Hubert Comon-Lundh
Florent Jacquemard
Nicolas Perrin

Tree Automata with Memory, Visibility and Structural Constraints

Research Report LSV-07-01

January 2007

Laboratoire Spécification et Vérification

Ecole Normale Supérieure de Cachan
61, avenue du Président Wilson
94235 Cachan Cedex France

CENTRE NATIONAL DE LA RECHERCHE SCIENTIFIQUE

Tree Automata with Memory, Visibility and Structural Constraints

Hubert Comon-Lundh[1], Florent Jacquemard[2] and Nicolas Perrin[3]

[1] LSV & ENS Cachan, comon@lsv.ens-cachan.fr
[2] INRIA Futurs & LSV, florent.jacquemard@inria.fr
[3] ENS Lyon 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, inclusion and universality are decidable for VTAM. Moreover, we propose an extension of VTAM whose transitions may be constrained by structural equality and disequality tests between memories, and show that this extension preserves the good closure and decidability properties.

1 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) 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 stack with a tree.

Tree automata with one memory are introduced in [4]. They compute bottom-up on a tree, with an auxiliary memory carrying a tree. 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 memory also generalize tree automata with equality tests between brothers [3].

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. 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 introduce the new formalism of Visibly Tree Automata with Memory (VTAM), extending on one hand Visibly pushdown languages to trees, including a tree structure instead of a stack (following former approaches [9, 14, 7]). 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 3 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.

In a second part of the paper (Section 4), we consider 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 such 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 equality tests and structural equality tests. 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.

We consider VTAM with structural equality tests, since the determinization and closure properties of Section 3 carry over this generalization, which we show in Section 4.4. Finally, we give in Section 4.5 some examples of languages that can be recognized by VTAM with structural equality and disequality tests: well-balanced binary trees, red-black trees, powerlists...

Generalisations of pushdown automata to trees (both for input and stack) are proposed in [9, 14, 7]. 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.

2 Preliminaries

Term algebra. A *signature* Σ is a finite set of function symbols with arity, denoted by f, $g \ldots$ 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 $vars(t)$. A *substitution* σ is a mapping from $\mathcal{X}$ to $\mathcal{T}(\Sigma, \mathcal{X})$ such that $\{x | \sigma(x) \neq x\}$, 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* $Pos(t)$ 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 $t[u]_p$.

Rewriting. We assume standard definitions and notations for term rewriting [8]. A *term rewriting system* (TRS) over a signature Σ is a finite set of rewrite rules $\ell \to r$, where $\ell \in \mathcal{T}(\Sigma, \mathcal{X})$ and $r \in \mathcal{T}(\Sigma, vars(\ell))$. A term $t \in \mathcal{T}(\Sigma, \mathcal{X})$ rewrites to s by a TRS $\mathcal{R}$ (denoted $t \to_{\mathcal{R}} s$) if there is a rewrite rule $\ell \to r \in \mathcal{R}$, 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 $\to_{\mathcal{R}}$ is denoted $\xrightarrow[\mathcal{R}]{*}$.

Tree Automata. Following definitions and notation of [5], 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 as a tuple $(Q, Q_{\mathsf{f}}, \Delta)$ where Σ is the computation signature, Q is a finite set of nullary state symbols, disjoint from Σ, $Q_{\mathsf{f}} \subseteq Q$ is the subset of final states and Δ is a set of rewrite rules of the form: $f(q_1, \ldots, q_n) \to q$, where $f \in \Sigma$ and $q_1, \ldots, q_n \in Q$. A term t is *accepted* 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.

3 Visibly Tree Automata with Memory

We propose in this section a subclass of the tree automata with one memory [4] which is stable under Boolean operations and has a decidable emptiness problem.

3.1 Definition of VTAM

Tree automata have been extended [4] to carry an unbounded information stored in a tree structure along the states in computations. This information is called *memory* in [4] and will 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 contains 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 differs from the one of [4]) which shall be restricted later on.

Definition 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 $f\big(q_1(m_1),\ldots,q_n(m_n)\big) \to q(m)$ where $f \in \Sigma_n$, $q_1,\ldots,q_n,q \in Q$ and $m_1,\ldots,m_n,m \in \mathcal{T}(\Gamma,\mathcal{X})$.*

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:

$$L(\mathcal{A},q) = \big\{t \;\big|\; t \xrightarrow[\Delta]{*} q(m),\ m \in \mathcal{T}(\Gamma)\big\}$$
$$M(\mathcal{A},q) = \big\{m \;\big|\; t \xrightarrow[\Delta]{*} q(m),\ t \in \mathcal{T}(\Sigma)\big\}.$$

The language of $\mathcal{A}$ is the union of languages of $\mathcal{A}$ is 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 subcategories) of operations on memory. We assume below a computation step at some position p of a term, where memories $m_1, \ldots, m_n$ have been reached at the positions immediately below p:

- PUSH: the new current memory m is build with a symbol $h \in \Gamma_n$ *pushed* at the top of memories reached: $f\big(q_1(m_1), \ldots, q_n(m_n)\big) \to q\big(h(m_1, \ldots, m_n)\big)$. 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 the memories reached: $f\big(\ldots, q_i(g(m'_1, \ldots, m'_k)), \ldots\big) \to q(m'_j)$. This corresponds to a function's *return* in a program.
- INT (internal): the new current memory is one of the memories reached: $f\big(q_1(m_1), \ldots, q_n(m_n)\big) \to q(m_i)$. This corresponds to an internal operation (neither call nor return) in a function of a program.

Next, we adhere to the *visibility* condition of [2]. The idea behind this restriction, which was already in [11], is that the symbol read (in a term in our case and [1], in a word in the case of [2]) by an automaton corresponds to an instruction of a program, and hence belongs to one of the three above families (call, return and 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. We assume from now on for the sake of simplicity that 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 arity. The signature Σ is partitioned in eight subsets:

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

The eight corresponding transition categories are defined formally in Figure 1. In this figure, $\bot$ is a special constant symbol in Γ, used to represent an empty memory. Note that the other constant symbols of Γ are not relevant since they can not be pushed or popped. Note that each POP rule has a variant which read an empty memory.

