the direction of transition rules), it is not clear that good properties (closure and decision) are preserved since, now, we get equality tests between memory contents, increasing the original expressive power; when going top-down we always duplicate the memory content and send one copy to each son, while going bottom-up we may have different memory contents at two brother positions.

2.3. Determinism. A VTAM \mathcal{A} is said *complete* if every term of $\mathcal{T}(\Sigma)$ belongs to $L(\mathcal{A}, q)$ for at least one state $q \in Q$. Every VTAM can be completed (with a polynomial overhead) by the addition of a trash state. Hence, we shall consider from now on only complete VTAM.

A VTAM $\mathcal{A} = (\Gamma, Q, Q_f, \Delta)$ is said *deterministic* iff:

- for all $a \in \Sigma_{\text{INT}_0}$ there is at most one rule in Δ with left-member a ,
- for all $f \in \Sigma_{\text{PUSH}} \cup \Sigma_{\text{INT}_1} \cup \Sigma_{\text{INT}_2}$, for all $q_1, q_2 \in Q$, there is at most one rule in Δ with left-member $f(q_1(y_1), q_2(y_2))$,
- for all $f \in \Sigma_{\text{POP}_{11}} \cup \Sigma_{\text{POP}_{12}}$ (respectively $\Sigma_{\text{POP}_{21}} \cup \Sigma_{\text{POP}_{22}}$), for all $q_1, q_2 \in Q$ and all $h \in \Gamma$, there is at most one rule in Δ with left-member $f(q_1(h(y_{11}, y_{12})), q_2(y_2))$ (respectively $f(q_1(y_1), q_2(h(y_{21}, y_{22})))$).

Theorem 2.3. *For every VTAM $\mathcal{A} = (\Gamma, Q, Q_f, \Delta)$ there exists a deterministic VTAM $\mathcal{A}^{\text{det}} = (\Gamma^{\text{det}}, Q^{\text{det}}, Q_f^{\text{det}}, \Delta^{\text{det}})$ such that $L(\mathcal{A}) = L(\mathcal{A}^{\text{det}})$, where $|Q^{\text{det}}|$ and $|\Gamma^{\text{det}}|$ both are $O(2^{|Q|^2})$.*

Proof. We follow the technique of [2] for the determinization of visibly pushdown automata: we do a subset construction and postpone the application (to the memory) of PUSH rules, until a matching POP is met. The construction of [2] is extended in order to handle the branching structure of the term read and of the memory.

With the visibility condition, for each symbol read, only one kind of memory operation is possible. This permits a uniform construction of the rules of \mathcal{A}^{det} for each symbol of Σ . As we shall see below, \mathcal{A}^{det} does not need to keep track of the contents of memory (of \mathcal{A}) during its computation, it only needs to memorize information on the reachability of states of \mathcal{A} , following the path (in the term read) from the position of the PUSH symbol which has pushed the top symbol of the current memory (let us call it the *last-memory-push-position*) to the current position in the term. We let :

$$Q^{det} := \{0, 1\} \times \mathcal{P}(Q) \times \mathcal{P}(Q^2)$$

Q_f^{det} is the subset of states whose second component contains a final state of Q_f . The first component is a flag indicating whether the memory is currently empty (value 0) or not (value 1). The second component is the subset of states of Q that \mathcal{A} can reach at current position, and the third component is a binary relation on Q which contains (q, q') iff starting from a state q and memory m at the last-memory-push-position, \mathcal{A} can reach the current position in state q' , and with the same memory m . We consider memory symbols made of pairs of states and PUSH symbols:

$$\Gamma^{det} := (Q^{det})^2 \times (\Sigma_{PUSH})$$

The components of a symbol $p \in \Gamma^{det}$ refer to the transition who pushed p : the first and second components of p are respectively the left and right initial states of the transition and the third component is the symbol read.

The transition rules of Δ^{det} are given below, according to the symbol read.

INT. For every i and for every $f \in \Sigma_{INT_i}$, we have the following rules in Δ^{det} :

$$f(\langle b_1, R_1, S_1 \rangle(y_1), \langle b_2, R_2, S_2 \rangle(y_2)) \rightarrow \langle b_1, R, S \rangle(y_1)$$

where $R := \{q \mid \exists q_1 \in R_1, q_2 \in R_2, f(q_1(y_1), q_2(y_2)) \rightarrow q(y_1) \in \Delta\}$, and S is the update of S_1 according to the INT₁-transitions of Δ , when $b_1 = 1$ (the case $b_1 = 0$ is similar):

$$S := \{(q, q') \mid \exists q_1 \in Q, q_2 \in R_2, (q, q_1) \in S_1 \text{ and } f(q_1(y_1), q_2(y_2)) \rightarrow q'(y_1) \in \Delta\}.$$

The case $f \in \Sigma_{INT_2}$ is similar.

PUSH. For every $f \in \Sigma_{PUSH}$, we have the following rules in Δ^{det} :

$$f(\langle b_1, R_1, S_1 \rangle(y_1), \langle b_2, R_2, S_2 \rangle(y_2)) \rightarrow \langle 1, R, Id_Q \rangle(p(y_1, y_2))$$

where $R := \{q \mid \exists q_1 \in R_1, q_2 \in R_2, h \in \Gamma, f(q_1(y_1), q_2(y_2)) \rightarrow q(h(y_1, y_2)) \in \Delta\}$, $Id_Q := \{(q, q) \mid q \in Q\}$ is used to initialize the memorization of state reachability from the position of the symbol f , and $p := \langle \langle b_1, R_1, S_1 \rangle, \langle b_2, R_2, S_2 \rangle, f \rangle$. Note that the two states reached just below the position of application of this rule are pushed on the top of the memory. They will be used later in order to update R and S when a matching POP symbol is read.

POP. For every $f \in \Sigma_{\text{POP}_{11}}$, we have the following rules in Δ^{det} :

$$f(\langle b_1, R_1, S_1 \rangle(H(y_{11}, y_{12})), \langle b_2, R_2, S_2 \rangle(y_2)) \rightarrow \langle b, R, S \rangle(y_{11})$$

where $H = \langle Q_1, Q_2, g \rangle$, with $Q_1 = \langle b'_1, R'_1, S'_1 \rangle \in Q^{det}$, $Q_2 = \langle b'_2, R'_2, S'_2 \rangle \in Q^{det}$.

$$\begin{aligned} b &= b'_1 \\ R &= \left\{ q \left| \begin{array}{l} \exists q'_1 \in R'_1, q'_2 \in R'_2, (q_0, q_1) \in S_1, q_2 \in R_2, h \in \Gamma, g(q'_1(y_1), q'_2(y_2)) \rightarrow \\ q_0(h(y_1, y_2)) \in \Delta, f(q_1(h(y_{11}, y_{12})), q_2(y_2)) \rightarrow q(y_{11}) \in \Delta \end{array} \right. \right\} \\ S &= \left\{ (q, q') \left| \begin{array}{l} \exists q'_1 \in S'_1(q), q'_2 \in R'_2, (q_0, q_1) \in S_1, q_2 \in R_2, h \in \Gamma, g(q'_1(y_1), q'_2(y_2)) \rightarrow \\ \rightarrow q_0(h(y_1, y_2)) \in \Delta, f(q_1(h(y_{11}, y_{12})), q_2(y_2)) \rightarrow q'(y_{11}) \in \Delta \end{array} \right. \right\} \end{aligned}$$

When a POP symbol is read, the top symbol of the memory, which is popped, contains the states reached just before the application of the matching PUSH. We use this information in order to update $\langle b_1, R_1, S_1 \rangle$ and $\langle b_2, R_2, S_2 \rangle$ to $\langle b, R, S \rangle$.

The cases $f \in \Sigma_{\text{POP}_{12}}$, $f \in \Sigma_{\text{POP}_{21}}$, $f \in \Sigma_{\text{POP}_{22}}$ are similar.

The above constructions ensure the three invariants stated above, after the definition of Q^{det} and corresponding to the three components of these states. It follows that $L(\mathcal{A}) = L(\mathcal{A}^{det})$. \square

2.4. Closure Properties. The tree automata with one memory of [7] are closed under union but not closed under intersection and complement (even their version without constraints). The visibility condition makes possible these closures for VTAM.

Theorem 2.4. *The class of tree languages of VTAM is closed under Boolean operations. One can construct VTAM for union, intersection and complement of given VTAM languages whose sizes are respectively linear, quadratic and exponential in the size of the initial VTAM.*

Proof. Let $\mathcal{A}_1 = (\Gamma_1, Q_1, Q_{f,1}, \Delta_1)$ and $\mathcal{A}_2 = (\Gamma_2, Q_2, Q_{f,2}, \Delta_2)$ be two VTAM on Σ . We assume wlog that Q_1 and Q_2 are disjoint.

For the union of the languages of \mathcal{A}_1 and \mathcal{A}_2 , we construct a VTAM \mathcal{A}_\cup whose memory signature, state set, final state set and rules set are the union of the respective memory signatures, state sets, final state sets and rules sets of the two given VTAM. We have $L(\mathcal{A}_\cup) = L(\mathcal{A}_1) \cup L(\mathcal{A}_2)$.

$$\mathcal{A}_\cup = (\Gamma_1 \cup \Gamma_2, Q_1 \cup Q_2, Q_{f,1} \cup Q_{f,2}, \Delta_1 \cup \Delta_2)$$

For the intersection of the languages of \mathcal{A}_1 and \mathcal{A}_2 , we construct a VTAM \mathcal{A}_\cap whose memory signature, state set and final state set are the Cartesian product of the respective memory signatures, state sets and final state sets of the two given VTAM.

$$\mathcal{A}_\cap = (\Gamma_1 \times \Gamma_2, Q_1 \times Q_2, Q_{f,1} \times Q_{f,2}, \Delta_\cap)$$

The rule set Δ_\cap of the intersection VTAM \mathcal{A}_\cap is obtained by "product" of rules of the two given VTAM with same function symbols. The product of rules means Cartesian products of the respective states and memory symbols pushed or popped. More precisely, Δ_\cap is the smallest set of rules such that:

- if Δ_1 contains $f(q_{11}(y_1), q_{12}(y_2)) \rightarrow q_1(h_1(y_1, y_2))$ and Δ_2 contains $f(q_{21}(y_1), q_{22}(y_2)) \rightarrow q_2(h_2(y_1, y_2))$, for some $f \in \Sigma_{\text{PUSH}}$, then Δ_\cap contains $f(\langle q_{11}, q_{21} \rangle(y_1), \langle q_{12}, q_{22} \rangle(y_2)) \rightarrow \langle q_1, q_2 \rangle(\langle h_1, h_2 \rangle(y_1, y_2))$.

- if Δ_1 contains $f(q_{11}(h_1(y_{11}, y_{12})), q_{12}(y_2)) \rightarrow q_1(y_{11})$ and Δ_2 contains $f(q_{21}(h_2(y_{11}, y_{12})), q_{22}(y_2)) \rightarrow q_2(y_{11})$ for some $f \in \Sigma_{\text{POP}_{11}}$, then Δ_\cap contains $f(\langle q_{11}, q_{2,1} \rangle(\langle h_1, h_2 \rangle(y_{11}, y_{12})), \langle q_{12}, q_{2,2} \rangle(y_2)) \rightarrow \langle q_1, q_2 \rangle(y_{11})$
- similarly for POP_{12} , POP_{21} and POP_{22}
- if Δ_1 contains $f(q_{11}(y_1), q_{21}(y_2)) \rightarrow q_1(y_1)$ and Δ_2 contains $f(q_{21}(y_1), q_{22}(y_2)) \rightarrow q_2(y_1)$ for some $f \in \Sigma_{\text{INT}_1}$, then Δ_\cap contains $f(\langle q_{11}, q_{2,1} \rangle(y_1), \langle q_{12}, q_{2,2} \rangle(y_2)) \rightarrow \langle q_1, q_2 \rangle(y_1)$
- and similarly for INT_2 , INT_0 .

We have then $L(\mathcal{A}_\cap) = L(\mathcal{A}_1) \cap L(\mathcal{A}_2)$. Note that the above product construction for \mathcal{A}_\cap is possible only because the visibility condition ensures that two rules with the same function symbol in left-side will have the same form. Hence we can synchronize memory operations on the same symbols.

For the complement, we use the construction of Theorem 2.3 and a completion (this operation preserves determinism), and take the complement of the final state set of the VTAM obtained. \square

2.5. Decision Problems. Every VTAM is a particular case of tree automaton with one memory of [7]. Since the emptiness problem (whether the language accepted is empty or not) is decidable for this latter class, it is also decidable for VTAM. However, whereas this problem is EXPTIME-complete for the automata of [7], it is only PTIME for VTAM.

Theorem 2.5. *The emptiness problem is PTIME-complete for VTAM.*

Proof. Assume given a VTAM $\mathcal{A} = (\Gamma, Q, Q_f, \Delta)$. By definition, for each state $q \in Q$, the language $L(\mathcal{A}, q)$ is empty iff the memory language $M(\mathcal{A}, q)$ is empty. For each state q , we introduce a predicate symbol P_q and we construct Horn clauses in such a way that $P_q(m)$ belongs to the least Herbrand model of this set of clauses, iff the configuration with state q and memory m is reachable by the automaton (i.e. $m \in M(\mathcal{A}, q)$).

For such a construction (already given in [7]), we simply forget the function symbol, associating to a transition rule $f(q_1(m_1), q_2(m_2)) \rightarrow q(m)$ the Horn clause $P_{q_1}(m_1), P_{q_2}(m_2) \Rightarrow P_q(m)$. Then, according to the restrictions in Definition 2.2, we get only Horn clauses of one of the following forms:

$$\begin{aligned}
& \Rightarrow P_q(c) \\
P_{q_1}(y_1), P_{q_2}(y_2) & \Rightarrow P_q(h(y_1, y_2)) \\
P_{q_1}(h(y_{11}, y_{12})), P_{q_2}(y_2) & \Rightarrow P_q(y_{11}) \\
P_{q_1}(h(y_{11}, y_{12})), P_{q_2}(y_2) & \Rightarrow P_q(y_{12}) \\
P_{q_1}(\perp), P_{q_2}(y_2) & \Rightarrow P_q(\perp) \\
P_{q_1}(y_1), P_{q_2}(y_2) & \Rightarrow P_q(y_1)
\end{aligned}$$

where all the variables are distinct. Such clauses belong to the class \mathcal{H}_3 of [19], for which it is proved in [19] that emptiness is decidable in cubic time. It follows that emptiness of VTAM is decidable in cubic time.

Hardness for PTIME follows from the PTIME-hardness of emptiness of finite tree automata [8]. \square

Another proof relying on similar techniques, but for a more general result, will be stated in Lemma 3.7 and can be found in Appendix 5.

The *universality* is the problem of deciding whether a given automaton recognizes all ground terms. *Inclusion* refers to the problem of deciding the inclusion between the respective languages of two given automata.

Corollary 2.6. *The universality and inclusion problem are EXPTIME-complete for VTAM.*

Proof. A VTAM \mathcal{A} is universal iff the language of its complement automaton $\overline{\mathcal{A}}$ is empty, and $L(\mathcal{A}_1) \subseteq L(\mathcal{A}_2)$ iff $L(\mathcal{A}_1) \cap L(\overline{\mathcal{A}_2}) = \emptyset$. With the bounds given in Theorem 2.4 these problems can be decided in EXPTIME for VTAM (these operations require a determinization of a given VTAM first).

The EXPTIME-hardness follows from the corresponding property of finite tree automata (see [8] for instance). \square

The *membership* problem is, given a term t and an automaton \mathcal{A} , to know whether t is accepted by \mathcal{A} .

Corollary 2.7. *The membership problem is decidable in PTIME for VTAM.*

Proof. Given a term t we can build a VTAM \mathcal{A}_t which recognizes exactly the language $\{t\}$. The intersection of \mathcal{A}_t with the given VTAM \mathcal{A} recognizes a non empty language iff t belongs to the language of \mathcal{A} . \square

3. VISIBLY TREE AUTOMATA WITH MEMORY AND CONSTRAINTS

In the late eighties, some models of tree recognizers were obtained by adding equality and disequality constraints in transitions of tree automata. They have been proposed in order to solve problems with term rewrite systems or constraints systems with non-linear patterns (terms with multiple occurrences of the same variable). The tree automata of [4] for instance can perform equality and disequality tests between subterms located at brother positions of the input term.

In the case of tree automata with memory, constraints are applied to the memory contents. Indeed, each bottom-up computation step starts with two states and two memories (and ends with one state and one memory), and therefore, it is possible to compare the contents of these two memories, with respect to some binary relation.

We state first the general definition of visibly tree automata with constraints on memories (Section 3.1), then give sufficient conditions on the binary relation for the emptiness decidability (Section 3.2) and show that, if in general regular binary relations do not satisfy these conditions (and indeed, the corresponding class of constrained VTAM has an undecidable emptiness problem, Section 3.3) some relevant examples do satisfy them. In particular, we study in Section 3.4.2 the case of VTAM with structural equality constraints. They enjoy not only decision properties but also good closure properties. Some relevant examples of tree languages recognized by constrained VTAM of this class are presented at the end of the section.

INT_1^R	$f_9(q_1(y_1), q_2(y_2))$	$\xrightarrow{y_1 R y_2}$	$q(y_1)$	$f_9 \in \Sigma_{\text{INT}_1^R}$
INT_2^R	$f_{10}(q_1(y_1), q_2(y_2))$	$\xrightarrow{y_1 R y_2}$	$q(y_2)$	$f_{10} \in \Sigma_{\text{INT}_2^R}$
INT_1^R	$f_{11}(q_1(y_1), q_2(y_2))$	$\xrightarrow{y_1 \neg R y_2}$	$q(y_1)$	$f_{11} \in \Sigma_{\text{INT}_1^R}$
INT_2^R	$f_{12}(q_1(y_1), q_2(y_2))$	$\xrightarrow{y_1 \neg R y_2}$	$q(y_2)$	$f_{12} \in \Sigma_{\text{INT}_2^R}$

Figure 2: New transition categories for $\text{VTAM}_{\neg R}^R$.

3.1. Definitions. Assume given a fixed equivalence relation R on $\mathcal{T}(\Gamma)$. We consider now two new categories for the symbols of Σ : INT_1^R and INT_2^R , in addition to the eight previous categories of page 6. The new categories correspond to the constrained versions of the transition rules INT_1 and INT_2 presented in Figure 2. The constraint $y_1 R y_2$ in the two first rules of Figure 2 is called *positive* and the constraint $y_1 \neg R y_2$ in the two last rules is called *negative*.

We shall not extend the rules **PUSH** and **POP** with constraints for some reasons explained in section 3.5. A ground term t rewrites to s by a constrained rule $f(q_1(y_1), q_2(y_2)) \xrightarrow{y_1 c y_2} r$ (where c is either R or $\neg R$) if there exists a position p of t and a substitution σ such that $t|_p = \ell\sigma$, $y_1\sigma c y_2\sigma$ and $s = t[r\sigma]_p$.

For example, if R is term equality, the transition is performed only when the memory contents are identical.

Definition 3.1. A *visibly tree automaton with memory and constraints* ($\text{VTAM}_{\neg R}^R$) on a signature Σ is a tuple $(\Gamma, R, Q, Q_f, \Delta)$ where Γ , Q , Q_f are defined as for **TAM**, R is an equivalence relation on $\mathcal{T}(\Gamma)$ and Δ is a set of rewrite rules in one of the above categories: **PUSH**, **POP**₁₁, **POP**₁₂, **POP**₂₁, **POP**₂₂, **INT**₀, **INT**₁, **INT**₂, INT_1^R , INT_2^R .

We let VTAM^R be the subclass of $\text{VTAM}_{\neg R}^R$ with positive constraints only. The acceptance of terms of $\mathcal{T}(\Sigma)$ and languages of term and memories are defined and denoted as in Section 2.1.

The definition of *complete* $\text{VTAM}_{\neg R}^R$ is the same as for **VTAM**. As for **VTAM**, every $\text{VTAM}_{\neg R}^R$ can be completed (with a polynomial overhead) by the addition of a trash state q_\perp . The only subtle difference concerns the constrained rules: for every $f_9 \in \text{INT}_1^R$ and every states q_1, q_2 ,

- if there is a rule $f_9(q_1(y_1), q_2(y_2)) \xrightarrow{y_1 R y_2} q(y_1)$ and no rule of the form $f_9(q_1(y_1), q_2(y_2)) \xrightarrow{y_1 \neg R y_2} q'(y_1)$, then we add $f_9(q_1(y_1), q_2(y_2)) \xrightarrow{y_1 \neg R y_2} q_\perp(y_1)$,
- if there is a rule $f_9(q_1(y_1), q_2(y_2)) \xrightarrow{y_1 \neg R y_2} q(y_1)$ and no rule of the form $f_9(q_1(y_1), q_2(y_2)) \xrightarrow{y_1 R y_2} q'(y_1)$, then we add $f_9(q_1(y_1), q_2(y_2)) \xrightarrow{y_1 R y_2} q_\perp(y_1)$,
- if there is no rule of the form $f_9(q_1(y_1), q_2(y_2)) \xrightarrow{y_1 R y_2} q(y_1)$ or $f_9(q_1(y_1), q_2(y_2)) \xrightarrow{y_1 \neg R y_2} q'(y_1)$, then we add $f_9(q_1(y_1), q_2(y_2)) \xrightarrow{y_1 R y_2} q_\perp(y_1)$ and $f_9(q_1(y_1), q_2(y_2)) \xrightarrow{y_1 \neg R y_2} q_\perp(y_1)$.

The definition of *deterministic* $\text{VTAM}_{\neg R}^R$ is based on the same conditions as for **VTAM** for the function symbols in categories of **PUSH**₀, **PUSH**, **POP**₁₁, ..., **POP**₂₂, **INT**₁, **INT**₂. For the function symbols of INT_1^R , INT_2^R , we have the following condition: for all $f \in \Sigma_{\text{INT}_1^R} \cup \Sigma_{\text{INT}_2^R}$ for all $q_1, q_2 \in Q$, there are at most two rules in Δ with left-member $f(q_1(y_1), q_2(y_2))$, and if there are two, one has a positive constraint and the other has a negative constraint.

We will see in Section 3.4 a subclass of $\text{VTAM}_{\neg R}^R$ that can be determinized (when R is structural equality) and another one that cannot (when R is syntactic equality).

3.2. Sufficient Conditions for Emptiness Decision. We propose here a generic theorem ensuring emptiness decision for $\text{VTAM}_{\neg R}^R$. The idea of this theorem is that under some condition on R , the transition rules with negative constraints can be eliminated.

Theorem 3.2. *Let R be an equivalence relation satisfying these two properties:*

- i. *for every automaton \mathcal{A} of VTAM^R and for every state q of \mathcal{A} , the memory language $M(\mathcal{A}, q)$ is effectively a regular tree language,*
- ii. *for every term $m \in \mathcal{T}(\Gamma)$, the cardinality of the equivalence class of m for R is finite and its elements can be enumerated.*

Then the emptiness problem is decidable for $\text{VTAM}_{\neg R}^R$.

Proof. The proof relies on the following Lemma 3.3 which states that the negative constraints in $\text{VTAM}_{\neg R}^R$ can be eliminated, while preserving the memory languages. The elimination can be done thanks to the condition *ii*, by replacement of the rules of $\text{INT}_1^{\neg R}$ and $\text{INT}_2^{\neg R}$ by rules of INT_1^R and INT_2^R .

Next, we can use *i* in order to decide emptiness for the VTAM^R obtained by elimination of negative constraints. Indeed, for all states q of \mathcal{A} , by definition, $L(\mathcal{A}, q)$ is empty iff $M(\mathcal{A}, q)$ is empty. \square

Lemma 3.3. *Let R satisfy the hypotheses *i* and *ii* of Theorem 3.2, and let $\mathcal{A} = (\Gamma, R, Q, Q_f, \Delta)$ be a $\text{VTAM}_{\neg R}^R$. There exists a VTAM^R $\mathcal{A}^+ = (\Gamma, R, Q^+, Q_f, \Delta^+)$ such that $Q \subseteq Q^+$, and for each $q \in Q$, $M(\mathcal{A}^+, q) = M(\mathcal{A}, q)$.*

Proof. The construction of \mathcal{A}^+ is by induction on the number n of rules with negative constraints in Δ and uses the bound on the size of equivalence classes, condition *ii* of the theorem.

The result is immediate if $n = 0$.

We assume that the result is true for $n - 1$ rules, and show that we can get rid of a rule of Δ with negative constraints (and replace it with rules unconstrained or with positive constraints). Let us consider one such rule:

$$f(q_1(y_1), q_2(y_2)) \xrightarrow{y_1 \neg R y_2} q(y_1) \quad (3.1)$$

We show that, under the induction hypothesis, we have the following lemma which will be used below in order to get rid of the rule (3.1).

Lemma 3.4. *Given $m_1, \dots, m_k \in M(\mathcal{A}, q_2)$, it is effectively decidable whether $M(\mathcal{A}, q_2) \setminus \{m_1, \dots, m_k\}$ is empty or not and, in case it is not empty, we can effectively build a m_{k+1} in this set.*

Proof. Let $[m_i]_R$ denote the equivalence class of m_i . By condition *ii*, every $[m_i]_R$ is finite, hence for each $i \leq k$, we can build a VTAM \mathcal{A}_i with a state p_i such that $M(\mathcal{A}_i, p_i)$ is the complement of $[m_i]_R$. We add all the rules of \mathcal{A}_i to \mathcal{A} , obtaining \mathcal{A}' (we assume that the state sets of $\mathcal{A}_1, \dots, \mathcal{A}_k, \mathcal{A}$ are disjoint, and that the states of $\mathcal{A}_1, \dots, \mathcal{A}_k$ are not final in \mathcal{A}').

Since R is an equivalence relation, we have:

$$y_1 \neg R m_i \text{ iff } y_1 \notin [m_i]_R \text{ iff } \exists y_2 \notin [m_i]_R, y_1 R y_2$$

Hence, if $y_2 = m_i$ is a witness for the rule (3.1), then we can apply instead a rule:

$$f(q_1(y_1), p_i(y_2)) \xrightarrow{y_1 R y_2} q(y_1) \quad (3.2)$$

Then we add to \mathcal{A}' the rules (3.2) as above and obtain \mathcal{A}'' . It can be shown that $M(\mathcal{A}'', q_2) = M(\mathcal{A}, q_2)$.

Let m_{k+1} be a term of $M(\mathcal{A}'', q_2) \setminus \{m_1, \dots, m_k\}$ of minimal size (if one exists). This term m_{k+1} can be created in a run of \mathcal{A}'' which does not use the rule (3.1). Otherwise, the witness for y_2 in the application of this rule would be a term of $M(\mathcal{A}'', q_2) \setminus \{m_1, \dots, m_k\}$ smaller than m_{k+1} (it cannot be one of $\{m_1, \dots, m_k\}$ because for these particular values of y_2 , we assume the application of (3.2)). It follows that $m_{k+1} \in M(\mathcal{A}'' \setminus (3.1), q_2)$. This automaton $\mathcal{A}_1 = \mathcal{A}'' \setminus (3.1)$ has $n - 1$ rules with negative constraints. Hence, by induction hypothesis, there is a VTAM^R \mathcal{A}_1^+ with m_{k+1} in its memory language $M(\mathcal{A}_1^+, q_2)$. By condition *i*, this language is regular and we can build m_{k+1} from a TA for this language. \square

Now, let us come back to the proof that we can replace rule (3.1), while preserving the memory languages.