Definition 2. *A* visibly tree automaton with memory *(VTAM) on Σ is a TAM (Γ, Q, Q_f, Δ) such that every rule of Δ belongs to one of the above categories* PUSH, POP_{11}, POP_{12}, POP_{21}, POP_{22}, INT_0, INT_1, INT_2.

A VTAM $\mathcal{A}$ is said *complete* if every term of $\mathcal{T}(\Sigma)$ belong 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.

$$
\begin{array}{lllll}
\mathsf{PUSH} & f_2\big(q_1(y_1), & q_2(y_2)\big) & \to q\big(h(y_1,y_2)\big) \\
\mathsf{POP}_{11} & f_3\big(q_1(h(y_{11},y_{12})), & q_2(y_2)\big) & \to q(y_{11}) \\
 & f_3\big(q_1(\bot), & q_2(y_2)\big) & \to q(\bot) \\
\mathsf{POP}_{12} & f_4\big(q_1(h(y_{11},y_{12})), & q_2(y_2)\big) & \to q(y_{12}) \\
 & f_4\big(q_1(\bot), & q_2(y_2)\big) & \to q(\bot) \\
\mathsf{POP}_{21} & f_5\big(q_1(y_1), & q_2(h(y_{21},y_{22}))\big) & \to q(y_{21}) \\
 & f_5\big(q_1(y_1), & q_2(\bot)\big) & \to q(\bot) \\
\mathsf{POP}_{22} & f_6\big(q_1(y_1), & q_2(h(y_{21},y_{22}))\big) & \to q(y_{22}) \\
 & f_6\big(q_1(y_1), & q_2(\bot)\big) & \to q(\bot) \\
\mathsf{INT}_0 & a & & \to q(\bot) \\
\mathsf{INT}_1 & f_7\big(q_1(y_1), & q_2(y_2)\big) & \to q(y_1) \\
\mathsf{INT}_2 & f_8\big(q_1(y_1), & q_2(y_2)\big) & \to q(y_2)
\end{array}
$$

where $q_1, \ldots, q_n \in Q$, y_1, y_2 are distinct variables of $\mathcal{X}$, $h \in \Gamma_2$, $a \in \Sigma_{\mathsf{INT}_0}$, and every f_i is in the corresponding partition of Σ ($f_2 \in \Sigma_{\mathsf{PUSH}}$, $f_3 \in \Sigma_{\mathsf{POP}_{11}}$, etc).

Fig. 1. VTAM transition categories

3.2 Determinism

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

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

Theorem 1. *For every VTAM $\mathcal{A} = (\Gamma, Q, Q_\mathsf{f}, \Delta)$ there exists a deterministic VTAM $\mathcal{A}^{det} = (\Gamma^{det}, Q^{det}, Q^{det}_\mathsf{f}, \Delta^{det})$ such that $L(\mathcal{A}) = L(\mathcal{A}^{det})$, where $|Q^{det}|$ and $|\Gamma^{det}|$ both are $O\big(2^{|Q|^2}\big)$.*

Proof. We follow the technique of [2] for the determinization of VPA: 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 more uniform construction of the rules of $\mathcal{A}^{det}$ for each symbol of Σ. As we shall see below, $\mathcal{A}^{det}$ wont need to keep track of the contents of memory (of $\mathcal{A}$) during its computation, it will only need to memorize information on the reachability of states of $\mathcal{A}$, following the path 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^{det}_f 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.

INT. For every $f \in \Sigma_{\mathsf{INT}_1}$, we have the following rules in Δ^{det}:

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

where $R = \big\{q \;\big|\; \exists q_1 \in R_1, q_2 \in R_2, f\big(q_1(y_1), q_2(y_2)\big) \to q(y_1) \in \Delta\big\}$ and S is the update of S_1 according to the INT_1-transitions of Δ (when $b_1 = 1$, the case $b_1 = 0$ is similar):

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

The case $f \in \Sigma_{\mathsf{INT}_2}$ is similar.
We consider memory symbols made of pairs of states and PUSH symbols:

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

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

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

where $R = \big\{q \;\big|\; \exists q_1 \in R_1, q_2 \in R_2, h \in \Gamma, f\big(q_1(y_1), q_2(y_2)\big) \to q\big(h(y_1, y_2)\big) \in \Delta\big\}$ and Id_Q is $\big\{(q, q) \;\big|\; q \in Q\big\}$ is used to initialize the memorization of state reachability from the position of the symbol f, and $p := \big\langle\langle b_1, R_1, S_1\rangle, \langle b_2, R_2, S_2\rangle, f\big\rangle$. 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 $f \in \Sigma_{\mathsf{POP}_{11}}$, we have the following rules in Δ^{det}:

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

where $h = \langle Q_1, Q_2, g\rangle$, with $Q_1 = \langle b_1', R_1', S_1'\rangle \in Q^{det}$, $Q_2 = \langle b_2', R_2', S_2'\rangle \in Q^{det}$.

$$R = \left\{ q \;\middle|\; \begin{array}{l} \exists q_1' \in R_1', q_2' \in R_2', (q_0, q_1) \in S_1, q_2 \in R_2, h \in \Gamma, g\big(q_1'(y_1), q_2'(y_2)\big) \to \\ \quad q_0\big(h(y_1, y_2)\big) \in \Delta, f\big(q_1(h(y_{11}, y_{12})), q_2(y_2)\big) \to q(y_{11}) \in \Delta \end{array} \right\}$$

$$S = \left\{ (q, q') \;\middle|\; \begin{array}{l} \exists q_1' \in S_1'(q), q_2' \in R_2', (q_0, q_1) \in S_1, q_2 \in R_2, h \in \Gamma, g\big(q_1'(y_1), q_2'(y_2)\big) \\ \to q_0\big(h(y_1, y_2)\big) \in \Delta, f\big(q_1(h(y_{11}, y_{12})), q_2(y_2)\big) \to q'(y_{11}) \in \Delta \end{array} \right\}$$

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 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 $L(\mathcal{A}) = L(\mathcal{A}^{det})$. □

3.3 Closure Properties

The tree automata with one memory of [4] 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. *The class of tree languages of VTAM is closed under Boolean operations (union, intersection, complement).*

Proof. (sketch, see Appendix A for the complete constructions).

For the union of two VTAM languages, we construct a VTAM 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.

For the intersection, we construct a VTAM 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. The rule set of the intersection VTAM 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. Note that such an operation 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 synchronise memory operations on the same symbols.

For the complement, we use the construction of Theorem 1 and take the complement of the final state set of the VTAM obtained. □

3.4 Decision Problems

Every VTAM is a particular case of tree automaton with one memory of [4]. Since the emptiness problem (whether the language accepted is empty or not) is decidable for this latter class, it is also decidable for VTAM. In comparison, the emptiness is decidable for nondeterministic visibly pushdown (top-down) tree automata (N-VPTA) of [1] but the class of languages of infinite trees that they define is not closed under complement. The alternating version of these automata (VPTA, [1]) is closed under Boolean operations but has an undecidable emptiness problem. We propose below a proof of decidability of emptiness which follows the same lines as [4].

Theorem 3. *The emptiness problem is decidable in EXPTIME for VTAM. The universality and inclusion problem are decidable in 2-EXPTIME for VTAM.*

Proof. Assume given a VTAM $\mathcal{A} = (\Gamma, Q, Q_{\mathsf{f}}, \Delta)$. 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. We show that each $M(\mathcal{A}, q)$ is recognized by an alternating two-way automaton, hence is regular (see e.g. [5]). We can construct in exponential time a TA $\mathcal{A}_q$ of size exponential in the size of $\mathcal{A}$ and accepting $L(\mathcal{A}, q)$. A proof of a more general result will stated in Lemma 1 and proved in Appendix C.

As usual, 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 $L(\mathcal{A}_1) \cap L(\overline{\mathcal{A}_2}) = \emptyset$. Since these operations require a determinization of a given VTAM first, these problems can be decided in 2-EXPTIME for VTAM. □

4 Visibly Tree Automata with Memory and Constraints

In the late eighties, some models of tree recognizers obtained by adding equality and disequality constraints in transitions of tree automata have been proposed in order to solve problems with term rewrite systems or constraints systems with non-linear patterns (terms with multiple occurrence of the same variable). The tree automata of [3] for instance can perform equality and disequality test between subterms of the term read located at brother positions.

In the case of tree automata with memory, we shall apply constraints to the contents of the memory. Indeed, each step of a bottom-up computation 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, then give sufficient conditions for the emptiness decidability and show some relevant examples which satisfy these conditions. Finally, we study in Section 4.4 the particular case of VTAM with structural equality constraints. They enjoy not only decision properties but also good closure properties.

4.1 Definitions

Assume given a fixed equivalence relation R on $\mathcal{T}(\Gamma)$. We consider now four new categories for the symbols of Σ: INT_1^R, INT_2^R, $\mathsf{INT}_1^{\neg R}$, $\mathsf{INT}_2^{\neg R}$, in addition to the eight previous categories of page 5. The four new categories correspond to the the constrained versions of the transition rules INT_1 and INT_2 presented in Figure 2.

$$\begin{array}{lll}
\mathsf{INT}_1^R & f_9\big(q_1(y_1),\ q_2(y_2)\big) & \xrightarrow{y_1\ R\ y_2} q(y_1) \\
\mathsf{INT}_2^R & f_{10}\big(q_1(y_1),\ q_2(y_2)\big) & \xrightarrow{y_1\ R\ y_2} q(y_2) \\
\mathsf{INT}_1^{\neg R} & f_{11}\big(q_1(y_1),\ q_2(y_2)\big) & \xrightarrow{y_1\ \neg R\ y_2} q(y_1) \\
\mathsf{INT}_2^{\neg R} & f_{12}\big(q_1(y_1),\ q_2(y_2)\big) & \xrightarrow{y_1\ \neg R\ y_2} q(y_2)
\end{array}$$

Fig. 2. New transition categories for $\mathrm{VTAM}_{\neg R}^{R}$.

We will not extend the rules PUSH and POP with constraints for some reasons explained below. A ground term t rewrites to s by a constrained rule $f\big(q_1(y_1), q_2(y_2)\big) \xrightarrow{y_1\ c\ y_2} r$ (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. *A* visibly tree automaton with memory and constraints *(*$\mathrm{VTAM}^R_{\neg R}$*) 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*,* $\mathsf{INT}_1^{\neg R}$*,* $\mathsf{INT}_2^{\neg R}$*.*

We denote VTAM^R the subclass of $\mathrm{VTAM}^R_{\neg R}$ with positive constraints only, i.e. without transition rules in $\mathsf{INT}_1^{\neg R}$ or $\mathsf{INT}_2^{\neg R}$. The acceptance of terms of $\mathcal{T}(\Sigma)$ and languages of term and memories are defined and denoted as in Section 3.1.

The definition of *deterministic* $\mathrm{VTAM}^R_{\neg R}$ is based on the same conditions as for VTAM for the function symbols in categories of PUSH_0, PUSH, POP_{11}, $\ldots$, POP_{22}, INT_1, INT_2, and for the function symbols of INT_1^R, INT_2^R, $\mathsf{INT}_1^{\neg R}$, $\mathsf{INT}_2^{\neg R}$, we use the same conditions as for INT_1, INT_2: for all $f \in \Sigma_{\mathsf{INT}_1^R} \cup \Sigma_{\mathsf{INT}_2^R} \cup \Sigma_{\mathsf{INT}_1^{\neg R}} \cup \Sigma_{\mathsf{INT}_2^{\neg R}}$, for all $q_1, q_2 \in Q$, there is at most one rule in Δ with left-member $f\big(q_1(y_1), q_2(y_2)\big)$.

4.2 Emptiness decision

We propose here a generic theorem for emptiness decision. The idea of this theorem is that under some condition on R, the transition rules with negative constraints can be eliminated.