If $M(\mathcal{A}, q_2) = \emptyset$ (which can be effectively decided according to lemma 3.4) then the rule (3.1) is useless and can be removed from \mathcal{A} without changing its memory language. Note that the condition $M(\mathcal{A}, q_2) = \emptyset$ is decidable because by hypothesis *i*, $M(\mathcal{A}, q_2)$ is regular.

Otherwise, let $m_1 \in M(\mathcal{A}, q_2)$ be built with Lemma 3.4 and let N_1 be the cardinal of the equivalence class $[m_1]_R$. We apply N_1 times the construction of Lemma 3.4. There are three cases:

- (1) if we find more than N_1 terms in $M(\mathcal{A}, q_2)$, then one of them, say m_k is not in $[m_1]_R$. Then (3.1) is useless for the point of view of memory languages: whatever value for y_1 , we know a $y_2 \in M(\mathcal{A}, q_2)$ which permits to fire the rule. Indeed, if $y_1 \in [m_1]_R$, then we can choose $y_2 = m_k$, and otherwise we choose $y_2 = m_1$. Hence (3.1) can be replaced without changing the memory language by:

$$f(q_1(y_1), q_0(y_2)) \rightarrow q(y_1) \quad (3.3)$$

where q_0 is any state of \mathcal{A} such that $M(\mathcal{A}, q_0) \neq \emptyset$. We can then apply the induction hypothesis to the VTAM^R_{-R} obtained.

- (2) if we find less than N_1 terms in $M(\mathcal{A}, q_2)$, but one is not in $[m_1]_R$. The case is the same as above.
- (3) if we find less than N_1 terms in $M(\mathcal{A}, q_2)$, all in $[m_1]_R$, it means that one of the applications of Lemma 3.4 was not successful, and hence that we have found all the terms of $M(\mathcal{A}, q_2)$. It follows that the rule (3.1) can be fired iff $y_1 \notin [m_1]_R$, i.e. there exists $y_2 \notin [m_1]_R$ such that $y_1 R y_2$. Hence, we can replace (3.1) by

$$f(q_1(y_1), p_1(y_2)) \xrightarrow{y_1 R y_2} q(y_1).$$

Then we can apply the induction hypothesis. \square

We present in Section 3.4 two examples of relations satisfying *i*. and *ii*.

3.3. Regular Tree Relations. We first consider the general case of $\text{VTAM}_{\neg R}^R$ where the equivalence R is based on an arbitrary regular binary relation on $\mathcal{T}(\Gamma)$. By regular binary relation, we mean a set of pairs of ground terms accepted by a tree automaton computing simultaneously in both terms of the pair. More formally, we use a coding of a pair of terms of $\mathcal{T}(\Sigma)$ into a term of $\mathcal{T}((\Sigma \cup \{\perp\})^2)$, where \perp is a new constant symbol (not in Σ). This coding is defined recursively by:

- $\otimes : \mathcal{T}(\Sigma) \cup \{\perp\} \times \mathcal{T}(\Sigma) \cup \{\perp\} \rightarrow \mathcal{T}((\Sigma \cup \{\perp\})^2)$
- for all $a, b \in \Sigma_0 \cup \{\perp\}$, $a \otimes b := \langle a, b \rangle$,
- for all $a \in \Sigma_0 \cup \perp$, $f \in \Sigma_2$, $t_1, t_2 \in \mathcal{T}(\Sigma)$, $f(t_1, t_2) \otimes a := \langle f, a \rangle(t_1 \otimes \perp, t_2 \otimes \perp)$ $a \otimes f(t_1, t_2) := \langle a, f \rangle(\perp \otimes t_1, \perp \otimes t_2)$,
- for all $f, g \in \Sigma_2$, $s_1, s_2, t_1, t_2 \in \mathcal{T}(\Sigma)$, $f(s_1, s_2) \otimes g(t_1, t_2) := \langle f, g \rangle(s_1 \otimes t_1, s_2 \otimes t_2)$.

Then, a binary relation $R \subseteq \mathcal{T}(\Sigma) \times \mathcal{T}(\Sigma)$ is called regular iff the set $\{s \otimes t \mid (s, t) \in R\}$ is regular. The above coding of pairs is unrelated to the product used in Theorem 2.4.

Theorem 3.5. *The membership problem for $\text{VTAM}_{\neg R}^R$ is NP-complete when R is a regular binary relation.*

Proof. Assume given a ground term $t \in \mathcal{T}(\Sigma)$ and a $\text{VTAM}_{\neg R}^R$ $\mathcal{A} = (\Gamma, R, Q, Q_f, \Delta)$. Because of the visibly condition, for every subterm s of t , we can compute in polynomial time in the size of s the shape denoted $\text{struct}(s)$, which is an abstraction of the memory reached when \mathcal{A} runs on s . More precisely, $\text{struct}(s)$ is an unlabeled tree, and every possible content of memory m reachable by \mathcal{A} in a computation $s \xrightarrow[\Delta]{*} q(m)$ is obtained by a labeling of the nodes of $\text{struct}(s)$ with symbols of Γ . Note that for all subterm s , the size of $\text{struct}(s)$ is smaller than the size of t .

Let us guess a decoration of every node of t with a state of Q and a labeling of $\text{struct}(s)$ (where s is the subterm of t at the given node), such that the root of t is decorated with a final state of Q_f . We can check in polynomial time whether this decoration represents a run of \mathcal{A} on t or not.

The NP-hardness is a consequence of Theorem 3.9, which applies to the particular case where R is the syntactic equality between terms. \square

Note that the NP algorithm works with every equivalence R based on a regular relation, but the the NP-hardness concerns only some cases of such relations. For instance, in Section 3.4, we will see one example of relation for which membership is NP-hard and another example for which it is in PTIME.

The class of $\text{VTAM}_{\neg R}^R$ when R is a binary regular tree relation constitutes a nice and uniform framework. Note however the condition *ii* of Theorem 3.2 is not always true in this case. Actually, this class is too expressive.

Theorem 3.6. *Given a regular binary relation R and an automaton \mathcal{A} in VTAM^R , the emptiness of $L(\mathcal{A})$ is undecidable.*

Proof. We reduce the blank accepting problem for a deterministic Turing machine \mathcal{M} . We encode configurations of \mathcal{M} as "right-combs" (binary trees) built with the tape and state symbols of \mathcal{M} , in Σ_{PUSH} (hence binary) and a constant symbol ε in Σ_{INT_0} . Let R be the regular relation which accepts all the pairs of configurations $c \otimes c'$ such that c' is a successor of c by \mathcal{M} . A sequence of configurations $c_0 c_1 \dots c_n$ (with $n \geq 1$) is encoded as a tree $t = f(c_0(f(c_1, \dots f(c_{n-1}, c_n)))$, where f is a binary symbol of $\Sigma_{\text{INT}_1}^R$.

We construct a $\text{VTAM}^R \mathcal{A}$ which accepts exactly the term-representations t of computation sequences of \mathcal{M} starting with the initial configuration c_0 of \mathcal{M} and ending with a final configuration c_n with blank tape. Following the type of the function symbols, the rules of \mathcal{A} will

- push all the symbols read in subterms of t corresponding to configurations,
- compare, with R , c_i and c_{i+1} (the memory contents in respectively the left and right branches) and store c_i in the memory, with a transition applied at the top of a subterm $f(c_i, f(c_{i+1}, \dots))$.

This way, \mathcal{A} checks that successive configurations in t correspond to transitions of \mathcal{M} , hence that the language of \mathcal{A} is not empty iff \mathcal{M} accepts the initial configuration c_0 . \square

3.4. Syntactic and Structural Equality and Disequality Constraints. We present now two examples of relations satisfying the conditions of Theorem 3.2: syntactic and structural term equality. The satisfaction of condition i will be proved with the help of the following crux Lemma.

Lemma 3.7. *Let R be a regular binary relation defined by a TA whose state set is $\{R_i \mid i = \{1..n\}\}$ and such that $\forall i, j \exists k, l, \forall x, y, z. xR_i y \wedge yR_j z \Leftrightarrow xR_k y \wedge xR_l z$. Let $\mathcal{A} = (\Gamma, R, Q, Q_f, \Delta)$ be a tree automaton with memory and constraints (not necessarily visibly). Then it is possible to compute in exponential time a finite tree automaton \mathcal{A}' , such that, for every state $q \in Q$, the language $M(\mathcal{A}, q)$ is the language accepted in some state of \mathcal{A}' .*

Proof. (Sketch) To prove this lemma, we first observe that the $M(\mathcal{A}, q)$ (for $q \in Q$) are actually the least sets that satisfies the following conditions (we assume here for simplicity that the non-constant symbols are binary and display only some of the implications; the others can be easily guessed):

$$\begin{aligned}
\forall x, y, z. \quad & x \in M(\mathcal{A}, q_1), y \in M(\mathcal{A}, q_2) \Rightarrow g(x, y) \in M(\mathcal{A}, q) \\
& \text{if there is a rule } f(q_1(x_1), q_2(x_2)) \rightarrow q(g(x_1, x_2)) \\
& g(x, y) \in M(\mathcal{A}, q_1), z \in M(\mathcal{A}, q_2) \Rightarrow x \in M(\mathcal{A}, q) \\
& \text{if there is a rule } f(q_1(g(x, y), q_2(z)) \rightarrow q(x) \\
& x \in M(\mathcal{A}, q_1), y \in M(\mathcal{A}, q_2), R(x, y) \Rightarrow x \in M(\mathcal{A}, q) \\
& \text{if there is a rule } f(q_1(x), q_2(y)) \xrightarrow{xRy} q(x) \\
& \dots
\end{aligned}$$

In terms of automata, this means that $M(\mathcal{A}, q)$ is a language recognized by a two-way alternating tree automaton with regular binary constraints. In other words, such languages are the least Herbrand model of a set of clauses of the form

$$\begin{array}{lll}
Q_1(y_1), Q_2(y_2), R(y_1, y_2) & \Rightarrow & Q_3(y_1) & \text{INT}_1, \text{INT}_2 \\
Q_1(y_1), Q_2(y_2) & \Rightarrow & Q_3(f(y_1, y_2)) & \text{PUSH} \\
& & \Rightarrow & Q_1(a) & \text{INT}_0 \\
Q_1(f(y_1, y_2)), Q_2(y_3) & \Rightarrow & Q_3(y_1) & \text{POP}_{11}, \text{POP}_{21} \\
Q_1(f(y_1, y_2)), Q_2(y_3) & \Rightarrow & Q_3(y_2) & \text{POP}_{12}, \text{POP}_{22}
\end{array}$$

The lemma then shows that languages that are recognized by two-way alternating tree automata with some particular regular constraints, are also recognized by a finite

tree automaton. This corresponds to classical reductions of two-way automata to one-way automata (see e.g [8], chapter 7, [13], or [12, 6] for the first relevant references). The idea of the reduction is to find shortcuts: moving up and down yields a move at the same level. Add such shortcuts as new rules, until getting a “complete set”. Then only keep the non-redundant rules: this yields a finite tree automaton. Such a procedure relies on the definitions of ordered strategies, redundancy and saturation (aka complete sets), which are classical notions in automated first-order theorem proving [13, 3, 20]. Indeed, formally, a “shortcut” must be a formula, which allows for smaller proofs than the proof using the two original rules. A *saturated set* corresponds to a set of formulas whose all shortcuts are already in the set.

The advantage of the clausal formalism is to enable an easy representation of the above shortcuts, as intermediary steps. Such shortcuts are clauses, but are not automata rules. Second, we may rely on completeness results for Horn clauses.

That is why, only for the proof of this lemma, which follows and extend the classical proofs adding some regular constraints, we switch to a first-order logic formalization. The complete proof can be found in Appendix 5. As in the classical proofs, we saturate the set of clauses by resolution with selection and eager splitting. This saturation terminates, and the set of clauses corresponding to finite tree automata transitions in the saturated set recognizes the language $M(\mathcal{A}, q)$, which is therefore regular. \square

The condition on R in the lemma allows to break chains such as $\exists x_1, \dots, x_n. xRx_1 \wedge x_1Rx_2 \wedge \dots \wedge x_nRy \wedge P(x, y)$, which would be a source of non-termination in the saturation procedure. We may indeed replace such chains by $\exists x_1, \dots, x_n. xRx_1x_1 \wedge xRx_2x_2 \wedge \dots \wedge xRx_nx_n \wedge xR_0y \wedge P(x, y)$, which can again be simplified into $\exists x_1. xSx_1 \wedge xR_0y \wedge P(x, y)$ where S is the intersection of R_1, \dots, R_n . Possible such intersections range in a finite set as the relation R is regular and the R_i s are states of the automaton accepting R .

Finally note that finding k, l in the lemma’s assumption can always be performed in an effective way since R is regular.

3.4.1. Syntactic Constraints. We first apply Lemma 3.7 to the class $\text{VTAM}_{\neq}^{\overline{=}}$ where $=$ denotes the equality between ground terms made of memory symbols. Note that it is a particular case of constrained VTAM_{-R}^R of the above section 3.3, since the term equality is a regular relation. The automata of the subclass with positive constraints only, $\text{VTAM}^{\overline{=}}$, are particular cases of tree automata with one memory of [7], and have therefore a decidable emptiness problem. We show below that $\text{VTAM}_{\neq}^{\overline{=}}$ fulfills the hypotheses of Theorem 3.2, and hence that the emptiness is also decidable for the whole class.

We can first verify that the relation $=$ checks the hypothesis of Lemma 3.7, hence the condition *i* of Theorem 3.2. Moreover, the relation $=$ obviously also checks the condition *ii* of Theorem 3.2.

Corollary 3.8. *The emptiness problem is decidable for $\text{VTAM}_{\neq}^{\overline{=}}$.*

A careful analysis of the proof of Theorem 3.2 permits to conclude to an EXPTIME complexity for this problem with $\text{VTAM}_{\neq}^{\overline{=}}$.