Theorem 4. *Let* R *be an equivalence relation satisfying these two properties:*

i. for all automaton $\mathcal{A}$ *of* VTAM^R *and for all state* q *of* $\mathcal{A}$*, the memory language* $M(\mathcal{A}, q)$ *is a regular tree language,*
ii. the size of every equivalence class of R *is bounded, and its elements can be enumerated.*

Then the emptiness problem is decidable for $\mathrm{VTAM}^R_{\neg R}$*.*

Proof. Let $\mathcal{A} = (\Gamma, R, Q, Q_f, \Delta)$ be a $\mathrm{VTAM}^R_{\neg R}$. We show (in Appendix B) that there exists a VTAM^R $A^+ = (\Gamma, R, Q^+, Q_f, \Delta^+)$ such that $Q \subseteq Q^+$, and for each $q \in Q$, $M(\mathcal{A}^+, q) = M(\mathcal{A}, q)$. The proof is by induction on the number n of rules with negative constraints (*i.e.* rules in categories $\mathsf{INT}_1^{\neg R}$ and $\mathsf{INT}_2^{\neg R}$) in Δ and uses the bound on the size of equivalence classes, condition *ii* of the theorem.

Using the condition i of the theorem, it follows that emptiness is decidable for $\mathcal{A}$, since by definition $L(\mathcal{A}, q)$ is empty iff $M(\mathcal{A}, q)$ is empty. □

We will see soon (Section 4.4) two examples of relations satisfying i and *ii*.

4.3 Regular tree relations

We first consider the general case 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 $\mathcal{T}\big((\Sigma \cup \{\bot\})^2\big)$, where $\bot$ is a new constant symbol (not in Σ). This coding is defined recursively by:

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

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 class of $\text{VTAM}^R_{\neg R}$ when R is a binary regular tree relation constitutes a nice and uniform framework. Note however the condition ii. of Theorem 4 is not always true in this case. Actually, it is too expressive.

Theorem 5. *The emptiness problem is undecidable for* VTAM^R *with some* R *based on a regular binary relation.*

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 $\Sigma_{\mathsf{INT}^R_1}$.

We construct a 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 is 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 and a transition applied at the top of a subterm $f(c_i, f(c_{i+1}, \dots))$ will 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. 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. □

4.4 Syntactic and structural equality and disequality constraints

We present now two examples of relations satisfying the conditions of Theorem 4. These results will be proved with the following crux Lemma.

Lemma 1. *Let R be a regular binary relation defined by a TA whose state set is $\{R_i \mid i = \{1..n\}\}$ and such that $\forall i, j \exists k, l\ R_i(x,y) \wedge R_j(y,z) \not\models R_k(x,y) \wedge R_l(x,z)$. Let $\mathcal{A} = (\Gamma, R, Q, Q_\mathsf{f}, \Delta)$ be a tree automaton with memory and constraints (not necessarily visibly). Then for every $q \in Q$, $M(\mathcal{A}, q)$ is regular.*

Proof. (Sketch, the complete proof can be found in Appendix C). We first observe that $M(\mathcal{A}, q)$ is the interpretation of q in the least Herbrand model of a set of Horn clauses computed from the rules Δ. We saturate this set of clauses by resolution with an selection and eager splitting. This saturation terminates, and the set of clauses corresponding to alternating automata transitions in the saturated set recognizes the language $M(\mathcal{A}, q)$, which is therefore regular. □

We first apply Lemma 1 to the class $\mathrm{VTAM}^{=}_{\neq}$ where $=$ denotes the equality between ground terms made of memory symbols.

Corollary 1. *The emptiness problem is decidable for* $\mathrm{VTAM}^{=}_{\neq}$.

Lemma 1 applies also to another class $\mathrm{VTAM}^{\equiv}_{\not\equiv}$, where $\equiv$ denotes structural term equality, defined recursively as the smallest equivalence relation 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.

Corollary 2. *The emptiness problem is decidable for* $\mathrm{VTAM}^{\equiv}_{\not\equiv}$.

A nice property of $\mathrm{VTAM}^{\equiv}_{\not\equiv}$ is that the construction for determinization of Section 3.2 still works for this class.

Theorem 6. *For every* $\mathrm{VTAM}^{\equiv}_{\not\equiv}$ $\mathcal{A} = (\Gamma, \equiv, Q, Q_\mathsf{f}, \Delta)$ *there exists a deterministic* $\mathrm{VTAM}^{\equiv}_{\not\equiv}$ $\mathcal{A}^{det} = (\Gamma^{det}, \equiv, Q^{det}, Q^{det}_\mathsf{f}, \Delta^{det})$ *such that* $L(\mathcal{A}) = L(\mathcal{A}^{det})$, *where* $|Q^{det}|$ *and* $|\Gamma^{det}|$ *both are* $O\big(2^{|Q|^2}\big)$.

Proof. We use the same construction as in the proof of Theorem 1, with a direct extension of the construction for INT to $\mathsf{INT}^{=}$ or $\mathsf{INT}^{\neq}$. 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. □

Theorem 7. *The class of tree languages of* $\mathrm{VTAM}^{\equiv}_{\not\equiv}$ *is closed under Boolean operations.*

Proof. We use the same constructions as in Theorem 2 (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 6). For instance, the product of the $\mathsf{INT}^{\equiv}_1$ rules $f_9\big(q_{11}(y_1), q_{12}(y_2)\big) \xrightarrow{y_1 \equiv y_2} q_1(y_1)$ and $f_9\big(q_{21}(y_1), q_{22}(y_2)\big) \xrightarrow{y_1 \equiv y_2} q_1(y_1)$, is $f_9\big(\langle q_{11}, q_{21}\rangle(y_1), \langle q_{12}, q_{22}\rangle(y_2)\big) \xrightarrow{y_1 \equiv y_2} \langle q_1, q_2\rangle(y_1)$. For the complementation, we use Theorem 6. □

Corollary 3. *The universality and inclusion problems are decidable for* $\mathrm{VTAM}^{\equiv}_{\not\equiv}$.

Proof. This is a consequence of Corollary 3 and Theorem 7. □

Constrained PUSH transitions. We did not consider a constrained extension of the rules PUSH. The main reason is that symbols of a new category $\mathsf{PUSH}^{\equiv}$, which test two memories for structural equality and then push a symbol on the top of them, permit construct a constrained 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 4, Lemma 1) of regularity of these languages for the classes considered.

4.5 Some $\mathrm{VTAM}^{\equiv}_{\not\equiv}$ languages

The regular tree languages and VPL are particular cases of VTAM languages. In some cases, the tree automata with equality and disequality tests between brothers [3] can be simulated by $\mathrm{VTAM}^{=}_{\neg=}$ which push all the symbol read up to (dis)equality tests. We present in this final section some other relevant examples of $\mathrm{VTAM}^{\equiv}_{\not\equiv}$ languages.

Well balanced binary trees. The $\mathrm{VTAM}^{\equiv}_{\not\equiv}$ with memory alphabet $\{f, \bot\}$, state set $\{q, q_\mathsf{f}\}$, unique final state q_f, and whose rules follow accepts the (non-regular) language of well balanced binary trees build with g (binary, in $\Sigma_{\mathsf{INT}^{\equiv}_1}$), f (binary, in Σ_{PUSH}) and a (constant in Σ_{INT_0}) with a g at the root position and only f's and a's below.

$$a \to q(\bot) \quad \begin{array}{l} f\big(q(y_1), q(y_2)\big) \\ g\big(q(y_1), q(y_2)\big) \end{array} \begin{array}{c} \to \\ \xrightarrow{y_1 \equiv y_2} \end{array} \begin{array}{l} q\big(f(y_1, y_2)\big) \\ q_\mathsf{f}(y_1) \end{array}$$

Powerlists. A powerlist [13] is roughly a list of length 2^n (for $n \geq 0$) whose elements are stored in the leaves of a balanced binary tree. This data structure has been used in [13] to specify data-parallel algorithms based on divide-and-conquer strategy and recursion (*e.g.* Batcher's merge sort and fast Fourier transform).

The following $\mathrm{VTAM}^{\equiv}_{\not\equiv}$ with memory alphabet $\{f, \bot\}$, state set $\{q, q_\mathsf{f}\}$ and unique final state q_f and whose rules follow accepts the language of powerlists of natural numbers presented in unary notation with the symbol s (binary, in Σ_{INT_2}) and 0 (constant in Σ_{INT_0}). We use artificially a successor symbol s of arity 2 instead of 1 as usual, because of the assumption that $\Sigma = \Sigma_0 \uplus \Sigma_2$ in Section 3.1 (2 for instance is written $s(0, s(0, 0))$). The other symbols are f (binary, in Σ_{PUSH}), and g (binary, in $\Sigma_{\mathsf{INT}^{\equiv}_1}$), used for the root of powerlist only (as above). The rules of the $\mathrm{VTAM}^{\equiv}_{\not\equiv}$ are the following:

$$\begin{array}{r} 0 \to q_0(\bot) \\ s(q_0(y_1), q_0(y_2)) \to q_0(y_2) \end{array} \quad \begin{array}{l} f\big(q_0(y_1), q_0(y_2)\big) \\ f\big(q(y_1), q(y_2)\big) \\ g\big(q(y_1), q(y_2)\big) \end{array} \begin{array}{c} \to \\ \to \\ \xrightarrow{y_1 \equiv y_2} \end{array} \begin{array}{l} q\big(f(y_1, y_2)\big) \\ q\big(f(y_1, y_2)\big) \\ q_\mathsf{f}(y_1) \end{array}$$

Note that only the f symbol is pushed on the memory. Therefore, only the upper structure of the powerlist is saved in the memory and tested at root position for structural equality. This way, we ensure that this upper part is well balanced, hence that the list has length 2^n.

Some equational properties of algebraic specifications of powerlists have been studied in the context of automatic induction theorem proving and sufficient completeness [12]. Tree automata with constraints have been acknowledged as a very powerful formalism in this context (see *e.g.* [6]). We therefore believe that a characterisation of powerlist (and the complement language) as $\mathrm{VTAM}^{\equiv}_{\not\equiv}$ for the automated verification of algorithms on this data structure.

Red-black trees. A red-black 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 check with standard TA rules. The fifth property make the language red-black trees not regular and we need $\mathrm{VTAM}^{\equiv}_{\not\equiv}$ rules to recognize it. It can be checked by pushing all the black nodes read, we use for this purpose a symbol $black \in \Sigma_{\mathsf{PUSH}}$. When a red node is read, the number of black nodes in both its sons are check to be equal (by a test $\equiv$ on the corresponding memories) and only one corresponding memory is kept. This is done with a symbol $red \in \Sigma_{\mathsf{INT}^{\equiv}_{1}}$. When a red 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. The structure test is done with an auxiliary symbol $aux \in \Sigma_{\mathsf{INT}^{\equiv}_{1}}$, located just above the *black* symbol. It means that the $\mathrm{VTAM}^{\equiv}_{\not\equiv}$ recognizes not exactly the red-black tree but a representation with additional nodes. This can be considered as already satisfying in the context of verification. In [10] 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, $\mathrm{VTAM}^{\equiv}_{\not\equiv}$ can also be used for similar purposes.

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 main contribution is then to extend this automaton model, constraining the transition rules with some regular conditions, while keeping decidability results. The structural equality and disequality tests appear to a be a good constraint class since we have then both decidability of emptiness and Boolean closure properties.

Acknowledgments

The authors wish to thank the reviewers at FOSSACS'07 for their useful 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/vptl.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. 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.
4. H. Comon and V. Cortier. Tree automata with one memory, set constraints and cryptographic protocols. *Theoretical Computer Science*, 331(1):143–214, Feb. 2005.
5. 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.
6. H. Comon and F. Jacquemard. Ground reducibility is exptime-complete. *Information and Computation*, 187(1):123–153, 2003.
7. J.-L. Coquidé, M. Dauchet, R. Gilleron, and S. Vágvölgyi. Bottom-up tree pushdown automata: classification and connection with rewrite systems. *Theoretical Computer Science*, 127(1):69–98, 1994.
8. N. Dershowitz and J.-P. Jouannaud. *Rewrite systems*, chapter Handbook of Theoretical Computer Science, Volume B, pages 243–320. Elsevier, 1990.
9. I. Guessarian. Pushdown tree automata. *Theory of Computing Systems*, 16(1):237–263, 1983.
10. 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.
11. 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.
12. D. Kapur. *Essays in Honor of Larry Wos*, chapter Constructors can be Partial Too. MIT Press, 1997.
13. J. Misra. Powerlist: A structure for parallel recursion. *ACM Transactions on Programming Languages and Systems*, 16(6):1737–1767, November 1994.
14. K. M. Schimpf and J. Gallier. Tree pushdown automata. *Journal of Computer and System Sciences*, 30(1):25–40, 1985.