Theorem 3.9. *The membership problem is NP-complete for $\text{VTAM}_{\neq}^{\overline{=}}$.*

Proof. An NP algorithm is given in the proof of Theorem 3.5. For the NP-hardness, we use a logspace reduction of 3-SAT. Let us consider an instance of 3-SAT with n propositional

variables X_1, \dots, X_n and a conjunction of m clauses:

$$\bigwedge_{i=1}^m (\alpha_{i,1} \vee \alpha_{i,2} \vee \alpha_{i,3})$$

where every $\alpha_{i,j}$ is either a variable X_k ($k \leq n$) or a negation of variable $\neg X_k$. We assume wlog that every variable occurs at most once in a clause.

We consider an encoding t of the given instance as a term over the signature Σ containing the symbols: X_1, \dots, X_n (constants), id , $false$, \neg (unary) and \wedge and \vee (binary). The encoding is:

$$t := C_\wedge [C_\vee[\delta_{1,1}(X_1), \dots, \delta_{1,n}(X_n)], \dots, C_\vee[\delta_{m,1}(X_1), \dots, \delta_{m,n}(X_n)]]$$

where C_\wedge (resp. C_\vee) is a context built solely with \wedge (resp. \vee) and where every $\delta_{i,j}$ is either:

- $\delta_{i,j} = id$ (interpreted as the identity) if one of $\alpha_{i,1}, \alpha_{i,2}, \alpha_{i,3}$ is X_j ,
- $\delta_{i,j} = \neg$ if one of $\alpha_{i,1}, \alpha_{i,2}, \alpha_{i,3}$ is $\neg X_j$,
- $\delta_{i,j} = false$ (interpreted as the constant function returning $false$) if X_j does not occur in $\alpha_{i,1}, \alpha_{i,2}, \alpha_{i,3}$.

Now, let us partition the signature Σ with: $X_1, \dots, X_n, \vee \in \text{PUSH}$, $id, false, \neg \in \text{INT}_1$ and $\wedge \in \text{INT}_1^-$; and let consider the memory signature $\Gamma = \{0, 1, \vee\}$. We construct now a $\text{VTAM}^\# \mathcal{A} = (\Gamma, =, \{q_0, q_1\}, \{q_1\}, \Delta)$ whose transition will, intuitively:

- guess an assignment for each constant symbol X_k of t , by mean of a non-deterministic choice of one state q_0 or q_1 ,
- compute the value of t with these assignments,
- push each tuple of assignment for each clause, in the contexts C_\vee ,
- check the coherence of assignments by means of equality tests between the tuples pushed, in the context C_\wedge .

More formally, we have the following transitions in Δ :

$$\begin{array}{llll} X_i & \rightarrow & q_0(0) & \\ X_i & \rightarrow & q_1(1) & i \leq n \\ id(q_\varepsilon(y_1)) & \rightarrow & q_\varepsilon(y_1) & \\ false(q_\varepsilon(y_1)) & \rightarrow & q_0(y_1) & \\ \neg(q_\varepsilon(y_1)) & \rightarrow & q_{1-\varepsilon}(y_1) & \text{with } \varepsilon \in \{0, 1\} \\ \vee(q_{\varepsilon_1}(y_1), q_{\varepsilon_2}(y_2)) & \rightarrow & q_{\varepsilon_1 \vee \varepsilon_2}(\vee(y_1, y_2)) & \\ \wedge(q_{\varepsilon_1}(y_1), q_{\varepsilon_2}(y_2)) & \xrightarrow{y_1=y_2} & q_{\varepsilon_1 \wedge \varepsilon_2}(y_1) & \text{with } \varepsilon_1, \varepsilon_2 \in \{0, 1\} \end{array}$$

We can verify that the above $\text{VTAM}^\# \mathcal{A}$ recognizes t iff the instance of 3-SAT has a solution. \square

$\text{VTAM}_{\neq}^\#$ is closed under union (using the same construction as before) but not under complementation. This is a consequence of the following Theorem.

Theorem 3.10. *The universality problem is undecidable for $\text{VTAM}_{\neq}^\#$.*

Proof. We reduce the blank accepting problem for a deterministic Turing machine \mathcal{M} . Like in the proof of Theorem 3.6, we encode *configurations* of \mathcal{M} as right-combs on a signature Σ containing the tape and state symbols of \mathcal{M} , considered as binary symbols of Σ_{PUSH} and a constant symbol ε in Σ_{PUSH} . A sequence of configurations c_0, c_1, \dots, c_n (with $n \geq 1$) is encoded as a tree $t = f(c_n(f(c_{n-1}, \dots f(c_0, \varepsilon))))$, where f is a binary symbol of $\Sigma_{\text{INT}_1^-}$. Such a tree is called a *computation* of \mathcal{M} if c_0 is the initial configuration, c_n is a final configuration

$$\begin{array}{rcccl}
& \varepsilon & \rightarrow & q_\varepsilon(\varepsilon) & f(q_B(y_1), q(y_2)) & \xrightarrow{y_1 \neq y_2} & q_f(y_1) \\
f(q_B(y_1), q_\varepsilon(y_2)) & \xrightarrow{y_1 \neq y_2} & q(y_1) & f(q_B(y_1), q_f(y_2)) & \xrightarrow{y_1 = y_2} & q_f(y_1) \\
f(q_B(y_1), q(y_2)) & \xrightarrow{y_1 = y_2} & q(y_1) & f(q_B(y_1), q_f(y_2)) & \xrightarrow{y_1 \neq y_2} & q_f(y_1)
\end{array}$$

Figure 3: The $\text{VTAM}_{\neq} \mathcal{A}_3$ in the proof of Theorem 3.10.

$$\begin{array}{rcccl}
\varepsilon \rightarrow q_\varepsilon(\varepsilon) & f(q_\forall(y_1), q_\varepsilon(y_2)) & \xrightarrow{y_1 \neq y_2} & q_\forall(y_1) & f(q_=(y_1), q_\square(y_2)) & \xrightarrow{y_1 \neq y_2} & q_f(y_1) \\
& f(q_\forall(y_1), q_\forall(y_2)) & \xrightarrow{y_1 = y_2} & q_\forall(y_1) & f(q_\forall(y_1), q_f(y_2)) & \xrightarrow{y_1 = y_2} & q_f(y_1) \\
& f(q_\square(y_1), q_\forall(y_2)) & \xrightarrow{y_1 = y_2} & q_\square(y_1) & & &
\end{array}$$

Figure 4: The $\text{VTAM}_{\neq} \mathcal{A}_4$ in the proof of Theorem 3.10.

and for all $0 \leq i < n$, c_{i+1} is the successor of c_i with \mathcal{M} . Moreover, we assume that all the c_i have the same length (for this purpose we complete the representations of configurations with blank symbols).

We want to construct a $\text{VTAM}_{\neq} \mathcal{A}$ which recognizes exactly the terms which are *not* computations of \mathcal{M} . Hence, \mathcal{A} recognizes all the terms of $\mathcal{T}(\Sigma)$ iff \mathcal{M} does not accept the initial blank configuration.

For the construction of \mathcal{A} , let us first observe that we can associate to \mathcal{M} a VTAM \mathcal{A}_\square which, while reading a configuration c_i , will push on the memory its successor c_{i+1} . The existence of such an automaton is guaranteed by the first fact that for each regular binary relation R , as defined in Section 3.3, there exists a VTAM which, for each $(s, t) \in R$, will push t while reading s , and by the second fact that the language of $c_i \otimes c_{i+1}$, hence the relation of successor configuration, are regular. Moreover, since only push operations are performed, we can ensure that \mathcal{A}_\square satisfies the visibly condition. Let us note q_\square the final state (which is assumed unique wlog) of the VTAM \mathcal{A}_\square . We also use the following VTAMs:

- \mathcal{A}_\forall : a VTAM with (unique) final state q_\forall which, while reading a configuration c_i will push on the memory any configuration with same length as c_i ,
- $\mathcal{A}_=$: a VTAM with final state $q_=$ which, while reading a configuration c_i will push c_i on the memory,
- \mathcal{A}_B : a VTAM with final state q_B which, while reading a configuration c_i will push on the memory a configuration with same length as c_i and containing only blank symbols.

The $\text{VTAM}_{\neq} \mathcal{A}$ is the union of the following automata:

- A_1 : a VTAM_{\neq} recognizing the terms of $\mathcal{T}(\Sigma)$ which are not representations of sequences of configurations (malformed terms). Its language is actually a regular tree language.
- A_2 : a VTAM_{\neq} recognizing the sequences of configurations $f(c_n(f(c_{n-1}, \dots f(c_0, \varepsilon))))$ such that c_0 is not initial or c_n is not final. Again, this is a regular tree language.
- A_3 : a VTAM_{\neq} recognizing the sequences of configurations with two configurations of different lengths. It contains the transitions rules of \mathcal{A}_B and the additional transitions described in Figure 3, which perform this test.
- A_4 : a VTAM_{\neq} recognizing the sequences of configurations $f(c_n(f(c_{n-1}, \dots f(c_0, \varepsilon))))$ such that all the c_i have the same length but there exists $0 \leq i < n$ such that c_{i+1} is not the successor of c_i by \mathcal{M} . This last VTAM_{\neq} contains the transitions of \mathcal{A}_\square , \mathcal{A}_\forall , $\mathcal{A}_=$, and the additional transitions described in Figure 4.

With the transition rules in Figure 4, the automaton \mathcal{A}_4 guesses a $i < n$ and, while reading each of the configurations c_j with $j \leq i$, it pushes the successor configuration of c_j , say c'_j (second column of figure 4). Then, while reading c_{i+1} \mathcal{A}_4 pushes c_{i+1} , and it checks that c'_i and c_{i+1} differ. After that, when reading each of the remaining configurations, \mathcal{A}_4 pushes c_{i+1} (third column of figure 4).

The $\text{VTAM}_{\neq}^{\equiv} \mathcal{A}_1$ to \mathcal{A}_4 cover all the cases of term $\mathcal{T}(\Sigma)$ not being an accepting computation of \mathcal{M} starting with the initial blank configuration. Hence the language of their union \mathcal{A} is $\mathcal{T}(\Sigma)$ iff \mathcal{M} does not accept the initial blank configuration. \square

Corollary 3.11. *$\text{VTAM}_{\neq}^{\equiv}$ is not effectively closed under complementation.*

Proof. It is a consequence of Corollary 3.8 (emptiness decision) and Theorem 3.10. \square

3.4.2. Structural Constraints. Lemma 3.7 applies also to another class $\text{VTAM}_{\neq}^{\equiv}$, where \equiv denotes structural equality of terms, defined recursively as the smallest equivalence relation on ground terms such that:

- $a \equiv b$ for all a, b of arity 0,
- $f(s_1, s_2) \equiv g(t_1, t_2)$ if $s_1 \equiv t_1$ and $s_2 \equiv t_2$, for all f, g of arity 2.

Note that it is a regular relation, and that it satisfies the hypothesis of Lemma 3.7 and the condition *ii* of Theorem 3.2.

Corollary 3.12. *The emptiness problem is decidable for $\text{VTAM}_{\neq}^{\equiv}$.*

Following the procedure in the proof of Theorem 3.2, we obtain a 2-EXPTIME complexity for this problem and this class.

The crucial property of the relations \equiv and \neq is that, unlike the above class $\text{VTAM}_{\neq}^{\equiv}$ or the general VTAM_{\neq}^R , they ignore the labels of the contents of the memory. They just care of the structure of these memory terms. A benefit of this property of $\text{VTAM}_{\neq}^{\equiv}$ is that the decision of the membership problem drops to PTIME for this class.

Theorem 3.13. *The membership problem is decidable in PTIME for $\text{VTAM}_{\neq}^{\equiv}$.*

Proof. Let $\mathcal{A} = (\Gamma, \equiv, Q, Q_f, \Delta)$ be a $\text{VTAM}_{\neq}^{\equiv}$ on Σ and let t be a term in $\mathcal{T}(\Sigma)$. Let $\text{sub}(t)$ be the set of subterms of t and let us construct a $\text{VTAM} \mathcal{A}' = (\Gamma, \text{sub}(t) \times Q, \{t\} \times Q_f, \Delta')$ on Σ' where the symbols of Σ' and Σ are the same, and we assume that the symbols in category INT_1^{\equiv} (resp. INT_2^{\equiv}) in the partition of Σ are in INT_1 (resp. INT_2) in the partition of Σ' . The transitions of Δ' are obtained by the following transformation of the transitions of Δ . We only describe the construction for the cases INT_1 and INT_1^{\equiv} with positive constraints. The other cases are similar.

- for every $f_7(q_1(y_1), q_2(y_2)) \rightarrow q(y_1) \in \Delta$, we add to Δ' all the transitions: $f_7(\langle q_1, t_1 \rangle(y_1), \langle q_2, t_2 \rangle(y_2)) \rightarrow \langle q, f(t_1, t_2) \rangle(y_1)$ such that $f(t_1, t_2) \in \text{sub}(t)$,
- for every $f_9(q_1(y_1), q_2(y_2)) \xrightarrow{y_1 \equiv y_2} q(y_1) \in \Delta$, we add to Δ' all the transitions as above (in this case, f_9 is assumed a symbol of category INT_1 in Σ') such that moreover $\text{struct}(t_1) = \text{struct}(t_2)$, where $\text{struct}(s)$ is defined, like in the proof of Theorem 3.5, as the shape (unlabeled tree) that will have the memory of \mathcal{A} after \mathcal{A} processed s .

The $\text{VTAM} \mathcal{A}'$ can be computed in time $O(\|t\|^2 \times \|\mathcal{A}\|)$. It recognizes at most one term, t , and it recognizes t iff \mathcal{A} recognizes t . Therefore, t is recognized by \mathcal{A} iff the language of \mathcal{A}' is not empty. This can be decided in PTIME according to Theorem 2.5. \square

Even more interesting, the construction for determinization of Section 2.3 still works for $\text{VTAM}_{\neq}^{\equiv}$.