Appendix

A Proof of Theorem 2 - Boolean closure for VTAM

Given two VTAM $\mathcal{A}_1 = (\Gamma_1, Q_1, Q_{\mathsf{f},1}, \Delta_1)$ and $\mathcal{A}_2 = (\Gamma_2, Q_2, Q_{\mathsf{f},2}, \Delta_2)$ on Σ, $\mathcal{A}_\cup$ defined by:

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

is a VTAM on Σ such that $L(\mathcal{A}_\cup) = L(\mathcal{A}_1) \cup L(\mathcal{A}_2)$, and $\mathcal{A}_\cap$, defined by:

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

is a VTAM on Σ such that $L(\mathcal{A}_\cup) = L(\mathcal{A}_1) \cap L(\mathcal{A}_2)$, where $\Delta_\cap$ is the smallest set of rules such that:

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

B Proof of Theorem 4 - Emptiness problem for $\mathrm{VTAM}^{R}_{\neg R}$

Let $\mathcal{A} = (\Gamma, R, Q, Q_\mathsf{f}, \Delta)$ be a $\mathrm{VTAM}^{R}_{\neg R}$. We show by induction on the number n of rules with negative constraints (*i.e.* rules in categories $\mathsf{INT}_1^{\neg R}$ and $\mathsf{INT}_2^{\neg R}$) in Δ that there exists a VTAM^R $A^+ = (\Gamma, R, Q^+, Q_\mathsf{f}, \Delta^+)$ such that $Q \subseteq Q^+$ and for each $q \in Q$, $M(\mathcal{A}^+, q) = M(\mathcal{A}, q)$. The proof of this result uses the condition *ii* of the theorem.

Using the condition i of the theorem, it follows that emptiness is decidable for $\mathcal{A}$, since by definition $L(\mathcal{A}, q)$ is empty iff $M(\mathcal{A}, q)$ is empty.
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), say:

$$f(q_1(y_1), q_2(y_2) \xrightarrow{y_1 \neg R\, y_2} q(y_1) \tag{1}$$

We use the following lemma:

Lemma 2. *Given $m_1, \ldots, m_k \in M(\mathcal{A}, q_2)$, if there exists $m_{k+1} \in M(\mathcal{A}, q_2) \setminus \{m_1, \ldots, m_k\}$, then we can build it.*

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 the $\mathcal{A}_i$ to $\mathcal{A}$, obtaining $\mathcal{A}'$ (we assume that the state sets of $\mathcal{A}_1, \ldots, \mathcal{A}_k, \mathcal{A}$ are disjoint, and that the states of $\mathcal{A}_1, \ldots, \mathcal{A}_k$ 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 (1), then we can apply instead:

$$f(q_1(y_1), p_i(y_2)) \xrightarrow{y_1\, R\, y_2} q(y_1) \tag{2}$$

We add the rule (2) to $\mathcal{A}'$ and obtain $\mathcal{A}''$. We can observe that $L(\mathcal{A}'') = L(\mathcal{A})$ and that $M(\mathcal{A}'', q_2) = M(\mathcal{A}, q_2)$. Let m_{k+1} be a term of $M(\mathcal{A}'', q_2) \setminus \{m_1, \ldots, m_k\}$ of minimal size (if one exists). The rule (1) is not used for the creation of m_{k+1} by $\mathcal{A}''$. Otherwise, the witness for y_2 in the application of this rule would be a term of $M(\mathcal{A}'', q_2) \setminus \{m_1, \ldots, m_k\}$ smaller than m_{k+1}. Therefore, $m_{k+1} \in M(\mathcal{A}'' \setminus (1), q_2)$. This automaton $\mathcal{A}''' = \mathcal{A}'' \setminus (1)$ has $n-1$ rules with negative constraints. Hence, by induction hypothesis, there is a VTAM^R $\mathcal{A}'''^+$ with m_{k+1} in its memory language $M(\mathcal{A}'''^+, q_2)$. By condition *i*, this language is regular and we can build m_{k+1} from a TA for this language. □

Now, let us come back to the proof that we can replace rule (1), while preserving the memory languages.
If $M(\mathcal{A}, q_2) = \emptyset$, then the rule (1) is useless and can be removed from $\mathcal{A}$ without changing its memory language.
Otherwise, let $m_1 \in M(\mathcal{A}, q_2)$ be built with Lemma 2 and let N be the cardinal of $[m_1]$. We apply N times the construction of Lemma 2. There are three cases:

- if we find more than N terms in $M(\mathcal{A}, q_2)$, then one of them, say m_k is not in $[m_1]$. Then (1) is useless for the point of view of memory languages: whatever value for y_1, we know an $y_2 in M(\mathcal{A}, q_2)$ which permits to fire the rule. Indeed, if $y_1 \in [m_1]$, then we can choose $y_2 = m_k$, and otherwise we choose $y_2 = m_1$. Hence (1) can be replaced without changing the memory language by:
$$f(q_1(y_1), q_0(y_2)) \to q(y_1) \tag{3}$$
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$^R_{\neg R}$ obtained.
- if we find less than N terms in $M(\mathcal{A}, q_2)$, but one is not in $[m_1]$. The case is the same as above.
- if we find less than N terms in $M(\mathcal{A}, q_2)$, all in $[m_1]$, it means that one of the applications of Lemma 2 was not successful, and hence that we have found all the terms of $M(\mathcal{A}, q_2)$. Following the proof of Lemma 2, the rule (1) is useless in this case and can be removed, and we can apply the induction hypothesis.

C Two-way tree automata with structural equality constraints are as expressive as standard tree automata

In this section, we shall prove Lemma 1. 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 automata (see e.g. [5]) and the proof for two-way automata with equality tests ([4]).