Theorem 3.14. *For every $\text{VTAM}_{\neq}^{\equiv} \mathcal{A} = (\Gamma, \equiv, Q, Q_f, \Delta)$ there exists a deterministic $\text{VTAM}_{\neq}^{\equiv} \mathcal{A}^{det} = (\Gamma^{det}, \equiv, Q^{det}, Q_f^{det}, \Delta^{det})$ such that $L(\mathcal{A}) = L(\mathcal{A}^{det})$, where $|Q^{det}|$ and $|\Gamma^{det}|$ both are $O(2^{|\mathcal{Q}|^2})$.*

Proof. We use the same construction as in the proof of Theorem 2.3, with a direct extension of the construction for INT to INT^{\equiv} . The key property for handling constraints is that the structure of memory (hence the result of the structural tests) is independent from the non-deterministic choices of the automaton. With the visibility condition it only depends on the term read. \square

Theorem 3.15. *The class of tree languages of $\text{VTAM}_{\neq}^{\equiv}$ is closed under Boolean operations. One can construct $\text{VTAM}_{\neq}^{\equiv}$ for union, intersection and complement of given $\text{VTAM}_{\neq}^{\equiv}$ languages whose sizes are respectively linear, quadratic and exponential in the size of the initial $\text{VTAM}_{\neq}^{\equiv}$.*

Proof. We use the same constructions as in Theorem 2.4 (VTAM) for union and intersection. For the intersection, in the case of constrained rules we can safely keep the constraints in product rules, thanks to the visibility condition (as the structure of memory only depends on the term read, see the proof of Theorem 3.14). For instance, the product of the INT_1^{\equiv} rules $f_9(q_{11}(y_1), q_{12}(y_2)) \xrightarrow{y_1 \equiv y_2} q_1(y_1)$ and $f_9(q_{21}(y_1), q_{22}(y_2)) \xrightarrow{y_1 \equiv y_2} q_1(y_1)$ is $f_9(\langle q_{11}, q_{21} \rangle(y_1), \langle q_{12}, q_{22} \rangle(y_2)) \xrightarrow{y_1 \equiv y_2} \langle q_1, q_2 \rangle(y_1)$. The product of two INT_1^{\neq} is constructed similarly. We do not need to consider the product of a rule INT_1^{\equiv} with a rule INT_1^{\neq} , and vice-versa, because in this case the product is empty (no rule is added to the $\text{VTAM}_{\neq}^{\equiv}$ for intersection). For the complementation, we use Theorem 3.14 and completion. \square

Corollary 3.16. *The universality and inclusion problems are decidable for $\text{VTAM}_{\neq}^{\equiv}$.*

Proof. This is a consequence of Corollary 3.12 and Theorem 3.15. \square

3.5. Constrained PUSH Transitions. Above, we always considered constraints in transitions with INT symbols only. We did not consider a constrained extension of the rules PUSH . The main reason is that symbols of a new category PUSH^{\equiv} , which test two memories for structural equality and then push a symbol on the top of them, permit us to construct a constrained $\text{VTAM} \mathcal{A}$ whose memory language $M(\mathcal{A}, q)$ is the set of well-balanced binary trees. This language is not regular, whereas the base of our emptiness decision procedure is the result (Theorem 3.2, Lemma 3.7) of regularity of these languages for the classes considered.

3.6. Contexts as Symbols and Signature Translations. Before looking for some examples of $\text{VTAM}_{\neq}^{\equiv}$ languages, we show a "trick" that (seemingly) adds expressiveness to $\text{VTAM}_{\neq}^{\equiv}$. One symbol can perform either a PUSH or a POP operation, or make an INT transition (constrained or not), but it cannot combine several of these operations. Here, we propose a way to combine several operations in one symbol, and thus increase the expressiveness of $\text{VTAM}_{\neq}^{\equiv}$, without losing the good properties of this class.

The trick is to replace symbols by *contexts*. For instance a context $g_2(g_1(\cdot, \cdot), g_0)$ can replace a symbol of arity 2. Assume that g_2 is a PUSH symbol, g_1 is an INT₁ symbol with test, and g_0 is an INT₀ symbol. This context first performs a test on the memories of the sons, and then a PUSH operation on the memory kept by g_1 (and on the \perp leaf created by g_0). Such a combination is normally not possible, and replacing symbols by contexts brings a lot of additional expressiveness.

Here is how we precisely proceed: we want to recognize a language (on a signature Σ) with a VTAM, and we have then to choose the categories for each symbol of the signature (PUSH, POP_{ij}, INT₁[≡], ...). As we will see in the examples below, it might be useful in practice to have some extra categories combining the powers of two or more categories of VTAM_≡[≠]. We can do that still with VTAM_≡[≠], by mean of an encoding of the terms of $\mathcal{T}(\Sigma)$. More precisely, we replace some symbols of the initial signature Σ by contexts built with new symbols. For instance, we replace a $g \in \Sigma$, which will perform the complex operation described above, by the context $g_2(g_1(\cdot, \cdot), g_0)$. Then, we will have to ensure that the new symbols (in our example g_0 , g_1 and g_2) are only used to form the contexts encoding the symbols of Σ . This can easily be done with local information maintained in the state of the automaton. The set of well formed terms, built with new symbols organized in allowed contexts, is a regular tree language. We will call the VTAM_≡[≠] signature obtained a *translation* of the initial signature. If L is a tree language on Σ , then $c(L)$ is the translation of L .

In summary, we have shown here a general method for adding new categories of symbols corresponding to (relevant) combinations of operations of VTAM_≡[≠], and hence defining extensions of VTAM_≡[≠] with the same good properties as VTAM_≡[≠]. By *relevant*, we mean that some combinations are excluded, like for instance, PUSH + constraint \equiv at the same time (see paragraph above). Such forbidden combination cannot be handled by our method. With similar encodings, we can deal with symbols of arity bigger than 2, *e.g.* $g(\cdot, \cdot, \cdot)$ can be replaced by $g_2(\cdot, g_1(\cdot, \cdot))$.

Note however first that this encoding concerns the recognized tree, *not the memories*. For instance, it is not possible to systematically encode the syntactic equality as structural equality (on memories) in this way. And indeed, the decision results are drastically different in the two cases.

Also note that, even if $c(L)$ is accepted by a VTAM, which implies that $\neg c(L)$ is also accepted by a VTAM, it may well be the case that $c(\neg L)$ is not recognized by a VTAM. So, the above trick does not show that we can extend our results to a wider class of tree languages.

3.7. Some VTAM_≡[≠] Languages. The regular tree languages and VPL are particular cases of VTAM languages. We present in this section some other examples of relevant tree languages translatable, using the method of Section 3.6, into VTAM_≡[≠] languages.

Well balanced binary trees. The VTAM_≡[≠] with memory signature $\{f, \perp\}$, state set $\{q, q_f\}$, unique final state q_f , and whose rules follow accepts the (non-regular) language of well balanced binary trees build with g and a .

Here a is a constant in Σ_{INT_0} , and g is in a new category, and is translated into the context $g_2(g_1(\cdot, \cdot), g_0)$, where $g_2 \in \Sigma_{\text{PUSH}}$, $g_1 \in \Sigma_{\text{INT}_1}^{\equiv}$, and $g_0 \in \Sigma_{\text{INT}_0}$.

$$\begin{array}{lcl}
a & \rightarrow & q_f(\perp) \\
g_0 & \rightarrow & q_0(\perp)
\end{array}
\quad
\begin{array}{lcl}
g_1(q_f(y_1), q_f(y_2)) & \xrightarrow{y_1 \equiv y_2} & q(y_1) \\
g_2(q(y_1), q_0(y_2)) & \longrightarrow & q_f(f(y_1, y_2))
\end{array}$$

Powerlists. A powerlist [18] is roughly a list of length 2^n (for $n \geq 0$) whose elements are stored in the leaves of a balanced binary tree. For instance, the elements may be integers represented in unary notation with the unary successor symbol s and the constant 0, and the balanced binary tree on the top of them can be built with a binary symbol g . This data structure has been used in [18] to specify data-parallel algorithms based on divide-and-conquer strategy and recursion (*e.g.* Batcher's merge sort and fast Fourier transform).

It is easy following the above construction to characterize translations of powerlists with a $\text{VTAM}_{\neq}^{\equiv}$. We do not push on the "leaves", *i.e.* on the elements of the powerlist, and compute in the higher part (the complete binary tree) as above.

Some equational properties of algebraic specifications of powerlists have been studied in the context of automatic induction theorem proving and sufficient completeness [17]. Tree automata with constraints have been acknowledged as a very powerful formalism in this context (see *e.g.* [9]). We therefore believe that a characterization of powerlists (and their complement language) with $\text{VTAM}_{\neq}^{\equiv}$ is useful for the automated verification of algorithms on this data structure.

Red-black trees. A red-black tree is a binary search tree following these properties:

- (1) every node is either red or black,
- (2) the root node is black,
- (3) all the leaves are black,
- (4) if a node is red, then both its sons are black,
- (5) every path from the root to a leaf contains the same number of black nodes.

The four first properties are local and can be checked with standard TA rules. The fifth property make the language red-black trees not regular and we need $\text{VTAM}_{\neq}^{\equiv}$ rules to recognize it. It can be checked by pushing all the black nodes read. We use for this purpose a symbol $black \in \Sigma_{\text{PUSH}}$.

When a red node is read, the number of black nodes in both its sons are checked to be equal (by a test \equiv on the corresponding memories) and only one corresponding memory is kept. This is done with a symbol $red \in \Sigma_{\text{INT}_1}^{\equiv}$.

When a black node is read, the equality of number of black nodes in its sons must also be tested, and a $black$ must moreover be pushed on the top of the memory kept. It means that two operations must be combined. We can do that by defining an appropriate context with the method of Section 3.6.

In [15] a special class of tree automata is introduced and used in a procedure for the verification of C programs which handle balanced tree data structures, like red-black tree. Based on the above example, we think that, following the same approach, $\text{VTAM}_{\neq}^{\equiv}$ can also be used for similar purposes.

BTINT ₁	$f_{13}(q_1(y_1), q_2(y_2))$	$\xrightarrow{1=2}$	$q(y_1)$	$f_{13} \in \Sigma_{\text{BTINT}_1}$
BTINT ₂	$f_{14}(q_1(y_1), q_2(y_2))$	$\xrightarrow{1=2}$	$q(y_2)$	$f_{14} \in \Sigma_{\text{BTINT}_2}$
BTINT ₁	$f_{15}(q_1(y_1), q_2(y_2))$	$\xrightarrow{1 \neq 2}$	$q(y_1)$	$f_{15} \in \Sigma_{\text{BTINT}_1}$
BTINT ₂	$f_{16}(q_1(y_1), q_2(y_2))$	$\xrightarrow{1 \neq 2}$	$q(y_2)$	$f_{16} \in \Sigma_{\text{BTINT}_2}$

Figure 5: New transition categories for BTVTAM_{-R}^R .

4. VISIBLY TREE AUTOMATA WITH MEMORY AND STRUCTURAL CONSTRAINTS AND BOGAERT-TISON CONSTRAINTS

In Section 3, we have only considered VTAM with constraints testing the *memories* contents. In this section, we go a bit further and add to VTAM_{-R}^R some Bogaert-Tison constraints [4], i.e. equality and disequality tests between *brother* subterms in the term read by the automaton.

We consider two new categories for the symbols which we call BTINT_1 and BTINT_2 , for "Bogaert-Tison Internal". A transition with a symbol in one of these categories will make no test on the memory contents, but rather an equality or disequality test between the brother subterms directly under the current position of computation. In Figure 5, we describe the new transitions categories. We use the same notation as in [4] for the constraints. Note that again, we only allow Bogaert-Tison constraints in internal rules.

For instance, if $f_{13}(t_1, t_2)$ is a subterm of the input tree, and if t_1 leads to $q_1(m_1)$, and t_2 to $q_2(m_2)$, then the transition rule $f_{13}(q_1(y_1), q_2(y_2)) \xrightarrow{1=2} q(y_1)$, of type BTINT_1 can be applied at this position *iff* $t_1 = t_2$.

Definition 4.1. A *visibly tree automaton with memory and constraints and Bogaert-Tison tests* (BTVTAM_{-R}^R) on a signature Σ is a tuple $(\Gamma, R, Q, Q_f, \Delta)$ where Γ, Q, Q_f are defined as for TAM, R is an equivalence relation on $\mathcal{T}(\Gamma)$ and Δ is a set of rewrite rules in one of the above categories: $\text{PUSH}, \text{POP}_{11}, \text{POP}_{12}, \text{POP}_{21}, \text{POP}_{22}, \text{INT}_0, \text{INT}_1, \text{INT}_2, \text{INT}_1^R, \text{INT}_2^R, \text{BTINT}_1, \text{BTINT}_2$.

The acceptance of terms of $\mathcal{T}(\Sigma)$ and languages of term and memories are defined and denoted as in Section 2.1.

The definition of *complete* BTVTAM_{-R}^R is the same as before. Every BTVTAM_{-R}^R can be completed (with a polynomial overhead) by the addition of a trash state q_\perp (the construction is similar to the one for VTAM_{-R}^R in Section 3.1).

The definition of *deterministic* BTVTAM_{-R}^R is based on the same conditions as for VTAM_{-R}^R for the function symbols in categories $\text{PUSH}_0, \text{PUSH}, \text{POP}_{11}, \dots, \text{POP}_{22}, \text{INT}_1, \text{INT}_2, \text{INT}_1^R, \text{INT}_2^R$, and for the function symbols of $\text{BTINT}_1, \text{BTINT}_2$, we use the same kind of conditions as for $\text{INT}_1^R, \text{INT}_2^R$: for all $f \in \Sigma_{\text{BTINT}_1} \cup \Sigma_{\text{BTINT}_2}$ for all $q_1, q_2 \in Q$, there are at most two rules in Δ with left-member $f(q_1(y_1), q_2(y_2))$, and if there are two, then their constraints have different signs.

Theorem 4.2. For every $\text{BTVTAM}_{\neq}^{\equiv} \mathcal{A} = (\Gamma, \equiv, Q, Q_f, \Delta)$ there exists a deterministic $\text{BTVTAM}_{\neq}^{\equiv} \mathcal{A}^{det} = (\Gamma^{det}, \equiv, Q^{det}, Q_f^{det}, \Delta^{det})$ such that $L(\mathcal{A}) = L(\mathcal{A}^{det})$, where $|Q^{det}|$ and $|\Gamma^{det}|$ both are $O(2^{|Q|^2})$.

Proof. We use, again, the same construction as in the proof of Theorem 2.3, with a direct extension of the construction for INT to INT^{\equiv} and BTINT. As mentioned in Theorem 3.14, the extension works for INT^{\equiv} because the results of the tests are independent from the non-deterministic choices of the automaton. For BTINT it is exactly the same (the brother terms are not changed by the automaton!). \square

Theorem 4.3. *The class of tree languages of $\text{BTVTAM}_{\neq}^{\equiv}$ is closed under Boolean operations.*

Proof. We use the same constructions as in Theorem 2.4 for union and intersection. For the intersection, as in Theorem 3.15, the constraints (even Bogaert-Tison tests) can be safely kept in product rules, thanks to the visibility condition. For the complementation, we use Theorem 4.2 and complementation. \square

The proof of the following theorem follows the same idea as the proof for Bogaert-Tison automata [4], but we need here to take care of the structural constraints on the memory contents. A consequence is that the complexity of emptiness decision is much higher.

Theorem 4.4. *The emptiness problem is decidable for $\text{BTVTAM}_{\neq}^{\equiv}$.*

Proof. Let \mathcal{A} be a $\text{BTVTAM}_{\neq}^{\equiv}$. First we determinize it into \mathcal{A}^{det} and assume that \mathcal{A}^{det} is also complete. Then, we delete the rules BTINT_1 of the form: $f(q_1(y_1), q_2(y_2)) \xrightarrow{1=2} q(y_1)$, with q_1 distinct from q_2 (idem for BTINT_2 rules) because they can't be used (the automaton is deterministic so one term cannot lead to two different states).

For the same reason, we change each rule BTINT_1 of the form: $f(q_1(y_1), q_2(y_2)) \xrightarrow{1 \neq 2} q(y_1)$ with q_1 distinct from q_2 (idem for BTINT_2^{\neq} rules) into the same rule but without the disequality test: $f(q_1(y_1), q_2(y_2)) \rightarrow q(y_1)$.

We call the newly obtained automaton \mathcal{A}^{new} . It is still deterministic and recognizes the same language as \mathcal{A}^{det} . Actually, the careful reader may notice that \mathcal{A}^{new} is not a true $\text{BTVTAM}_{\neq}^{\equiv}$, because some unconstrained rules may involve symbols in BTINT in this automaton. However, it is just an intermediate step in the construction of another automaton \mathcal{A}' below.

Now, we consider the remaining BTINT_1 or BTINT_2 rules with negative Bogaert-Tison constraints, which are of the form: $f(q_1(y_1), q_1(y_2)) \xrightarrow{1 \neq 2} q(y_1)$ (or $q(y_2)$). We denote them by $R_1, \dots, R_i, \dots, R_N$, and denote by q_i the state in the left member of R_i , for each $i \leq N$. We also denote the corresponding BTINT_1 or BTINT_2 rules by $S_1, \dots, S_i, \dots, S_N$. Note that, since \mathcal{A}^{det} is deterministic and complete, we can associate to each rule of BTINT_i , whose constraint is negative, a unique rule of BTINT_i with a positive constraint and the same states in its left member. So, the state in the left member of S_i is the same q_i as for R_i .

It is important to notice that if a rule R_i can effectively be used, then there must exist two distinct terms leading to the state q_i (we will call them witnesses). If not, the rule can be removed.

So, our purpose is now to find, for each rule R_i , whether two witnesses exist or not. We let \mathcal{R} be initially $\{R_1, \dots, R_N\}$. Suppose that at least one R_i rule can be used, and consider a run on a term t that uses such a rule. We consider an innermost application of a rule R_i in this run on a subterm $f(t_1, t_2)$. The run on t_1 and the run on t_2 both lead to the state q_i , without any use of an R_j rule.

Let us remove all the R_i rules from \mathcal{A}^{new} , and we remove all the equality tests in the S_i rules. Let \mathcal{A}' be the resulting automaton. It is a deterministic $\text{VTAM}_{\neq}^{\equiv}$ (considering the

symbols in BTINT as INT symbols in this new automaton), and each term in $L(\mathcal{A}', q_i)$ can be transformed (we will call it *BT-transformation*) into a term in $L(\mathcal{A}^{new}, q_i)$: each time we use a modified S_i rule, for instance of type BTINT₁, on a subtree $f(t_1, t_2)$, we replace t_2 with t_1 so that the equality test is satisfied (and the resulting memory is unchanged). Important: all the replacements must be performed bottom-up.

The proof of the emptiness decidability of VTAM $\stackrel{\equiv}{\neq}$ (Corollary 3.12) is constructive, hence if we choose a reachable state q_j , we can find a term in $L(\mathcal{A}', q_j)$ to this state, and then convert it into a witness. So, we can find a first witness $t_A \in L(\mathcal{A}^{new}, q_j)$.

If no witness can be found, then all the R_i rules are useless and we can definitely remove them all. Otherwise, we still need to find another witness, and if there is at least one such other witness, then one of them can be recognized without using a R_i rule. We can construct a VTAM $\stackrel{\equiv}{\neq}$ recognizing all the terms whose BT-transformation leads to t_A . To design it, we read t_A top-down (knowing the state of \mathcal{A}' at each node), and each time we see a subterm $f(t_1, t_2)$ to which a modified S_i rule has to be applied, for instance a modified BTINT₁ (resp. BTINT₂) rule, the right (resp. left) son of f only needs to be a term in $L(\mathcal{A}', q_i)$, and the left (resp. right) son of f only needs to be BT-transformed into t_1 (resp. t_2). Once this VTAM $\stackrel{\equiv}{\neq}$ is constructed, we can combine it with \mathcal{A}' in order to obtain a VTAM $\stackrel{\equiv}{\neq}$ recognizing all the terms leading \mathcal{A}' to q_j (the state reached by \mathcal{A}' on t_A) except the terms whose BT-transformation is t_A . Then we find another term in $L(\mathcal{A}', q_j)$ (if it exists) and its BT-transformation is not t_A : it is actually another witness t_B .

When we have two witnesses for a rule R_j , we remove it from \mathcal{R} , and we add this rule R_j to \mathcal{A}' , but without the disequality test. The automaton \mathcal{A}' keeps its good property: a term t leading \mathcal{A}' to some state q can be BT-transformed into a term leading \mathcal{A}^{new} to state q : when we "meet" the use of a rule formerly in the set \mathcal{R} on $f(t_1, t_1)$ during the bottom-up exploration of t , we replace the right (for a rule that was of type BTINT₁ and with negative constraints) or the left son (otherwise) by a witness different from t_1 , so that the disequality test is satisfied. Note that even if t_1 is a witness, we can do so because we have found two witnesses.

With the new rule in \mathcal{A}' we look for 2 witnesses for some remaining R_i rule. Again, we can show that if a couple of witnesses exists, then at least one couple can be found without any use of the remaining R_i rules. When we find a first witness t_A for a remaining rule R_j , we can find another one (if it exists) using approximately the same technique as previously: we read t_A top-down, and when we see a rule formerly in \mathcal{R} , used on $f(t_1, t_2)$ (e.g. a rule formerly of type BTINT₁ with a negative constraint), we just go on recursively, saying that the left son must be a term whose BT-transformation is t_1 , and the right son must be either:

- a term whose BT-transformation is t_2 ,
- or, if our BT-transformation would change $f(t_1, t_1)$ into $f(t_1, t_2)$, a term whose BT-transformation is t_1 .

As previously, we construct a VTAM $\stackrel{\equiv}{\neq}$, fully using the Boolean closure of this class, that recognizes the terms in $L(\mathcal{A}', q_j)$ (the state reached by \mathcal{A}' on t_A), except those whose BT-transformation is t_A , and therefore we can find another witness (if it exists) t_B .

We continue to use this method, finding couples of witnesses, until there is no rule in the set \mathcal{R} anymore, or until we are not able to find a new couple of witnesses anymore: in that latter case, we remove the remaining R_i rules because they are useless.

So, now we use the final version of \mathcal{A}' obtained in order to find a term leading to a final state, and since we have a couple of witnesses for each rule formerly in the set \mathcal{R} , we can

BT-transform it into a term accepted by \mathcal{A}^{new} (hence by \mathcal{A}). If such a term does not exist, the language recognized by \mathcal{A}^{new} (i.e. the language recognized by \mathcal{A}) is empty. \square

5. CONCLUSION

Having a tree memory structure instead of a stack is sometimes more relevant (even when the input functions symbols are only of arities 1 and 0). We have shown how to extend the visibly pushdown languages to such memory structures, keeping determinization and closure properties of VPL. Our second contribution is then to extend this automaton model, constraining the transition rules with some regular conditions on memory contents. The structural equality and disequality tests appear to be a good class of constraints since we have then both decidability of emptiness and Boolean closure properties. Moreover, they can be combined (while keeping decidability and closure results) with equality and disequality tests a la [4], operating on brothers subterms of the term read.

Several further studies can be done on the automata of this paper. For instance, the problem of the closure of the corresponding tree languages under certain classes of term rewriting systems is particularly interesting, as it can be applied to the verification of infinite state systems with *regular model checking* techniques. It could be interesting as well to study how the definition of VTAM can be extended to deal with unranked trees, with the perspective of applications to problems related to semi-structured documents processing.

Acknowledgments. The authors wish to thank Pierre Réty for having noted some mistakes in the examples in the extended abstract, and for having sent us a basis of comparison of VTAM with (top down) Visibly Pushdown Tree Automata, and Jean Goubault-Larrecq for his suggestion to refer to \mathcal{H}_3 [19] in the proof of Theorem 2.5, and the reviewers for their useful and numerous remarks and suggestions.

REFERENCES

- [1] R. Alur, S. Chaudhuri, and P. Madhusudan. Visibly pushdown tree languages. Available on: <http://www.cis.upenn.edu/~swarat/pubs/vpt1.ps>, 2006.
- [2] R. Alur and P. Madhusudan. Visibly pushdown languages. In L. Babai, editor, *Proceedings of the 36th Annual ACM Symposium on Theory of Computing (STOC 2004)*, pages 202–211. ACM, 2004.
- [3] L. Bachmair and H. Ganzinger. Resolution theorem proving. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, chapter 2. North Holland, 2001.
- [4] B. Bogaert and S. Tison. Equality and Disequality Constraints on Direct Subterms in Tree Automata. In *9th Symp. on Theoretical Aspects of Computer Science, STACS*, volume 577 of *LNCS*, pages 161–171. Springer, 1992.
- [5] J. Chabin and P. Réty. Visibly pushdown languages and term rewriting. In *Proc. 6th International Symposium Frontiers of Combining Systems (FroCoS)*, volume 4720 of *Lecture Notes in Computer Science*, pages 252–266. Springer, 2007.
- [6] W. Charatonik and A. Podelski. Set constraints with intersection. In *Proc. IEEE Symposium on Logic in Computer Science*, Warsaw, 1997.
- [7] H. Comon and V. Cortier. Tree automata with one memory, set constraints and cryptographic protocols. *Theoretical Computer Science*, 331(1):143–214, Feb. 2005.
- [8] H. Comon, M. Dauchet, R. Gilleron, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi. *Tree Automata Techniques and Applications*. <http://www.grappa.univ-lille3.fr/tata>, 1997.
- [9] H. Comon and F. Jacquemard. Ground reducibility is EXPTIME-complete. *Information and Computation*, 187(1):123–153, 2003.

- [10] J.-L. Coquidé, M. Dauchet, R. Gilleron, and S. Vágvolgyi. Bottom-up tree pushdown automata: classification and connection with rewrite systems. *Theoretical Computer Science*, 127(1):69–98, 1994.
- [11] N. Dershowitz and J.-P. Jouannaud. *Rewrite systems*, chapter Handbook of Theoretical Computer Science, Volume B, pages 243–320. Elsevier, 1990.
- [12] T. Frühwirth, E. Shapiro, M. Vardi, and E. Yardeni. Logic programs as types for logic programs. In *Proc. of the 6th IEEE Symposium on Logic in Computer Science*, pages 300–309, 1991.
- [13] J. Goubault-Larrecq. Résolution ordonnée avec sélection et classes décidables en logique du premier ordre. Lecture Notes, 2006. available at <http://www.lsv.ens-cachan.fr/~goubault/S0resol.pdf>.
- [14] I. Guessarian. Pushdown tree automata. *Theory of Computing Systems*, 16(1):237–263, 1983.
- [15] P. Habermehl, R. Iosif, and T. Vojnar. Automata-based verification of programs with tree updates. In *Proc. 12th Intern. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'06)*, volume 3920 of *LNCS*, April 2006.
- [16] T. Jensen, D. L. Métayer, and T. Thorn. Verification of control flow based security policies. In *Proceedings of the IEEE Symposium on Research in Security and Privacy*, pages 89–103. IEEE Computer Society Press, 1999.
- [17] D. Kapur. *Essays in Honor of Larry Wos*, chapter Constructors can be Partial Too. MIT Press, 1997.
- [18] J. Misra. Powerlist: A structure for parallel recursion. *ACM Transactions on Programming Languages and Systems*, 16(6):1737–1767, November 1994.
- [19] F. Nielson, H. R. Nielson, and H. Seidl. Normalizable horn clauses, strongly recognizable relations and spi. In *Proc. 9th Static Analysis Symposium (SAS)*, volume 2477 of *LNCS*, pages 20–35, 2002.
- [20] R. Nieuwenhuis and A. Rubio. Paramodulation-based theorem proving. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, chapter 7. North Holland, 2001.
- [21] K. M. Schimpf and J. Gallier. Tree pushdown automata. *Journal of Computer and System Sciences*, 30(1):25–40, 1985.