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

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

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

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

We assume w.l.o.g 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:

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

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

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

It is also satisfied by the universal binary relation and by the equality relation. That is why this generalizes corresponding results of [5, 4].

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

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

These clauses have a least Herbrand model. We write $[\![Q]\!]$ 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, $[\![Q]\!]$ 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

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

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

$$\begin{array}{ll} (s_2) & \underline{\mathsf{NE}_{Q_1}}, \underline{\mathsf{NE}_{Q_2}} \Rightarrow \mathsf{NE}_{Q_3} \\ (s_3) & \Rightarrow \underline{\mathsf{NE}_{Q}} \end{array}$$

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

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

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

$$\begin{array}{ll} (5.1) & Q_1(y_1), Q_2(y_2), \underline{Q_3(g(y_3, y_4))}, R_1(y_1, y_3), R_2(y_2, y_4) \Rightarrow Q_4(f(y_1, y_2)) \\ (5.2) & Q_1(y_1), Q_2(y_2), \underline{Q_3(a)}, S_1(y_1), S_2(y_2) \Rightarrow Q_4(f(y_1, y_2)) \\ (5.3) & \underline{Q_1(a)} \Rightarrow Q_2(b) \\ (5.4) & S_1(y_1), S_2(y_2), \underline{Q_1(f(y_1, y_2))} \Rightarrow Q_2(a) \end{array}$$

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

$$(6)\ \underline{\mathsf{NE}_{Q_3}} Q_1(y_1) \Rightarrow Q_2(y_1)$$

and eventually

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

(5.1) + (2) yields

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

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:

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

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

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

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

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

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

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

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

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

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 $\mathsf{NE}_{\mathcal{S}}$. 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):

$$\begin{array}{ll}(3) & Q_1(f(y_1,y_2)), \underline{\mathsf{NE}_{Q_2}} \Rightarrow Q_3(y_1)\\ (4) & Q_1(f(y_1,y_2)), \underline{\mathsf{NE}_{Q_2}} \Rightarrow Q_3(y_2)\\ (3b) & \underline{Q_1(f(y_1,y_2))} \Rightarrow Q_2(y_1)\\ (4b) & \underline{Q_1(f(y_1,y_2))} \Rightarrow Q_2(y_2)\end{array}$$

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

$$\begin{array}{ll}(\mathsf{P}_1) & P_1(x),\ldots,P_n(x),P'_1(y),\ldots,P'_m(y) \Rightarrow \underline{Q(f(x,y))}\\ (\mathsf{P}_2) & \Rightarrow \underline{P(a)}\\ (\mathsf{P}_3) & \underline{\mathsf{NE}_{\mathcal{S}}}, P_1(x),\ldots,P_n(x),P'_1(y),\ldots,P'_m(y) \Rightarrow \overline{Q(f(x,y))}\\ (\mathsf{P}_4) & \underline{\mathsf{NE}_{\mathcal{S}}} \Rightarrow Q(a)\end{array}$$

3. Intermediate clauses

$$\begin{array}{ll}(\mathsf{I}_1) & P_1(x),\ldots,P_n(x),P'_1(y),\ldots,P'_m(y),\underline{P''_1(f(x,y))},\ldots,\underline{P''_k(f(x,y))} \Rightarrow Q(f(x,y))\\ (\mathsf{I}_2) & \underline{P_1(a)},\ldots,\underline{P_n(a)} \Rightarrow Q(a)\\ (\mathsf{I}_3) & S_1(x_1),S_2(x_2),\overline{Q_1(a)} \Rightarrow Q_2(g(x_1,x_2))\\ (\mathsf{I}_4) & \underline{Q_1(a)} \Rightarrow Q_2(b)\end{array}$$

4. Alternating clauses

$$\begin{array}{ll}(\mathsf{A}_1) & \underline{\mathsf{NE}_{\mathcal{S}}}, P_1(x),\ldots,P_n(x) \Rightarrow Q(x)\\ (\mathsf{A}_2) & \underline{P_1(x)},\ldots,\underline{P_n(x)} \Rightarrow Q(x)\end{array}$$

In addition, we have clauses obtained by splitting:

5. Split clauses

$$\begin{array}{ll}(\mathsf{S}_1) & \underline{R_j(x,y)}, Q_i(y) \Rightarrow Q_i^{R_j}(x)\\ (\mathsf{S}_1b) & \underline{R_j(y,x)}, Q_i(y) \Rightarrow Q_i^{-R_j}(x)\\ (\mathsf{S}_2) & R_1(x_1,y_1),R_2(x_2,y_2),\underline{Q_i(f(y_1,y_2))} \Rightarrow Q_i^{\pm R_j}(g(x_1,x_2))\\ (\mathsf{S}_3) & S_1(x),S_2(y),\underline{Q_i(f(x,y))} \Rightarrow Q_i^{\pm R_j}(a)\\ (\mathsf{S}_4) & \underline{P_1(x)},\ldots,\underline{P_n(x)} \Rightarrow \mathsf{NE}_{\{P_1,\ldots,P_n\}}\\ (\mathsf{S}_5) & P_1(x),\ldots,P_n(x),P'_1(y),\ldots,P'_m(y),\underline{P''_1(f(x,y))},\ldots,\underline{P''_k(f(x,y))} \Rightarrow \mathsf{NE}_{\mathcal{S}}\end{array}$$

6. Propositional clauses

$$\begin{array}{ll}(\mathsf{E}_1) & \underline{\mathsf{NE}_{\mathcal{S}_1}}, \ldots, \underline{\mathsf{NE}_{\mathcal{S}_n}} \Rightarrow \mathsf{NE}_{\mathcal{S}} \\ (\mathsf{E}_2) & \Rightarrow \underline{\mathsf{NE}_{\mathcal{S}}} \\ (\mathsf{E}_3) & \underline{P_1(a)}, \ldots \underline{P_n(a)} \Rightarrow \mathsf{NE}_{\mathcal{S}}\end{array}$$

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 (A_1) and a split clause (S_4).
Intermediate + Push yields a Push clause or an intermediate clause
Alternating + Push yields an intermediate clause (I_1) or (I_2).
Split + R yields a split clause $(\mathsf{S})_2$ or (S_3) or an intermediate clause (I_3) or (I_4).
($\mathbf{S}_2$) **+ Push** yields clauses (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 (S_4).
($\mathbf{S}_4$)**+ Push** yields split clauses (S_5) or propositional clause (E_3).
($\mathbf{S}_5$)**+ Push** yields split clauses (S_5) or propositional clause (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 (P_1) and (P_2) (and automata clauses defining the S states). Let, for any state Q, $[\![Q]\!]_{\mathcal{A}}$ be the language accepted in state Q by $\mathcal{A}$. We claim that $[\![Q]\!] = [\![\mathcal{A}]\!]$.