APPENDIX: TWO-WAY TREE AUTOMATA WITH STRUCTURAL EQUALITY CONSTRAINTS
ARE AS EXPRESSIVE AS STANDARD TREE AUTOMATA.

In this section, we complete the proof of Lemma 3.7. We show actually a more general result: we consider two-way alternating tree automata with some regular constraints and show that the language they recognize is also accepted by a standard tree automaton. This generalizes the proof for two-way alternating tree automata (see e.g. [8] chapter 7) and the proof for two-way automata with equality tests [7], which itself relies on a transformation from two-way automata to one-way automata [6].

Two-way automata are, as usual, automata that can move up and down and alternation consists (as usual) in spawning to copies of the tree in different states, requiring acceptance of both copies. In the logical formalism, alternation simply corresponds to clauses $q_1(x), q_2(x) \rightarrow q(x)$, requiring to accept x both in state q_1 and in state q_2 if one wants to accept x in state q .

For simplicity, we assume that all function symbols have arity 0 or 2. Lexical conventions:

- f, g, h, \dots are ranging over symbols of arity 2. Unless explicitly stated they may denote identical symbols.
- a, b, c, \dots range over constants
- $x, x_1, \dots, x_i, \dots, y, \dots, y_i, z, \dots, z_i, \dots$ are (universally quantified) first-order variables,
- $S, S_1, S_2, \dots, S_i, \dots$ range over states symbols for a fixed given tree automaton
- Q, Q_1, Q_2, \dots , range over states symbols of the tree automaton with memory
- R, R_1, R_2, \dots , range over state symbols of the binary recognizable relations.

We assume that R_i are recognizable relations defined by clauses of the form:

$$\begin{array}{lll}
 (A) & & \Rightarrow \underline{R(a, b)} \\
 (B) & S_1(x), S_2(y) & \Rightarrow \underline{R(f(x, y), a)} \\
 (C) & S_1(x), S_2(y) & \Rightarrow \underline{R(a, f(x, y))} \\
 (D) & R_1(x_1, x_2), R_2(y_1, y_2) & \Rightarrow \underline{R_3(f(x_1, y_1), g(x_2, y_2))} \\
 (E) & S_1(x), S_2(y) & \Rightarrow \underline{S(f(x, y))} \\
 (F) & & \Rightarrow \underline{S(a)}
 \end{array}$$

We assume wlog that there is a state S_\top in which all trees are accepted (a “trash state”).

Moreover, we will need in what follows an additional property of the R_i 's:

$$\forall i, j, \exists k, l, R_i(x, y) \wedge R_j(y, z) \models R_k(x, y) \wedge R_l(x, z)$$

This property is satisfied by the structural equivalence, for which there is only one index i : $R_i \equiv \equiv$ and we have indeed

$$x \equiv y \wedge y \equiv z \models x \equiv y \wedge x \equiv z$$

It is also satisfied by the universal binary relation and by the equality relation. That is why this generalizes corresponding results of [8, 7].

Our automata are defined by a finite set of clauses of the form:

$$\begin{aligned}
(1) \quad & \underline{Q_1(y_1)}, \underline{Q_2(y_2)}, R(y_1, y_2) \Rightarrow Q_3(y_1) \\
(2) \quad & \underline{Q_1(y_1)}, \underline{Q_2(y_2)} \Rightarrow \underline{Q_3(f(y_1, y_2))} \\
(2b) \quad & \Rightarrow \underline{Q_1(a)} \\
(3) \quad & \underline{Q_1(f(y_1, y_2))}, Q_2(y_3) \Rightarrow \underline{Q_3(y_1)} \\
(4) \quad & \underline{Q_1(f(y_1, y_2))}, Q_2(y_3) \Rightarrow Q_3(y_2)
\end{aligned}$$

These clauses have a least Herbrand model. We write $\llbracket Q \rrbracket$ the interpretation of Q in this model. This is the language recognized by the automaton in state Q .

The goal is to prove that, for every Q , $\llbracket Q \rrbracket$ is recognized by a finite tree automaton. We use a selection strategy, with splitting and complete the rules (1)-(4) above. We show that the completion terminates and that we get out of it a tree automaton which accepts exactly the memory contents. Splitting will introduce nullary predicate symbols (propositional variables).

We consider the following selection strategy. Let E_1 be the set of literals which contain at least one function symbol and E_2 be the set of negative literals

- (1) If the clause contains a negative literal $\neg R(u, v)$ or a negative literal $\neg S(u)$ where either u, v is not a variable, then select such literals only. This case is ruled out in what follows
- (2) If the clause contains at least one negated propositional variable, select the negated propositional variables only. This case is ruled out in what follows
- (3) If $E_1 \cap E_2 \neq \emptyset$, then select $E_1 \cap E_2$
- (4) If $E_1 \neq \emptyset$ and $E_1 \cap E_2 = \emptyset$, then select E_1
- (5) If $E_1 = \emptyset$ and $E_2 \neq \emptyset$, then select the negative literals $\neg R(x, y)$ and $\neg S(x)$ if any, otherwise select E_2
- (6) Otherwise, select the only literal of the clause

In what follows (and precedes), selected literals are underlined.

We introduce the procedure by starting to run the completion with the selection strategy, before showing the general form of the clauses we get.

First, clauses of the form (3), (4) are replaced (using splitting) with clauses of the form

$$\begin{aligned}
(3) \quad & \underline{Q_1(f(y_1, y_2))}, \underline{\mathbf{NE}_{Q_2}} \Rightarrow Q_3(y_1) \\
(4) \quad & \underline{Q_1(f(y_1, y_2))}, \underline{\mathbf{NE}_{Q_2}} \Rightarrow Q_3(y_2) \\
(s_1) \quad & \underline{Q_2(x)} \Rightarrow \mathbf{NE}_{Q_2}
\end{aligned}$$

Overlapping (s_1) and (2, 2b) may yield clauses of the form

$$\begin{aligned}
(s_2) \quad & \underline{\mathbf{NE}_{Q_1}}, \underline{\mathbf{NE}_{Q_2}} \Rightarrow \mathbf{NE}_{Q_3} \\
(s_3) \quad & \underline{\mathbf{NE}_{Q_1}}, \underline{\mathbf{NE}_{Q_2}} \Rightarrow \underline{\mathbf{NE}_Q}
\end{aligned}$$

together with new clauses of the form (s_1) . Eventually, we may reach, using (s_3) and (3-4) clauses:

$$\begin{aligned}
(3b) \quad & \underline{Q_1(f(y_1, y_2))} \Rightarrow Q_3(y_1) \\
(4b) \quad & \underline{Q_1(f(y_1, y_2))} \Rightarrow Q_3(y_2)
\end{aligned}$$

(1) + (2) yields clauses of the form

$$(5.1) \quad Q_1(y_1), Q_2(y_2), \underline{Q_3(g(y_3, y_4))}, R_1(y_1, y_3), R_2(y_2, y_4) \Rightarrow Q_4(f(y_1, y_2))$$

$$(5.2) \quad \underline{Q_1(y_1), Q_2(y_2), Q_3(a)}, S_1(y_1), S_2(y_2) \Rightarrow Q_4(f(y_1, y_2))$$

$$(5.3) \quad \underline{Q_1(a)} \Rightarrow Q_2(b)$$

$$(5.4) \quad S_1(y_1), S_2(y_2), \underline{Q_1(f(y_1, y_2))} \Rightarrow Q_2(a)$$

(2) +(3b) and (2) + (4b) yield clauses of the form (after splitting):

$$(6) \quad \underline{\mathbf{NE}_{Q_3}}, Q_1(y_1) \Rightarrow Q_2(y_1)$$

and eventually

$$(6b) \quad \underline{Q_1(y_1)} \Rightarrow Q_2(y_1)$$

(5.1) + (2) yields

$$(7.1) \quad Q_1(y_1), Q_2(y_2), Q_3(y_3), Q_4(y_4), R_1(y_1, y_3), R_2(y_2, y_4) \Rightarrow \underline{Q_5(f(y_1, y_2))}$$

We split (7.1) : we introduce new predicate symbols $Q_i^{R_j}$ defined by

$$Q_i(y), R_j(x, y) \Rightarrow Q_i^{R_j}(x)$$

Then clauses (7.1) becomes:

$$(7.1) \quad Q_1(y_1), Q_2(y_2), Q_3^{R_1}(y_1), Q_4^{R_2}(y_2) \Rightarrow \underline{Q_5(f(y_1, y_2))}$$

(5.2) + (2b) yields clauses of the form

$$(7.2) \quad Q_1(y_1), Q_2(y_2), S_1(y_1), S_2(y_2) \Rightarrow \underline{Q_3(f(y_1, y_2))}$$

(6b) + (2) yields new clauses of the form (2). (7.1) + (5.1) yields clauses of the form:

$$(8.1) \quad Q_1(y_1), Q_2(y_2), Q_3^{R_3}(y_1), Q_4^{R_4}(y_2), Q_5(y_3), Q_6(y_4), R_1(y_3, y_1), R_2(y_4, y_2) \\ \Rightarrow \underline{Q_7(f(y_3, y_4))}$$

At this point, we use the property of R and split the clause:

$$\exists y_1. Q_1(y_1) \wedge Q_3^{R_3}(y_1) \wedge R_1(y_3, y_1) \models Q_1^{R_4}(y_1) \wedge Q_3^{R_5}(y_1)$$

Hence clauses (9.1) can be rewritten into clauses of the form:

$$(8.1) \quad Q_1^{R_1}(y_1), Q_3^{R_3}(y_1), Q_5(y_1), Q_2^{R_2}(y_2), Q_4^{R_4}(y_2), Q_6(y_2) \Rightarrow \underline{Q_7(f(y_1, y_2))}$$

Finally, if we let \mathcal{Q} be the set of predicate symbols consisting of

- Symbols S_i
- Symbols Q_i
- Symbols $Q_i^{R_j}$

For every subset \mathcal{S} of \mathcal{Q} , we introduce a propositional variable $\mathbf{NE}_{\mathcal{S}}$. Clauses are split, introducing new propositional variables (or predicate symbols $Q_i^{R_j}$) in such a way that in all clauses except split clauses, the variables occurring on the left, also occur on the right of the clause. And, in split clauses, there is only one variable occurring on the left and not on the right.

We let \mathcal{C} be the set of clauses obtained by repeated applications of resolution with splitting, with the above selection strategy (a priori \mathcal{C} could be infinite). We claim that all

generated clauses are of one of the following forms (Where the P_i 's and the P'_i 's belong to \mathcal{Q} , Q 's states might actually be $Q_i^{R_j}$)

1. Pop clauses. (the original clauses, which are not subsumed by the new clauses):

$$\begin{aligned} (3) \quad & Q_1(f(y_1, y_2)), \underline{\mathbf{NE}_{Q_2}} \Rightarrow Q_3(y_1) \\ (4) \quad & Q_1(f(y_1, y_2)), \underline{\mathbf{NE}_{Q_2}} \Rightarrow Q_3(y_2) \\ (3b) \quad & \underline{Q_1(f(y_1, y_2))} \Rightarrow Q_2(y_1) \\ (4b) \quad & \underline{Q_1(f(y_1, y_2))} \Rightarrow Q_2(y_2) \end{aligned}$$

Note that, clause (1) is a particular case of the alternating clauses below, since it can be written

$$Q_1(y_1), Q_2^R(y_1) \Rightarrow Q_3(y_1)$$

2. Push clauses.

$$\begin{aligned} (\mathbf{P}_1) \quad & P_1(x), \dots, P_n(x), P'_1(y), \dots, P'_m(y) \Rightarrow \underline{Q(f(x, y))} \\ (\mathbf{P}_2) \quad & \Rightarrow \underline{P(a)} \\ (\mathbf{P}_3) \quad & \underline{\mathbf{NE}_S}, P_1(x), \dots, P_n(x), P'_1(y), \dots, P'_m(y) \Rightarrow \underline{Q(f(x, y))} \\ (\mathbf{P}_4) \quad & \underline{\mathbf{NE}_S} \Rightarrow \underline{Q(a)} \end{aligned}$$

3. Intermediate clauses.

$$\begin{aligned} (\mathbf{l}_1) \quad & P_1(x), \dots, P_n(x), P'_1(y), \dots, P'_m(y), \underline{P''_1(f(x, y))}, \dots, \underline{P''_k(f(x, y))} \Rightarrow Q(f(x, y)) \\ (\mathbf{l}_2) \quad & \underline{P_1(a)}, \dots, \underline{P_n(a)} \Rightarrow Q(a) \\ (\mathbf{l}_3) \quad & S_1(x_1), S_2(x_2), \underline{Q_1(a)} \Rightarrow Q_2(g(x_1, x_2)) \\ (\mathbf{l}_4) \quad & \underline{Q_1(a)} \Rightarrow Q_2(b) \end{aligned}$$

4. Alternating clauses.

$$\begin{aligned} (\mathbf{A}_1) \quad & \underline{\mathbf{NE}_S}, P_1(x), \dots, P_n(x) \Rightarrow Q(x) \\ (\mathbf{A}_2) \quad & \underline{P_1(x)}, \dots, \underline{P_n(x)} \Rightarrow Q(x) \end{aligned}$$

In addition, we have clauses obtained by splitting:

5. Split clauses.

$$\begin{aligned} (\mathbf{S}_1) \quad & \underline{R_j(x, y)}, Q_i(y) \Rightarrow Q_i^{R_j}(x) \\ (\mathbf{S}_{1b}) \quad & \underline{R_j(y, x)}, Q_i(y) \Rightarrow Q_i^{-R_j}(x) \\ (\mathbf{S}_2) \quad & R_1(x_1, y_1), R_2(x_2, y_2), \underline{Q_i(f(y_1, y_2))} \Rightarrow Q_i^{\pm R_j}(g(x_1, x_2)) \\ (\mathbf{S}_3) \quad & S_1(x), S_2(y), \underline{Q_i(f(x, y))} \Rightarrow Q_i^{\pm R_j}(a) \\ (\mathbf{S}_4) \quad & \underline{P_1(x)}, \dots, \underline{P_n(x)} \Rightarrow \mathbf{NE}_{\{P_1, \dots, P_n\}} \\ (\mathbf{S}_5) \quad & P_1(x), \dots, P_n(x), P'_1(y), \dots, P'_m(y), \underline{P''_1(f(x, y))}, \dots, \underline{P''_k(f(x, y))} \Rightarrow \mathbf{NE}_S \end{aligned}$$

6. Propositional clauses.

$$\begin{array}{l}
(\mathbf{E}_1) \quad \underline{\mathbf{NE}_{S_1}}, \dots, \underline{\mathbf{NE}_{S_n}} \Rightarrow \mathbf{NE}_S \\
(\mathbf{E}_2) \quad \underline{\hspace{10em}} \Rightarrow \underline{\mathbf{NE}_S} \\
(\mathbf{E}_3) \quad \underline{P_1(a)}, \dots, \underline{P_n(a)} \Rightarrow \underline{\mathbf{NE}_S}
\end{array}$$

Every resolution step using the selection strategy of two of the above clauses yield a clause in the above set

POP+PUSH: yields an alternating clause (\mathbf{A}_1) and a split clause (\mathbf{S}_4).

INT + PUSH: yields a Push clause or an intermediate clause

alternating + PUSH: yields an intermediate clause (\mathbf{I}_1) or (\mathbf{I}_2).

split + R : yields a split clause (\mathbf{S}_2) or (\mathbf{S}_3) or an intermediate clause (\mathbf{I}_3) or (\mathbf{I}_4).

(\mathbf{S}_2) + PUSH: yields clauses (\mathbf{S}_1) and push clauses. Note that here, we use the property of the relation R to split clauses, which may involve predicates $Q_i^{R_j}$.

(\mathbf{S}_3) + PUSH: yields push clause and split clauses (\mathbf{S}_4).

(\mathbf{S}_4) + PUSH: yields split clauses (\mathbf{S}_5) or propositional clause (\mathbf{E}_3).

(\mathbf{S}_5) + PUSH: yields split clauses (\mathbf{S}_5) or propositional clause (\mathbf{E}_1).

It follows that all clauses of \mathcal{C} are of the above form. Since there are only finitely many such clauses, \mathcal{C} is finite and computed in finite (exponential) time.

Now, we let \mathcal{A} be the alternating tree automaton defined by clauses (\mathbf{P}_1) and (\mathbf{P}_2) (and automata clauses defining the S states). Let, for any state Q , $\llbracket Q \rrbracket_{\mathcal{A}}$ be the language accepted in state Q by \mathcal{A} . We claim that $\llbracket Q \rrbracket = \llbracket \mathcal{A} \rrbracket$.

To prove this, we first show (the proof is omitted here) that $\mathbf{NE}_{\{P_1, \dots, P_n\}}$ is in \mathcal{C} iff $\llbracket P_1 \rrbracket_{\mathcal{A}} \cap \dots \cap \llbracket P_n \rrbracket_{\mathcal{A}} \neq \emptyset$.

Then observe that $\llbracket Q \rrbracket$ is also the interpretation of Q in the least Herbrand model of \mathcal{C} : indeed, all computations yielding \mathcal{C} are correct. Since $\llbracket Q \rrbracket_{\mathcal{A}} \subseteq \llbracket Q \rrbracket$ is trivial, we only have to prove the converse inclusion. For every $t \in \llbracket Q \rrbracket$ there is a proof of $Q(t)$ using the clauses in \mathcal{C} .

Assume, by contradiction, that there is a term t and a predicate symbol Q such that all proofs of $Q(t)$ using the clauses in \mathcal{C} involve at least a clause, which is not an automaton clause. Then, considering an appropriate sub-proof, there is a term u and a predicate symbol P such that all proofs of $P(u)$ involve at least one non-automaton clause and there is a proof of $P(u)$ which uses exactly one non-automaton clause, at the last step of the proof.

We investigate all possible cases for the last clause used in the proof of $P(u)$ and derive a contradiction in each case.

Clause \mathbf{I}_1 : The last step of the proof is

$$\frac{P_1(u_1), \dots, P_n(u_1), P'_1(u_2), \dots, P'_m(u_2), P''_1(f(u_1, u_2)), \dots, P''_k(f(u_1, u_2))}{P(f(u_1, u_2))}$$

and we assume $u = f(u_1, u_2)$. Assume also that, among the proofs we consider, k is minimal. (If $k = 0$ then we have a push clause, which is supposed not to be the case).

By hypothesis, for all i , $u_1 \in \llbracket P_i \rrbracket_{\mathcal{A}}$, $u_2 \in \llbracket P'_i \rrbracket_{\mathcal{A}}$ and $f(u_1, u_2) \in \llbracket P''_i \rrbracket_{\mathcal{A}}$. In particular, if we consider the last clause used in the proof of $P''_k(u)$:

$$Q_1(x), \dots, Q_r(x), Q'_1(y), \dots, Q'_s(y) \Rightarrow P''_k(f(x, y))$$

belongs to \mathcal{C} . Then, overlapping this clause with the above clause l_1 , the following clause belongs also to \mathcal{C} :

$$P_1(x), \dots, P_n(x), Q_1(x), \dots, Q_r(x), \\ P'_1(y), \dots, P'_m(y), Q'_1(y), \dots, Q'_s(y), P''_1(f(x, y)), \dots, P''_{k-1}(f(x, y)) \Rightarrow P(f(x, y))$$

and therefore we have another proof of $P(u)$:

$$\frac{P_1(u_1), \dots, P_n(u_1), Q_1(u_1), \dots, Q_r(u_1) \\ P'_1(u_2), \dots, P'_m(u_2), Q'_1(u_2), \dots, Q'_s(u_2), P''_1(f(u_1, u_2)), \dots, P''_{k-1}(f(u_1, u_2))}{P(f(u_1, u_2))}$$

which contradicts the minimality of k .

Clause (A₁): The last step of the proof is

$$\frac{P_1(u), \dots, P_n(u)}{P(u)}$$

By hypothesis, the proofs of $P_i(u)$ only use automata clauses: $\forall i. u \in \llbracket P_i \rrbracket_{\mathcal{A}}$. Let the push rule

$$Q_1(x), \dots, Q_m(x), Q'_1(y), \dots, Q'_p(y) \Rightarrow P_n(f(x, y))$$

be the last clause used in the proof of $P(u)$. Overlapping this clause and the clause **A₁** above, there is another clause in \mathcal{C} yielding a proof of $P(u)$:

$$Q_1(x), \dots, Q_m(x), Q'_1(y), \dots, Q'_p(y), P_1(f(x, y)), \dots, P_{n-1}(f(x, y)) \Rightarrow P(f(x, y))$$

And we are back to the case of l_1 .

Clause (3b):

$$\frac{Q_1(f(u, t))}{P(u)}$$

By hypothesis $f(t, u) \in \llbracket Q_1 \rrbracket_{\mathcal{A}}$. Hence there is a push clause

$$P_1(x), \dots, P_n(x), P'_1(y), \dots, P'_m(y) \Rightarrow Q_1(f(x, y))$$

such that $t \in \llbracket P_1 \rrbracket_{\mathcal{A}} \cap \dots \cap \llbracket P_n \rrbracket_{\mathcal{A}}$ and $u \in \llbracket P'_1 \rrbracket_{\mathcal{A}} \cap \dots \cap \llbracket P'_m \rrbracket_{\mathcal{A}}$. By resolution on the clause (3b), there is also in \mathcal{C} a clause

$$P_1(x), \dots, P_n(x), \mathbf{NE}_{\{P'_1, \dots, P'_m\}} \Rightarrow Q(x)$$

However, since $\llbracket P'_1 \rrbracket_{\mathcal{A}} \cap \dots \cap \llbracket P'_m \rrbracket_{\mathcal{A}} \neq \emptyset$, $\mathbf{NE}_{\{P'_1, \dots, P'_m\}}$ is also in \mathcal{C} and, by resolution again

$$P_1(x), \dots, P_n(x) \Rightarrow Q(x)$$

is a clause of \mathcal{C} .

Then we are back to the case of **A₁**.

Clause (3): The last step of the proof is

$$\frac{Q_1(f(u, t)) \quad \mathbf{NE}_{Q_2}}{P(u)}$$

Since $\mathbf{NE}_{Q_2} \in \mathcal{C}$ in this case, by saturation of \mathcal{C} , there is a clause $Q_1(x, y) \Rightarrow Q(x)$ in \mathcal{C} , and we are back to the case of (3b).

Other cases: they are quite similar to the previous ones. Let us only consider the case of clause (\mathbf{S}_2) , which is slightly more complicated.

$$\frac{R_1(u_1, v_1) \ R_2(u_2, v_2) \ Q_i(f(v_1, v_2))}{Q_i^{R_j}(g(u_1, u_2))}$$

Assume moreover that $u = g(u_1, u_2)$ is a minimal size term such that, for some Q_i, R_j , $Q_i^{R_j}(u)$ is provable using as a last step an inference \mathbf{S}_2 , and is not provable by automata clauses only,

As before, we consider the overlap between \mathbf{S}_2 and a push clause. We get

$$R_1(x_1, y_1), R_2(x_2, y_2), P_1(y_1), \dots, P_n(y_1), P'_1(y_2), \dots, P'_m(y_2) \Rightarrow Q_i^{R_j}(g(x_1, x_2))$$

Hence, the following clauses belong to \mathcal{C} (when P_i, P'_i are not themselves predicates Q^R ; otherwise, we have to use the property on R relations and split in another way, using the S_\top predicate, as shown later):

$$\begin{aligned} \frac{R_1(x_1, y_1), P_i(y_1)}{P_1^{R_1}(x_1)} &\Rightarrow P_i^{R_1}(x_1) \\ \frac{R_2(x_2, y_2), P'_i(y_2)}{P_1^{R_2}(x_2)} &\Rightarrow P'_i^{R_2}(x_2) \\ P_1^{R_1}(x_1), \dots, P_n^{R_1}(x_1), P_1^{R_2}(x_2), \dots, P_m^{R_2}(x_2) &\Rightarrow \underline{Q_i^{R_j}(g(x_1, x_2))} \end{aligned}$$

and we have the following proof of $g(u_1, u_2)$:

$$\frac{\frac{R_1(u_1, v_1) \ P_1(v_1)}{P_1^{R_1}(u_1)} \quad \dots \quad \frac{R_1(u_1, v_n) \ P_n(v_1)}{P_n^{R_1}(u_1)} \quad \frac{R_2(u_2, w_1) \ P'_2(w_1)}{P_1^{R_2}(u_2)} \quad \dots \quad \frac{R_2(u_2, w_m) \ P'_m(w_m)}{P_m^{R_2}(u_2)}}{Q_i^{R_j}(g(u_1, u_2))}$$

Now, by overlapping again $R_1(x_1, y_1)$ and $R_2(x_2, y_2)$ with their defining clause, we compute “shortcut clauses” belonging to \mathcal{C} and get another proof (for instance assuming $v_1 = f(v_{11}, v_{12})$ and $u_1 = h(u_{11}, u_{12})$):

$$\frac{\frac{R_{11}(u_{11}, v_{11}) \ R_{12}(u_{12}, v_{12}) \ P_1(f(v_{11}, v_{12}))}{P_1^{R_1}(u_1)} \quad \dots \quad \frac{R_2(u_2, w_1) \ P'_2(w_1)}{P_1^{R_2}(u_2)} \quad \dots \quad \frac{R_2(u_2, w_m) \ P'_m(w_m)}{P_m^{R_2}(u_2)}}{Q_i^{R_j}(g(u_1, u_2))}$$

By minimality of u , $u_1 \in \llbracket P_1^{R_1} \rrbracket_{\mathcal{A}}$. Similarly, for every i , $u_1 \in \llbracket P_i^{R_1} \rrbracket_{\mathcal{A}}$. $u_2 \in \llbracket P_i^{R_2} \rrbracket_{\mathcal{A}}$ and it follows that $g(u_1, u_2) \in \llbracket Q_i^{R_j} \rrbracket_{\mathcal{A}}$.

Finally, let us consider the case where some P_i is itself a predicate symbol Q^R , in which case we do not have a predicate $(Q^R)^{R_1}$. We use then the assumed property of the predicates R_i : $R_1(x, y) \wedge R(y, z) \models R'_1(x, y) \wedge R'(x, z)$, hence

$$(\exists u, \exists v. R_1(x, u) \wedge R(u, v) \wedge Q(v)) \models (\exists u. R_1(x, u) \wedge S_\top(u)) \wedge (\exists v. R(x, v) \wedge Q(v))$$

Hence we need two split clauses instead of one:

$$\begin{aligned} R'_1(x, y) &\Rightarrow S_\top^{R'_1}(x) \\ R'(x, y), Q(y) &\Rightarrow Q^{R'}(x) \end{aligned}$$

And $R_1(x_1, y_1), Q^R(y_1)$ is replaced with $S_{\top}^{R'}(x_1), Q^{R'}(x_1)$. Note that such a transformation is not necessary when there is a single transitive binary relation, as in our application: then $R(x, y) \wedge Q^R(y)$ is simply replaced with $Q^R(x)$.

To sum up: if there is a proof of $P(u)$ using clauses of \mathcal{C} , then, by saturation of the clauses of \mathcal{C} w.r.t. overlaps with push clauses, we can rewrite the proof into a proof using push clauses only: $u \in \llbracket P \rrbracket_{\mathcal{A}}$. This proves that $\llbracket P \rrbracket = \llbracket P \rrbracket_{\mathcal{A}}$.

Finally, it is easy (and well-known) to compute a standard bottom-up automaton accepting the same language as an alternating automaton; this only requires a subset construction. That is why the language accepted by our two-way automata with structural equality constraints is actually a recognizable language. The overall size of the resulting automaton (and its computation time) are simply exponential, but we know that, already for alternating automata, we cannot do better.