To prove this, we first show (the proof is omitted here) that $\mathsf{NE}_{\{P_1,\ldots,P_n\}}$ is in $\mathcal{C}$ iff $[\![P_1]\!]_{\mathcal{A}} \cap \ldots \cap [\![P_n]\!]_{\mathcal{A}} \neq \emptyset$.

Then observe that $[\![Q]\!]$ is also the interpretation of Q in the least Herbrand model of $\mathcal{C}$: indeed, all computations yielding $\mathcal{C}$ are correct. Since $[\![Q]\!]_{\mathcal{A}} \subseteq [\![Q]\!]$ is trivial, we only have to prove the converse inclusion. For every $t \in [\![Q]\!]$ 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.

Claude $\mathbf{I}_1$ The last step of the proof is

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

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, $u_1 \in [\![P_i]\!]_{\mathcal{A}}$, $u_2 \in [\![P'_i]\!]_{\mathcal{A}}$ and $f(u_1, u_2) \in [\![P''_i]\!]_{\mathcal{A}}$. In particular, if we consider the last clause used in the proof of $P''_k(u)$:

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

belongs to $\mathcal{C}$. Then, overlapping this clause with the above clause I_1, the following clause belongs also to $\mathcal{C}$:

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

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

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

which contradicts the minimality of k.

Clause (A_1) The last step of the proof is

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

By hypothesis, the proofs of $P_i(u)$ only use automata clauses: $\forall i. u \in [\![P_i]\!]_{\mathcal{A}}$. Le the push rule

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

be the last clause used in the proof of $P(u)$. Overlapping this clause and the clause A_1 above, there is another clause in $\mathcal{C}$ yielding a proof of $P(u)$:

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

And we are back to the case of I_1.

Clause (3b)

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

By hypothesis $f(t, u) \in [\![Q_1]\!]_{\mathcal{A}}$. Hence there is a push clause

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

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

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

However, since $[\![P'_1]\!]_{\mathcal{A}} \cap \ldots \cap [\![P'_m]\!]_{\mathcal{A}} \neq \emptyset$, $\mathsf{NE}_{\{P'_1,\ldots,p'_m\}}$ is also in $\mathcal{C}$ and, by resolution again

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

is a clause of $\mathcal{C}$.

Then we are back to the case of A_1.

Clause (3) The last step of the proof is

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

Since $\mathsf{NE}_{Q_2} \in \mathcal{C}$ 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 (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 S_2, and is not provable by automata clauses only,

As before, we consider the overlap between S_2 and a push clause. We get

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

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):

$$\begin{aligned} \underline{R_1(x_1,y_1)}, P_i(y_1) &\Rightarrow P_i^{R_1}(x_1) \\ \underline{R_2(x_2,y_2)}, P'_i(y_2) &\Rightarrow {P'_i}^{R_2}(x_2) \\ P_1^{R_1}(x_1), \ldots, P_n^{R_1}(x_1), {P'_1}^{R_2}(x_2), \ldots, {P'_m}^{R_2}(x_2) &\Rightarrow \underline{Q_i^{R_j}(g(x_1,x_2))} \end{aligned}$$

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

$$\frac{\dfrac{R_1(u_1,v_1)\ P_1(v_1)}{P_1^{R_1}(u_1)} \quad \cdots \quad \dfrac{R_1(u_1,v_n)\ P_n(v_1)}{P_n^{R_1}(u_1)} \quad \dfrac{R_2(u_2,w_1)\ P'_2(w_1)}{{P'_1}^{R_2}(u_2)} \quad \cdots \quad \dfrac{R_2(u_2,w_m)\ P'_m(w_m)}{{P'_m}^{R_2}(u_2)}}{Q_i^{R_j}(g(u_1,u_2))}$$

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 $u_1 = h(u_{11},u_{12})$):

$$\dfrac{\dfrac{R_{11}(u_{11},v_{11})\ R_{12}(u_{12},v_{12})\ P_1(f(v_{11},v_{12}))}{P_1^{R_1}(u_1)} \quad \cdots \quad \dfrac{R_2(u_2,w_1)\ P_2'(w_1)}{{P_1'}^{R_2}(u_2)} \quad \cdots \quad \dfrac{R_2(u_2,w_m)\ P_m'(w_m)}{{P_m'}^{R_2}(u_2)}}{Q_i^{R_j}(g(u_1,u_2))}$$

By minimality of u, $u_1 \in [\![P_1^{R_1}]\!]_{\mathcal{A}}$. Similarly, for every i, $u_1 \in [\![P_i^{R_1}]\!]_{\mathcal{A}}$. $u_2 \in [\![{P_i'}^{R_2}]\!]_{\mathcal{A}}$ and it follows that $g(u_1,u_2) \in [\![Q_i^{R_j}]\!]_{\mathcal{A}}$.
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: $R_1(x,y) \wedge R(y,z) \models\!\models R_1'(x,y) \wedge R'(x,z)$, hence

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

Hence we need two split clauses instead of one:

$$R_1'(x,y) \Rightarrow S_\top^{R_1'}(x)$$
$$R'(x,y), Q(y) \Rightarrow Q^{R'}(x)$$

And $R_1(x_1,y_1), Q^R(y_1)$ is replaced with $S_\top^{R_1'}(x_1), Q^{R'}(x_1)$. 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: $u \in [\![P]\!]_{\mathcal{A}}$. This proves that $[\![P]\!] = [\![P]\!]_{\mathcal{A}}$.

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.