Propositional Tree Automata*

Joe Hendrix[1], Hitoshi Ohsaki[2], and Mahesh Viswanathan[1]

[1] Department of Computer Science, University of Illinois at Urbana-Champaign
Thomas M. Siebel Center for Computer Science, Urbana, IL 61801-2302, USA
{jhendrix,vmahesh}@uiuc.edu

[2] National Institute of Advanced Industrial Science and Technology
Nakoji 3–11–46, Amagasaki, Hyogo 661-0974, Japan
ohsaki@ni.aist.go.jp

Abstract. In the paper, we introduce a new tree automata framework, called *propositional tree automata*, capturing the class of tree languages that are closed under an equational theory and Boolean operations. This framework originates in work on developing a sufficient completeness checker for specifications with rewriting modulo an equational theory. Propositional tree automata recognize regular equational tree languages. However, unlike regular equational tree automata, the class of propositional tree automata is closed under Boolean operations. This extra expressiveness does not affect the decidability of the membership problem. This paper also analyzes in detail the emptiness problem for propositional tree automata with associative theories. Though undecidable in general, we present a semi-algorithm for checking emptiness based on machine learning that we have found useful in practice.

1 Introduction

Tree automata techniques have been commonly used in checking consistency of tree structures. Typical examples include checking sufficient completeness of algebraic specifications [6] and the consistency of semi-structured documents [16]. These applications benefit from the good closure properties and positive decidability results for tree automata. Recently, there are more advanced applications including protocol verification [2, 11], type inference [8, 10], querying in databases [26, 27] and theorem proving [18].

One limitation of tree automata in these applications is that the regularity of languages is not preserved when closed with respect to congruences. In other words, when some algebraic laws such as associativity and commutativity are taken into account, the congruence closure of a regular tree language may no longer be regular. In applications, this lack of closure has required users of tree automata techniques to use complicated and specialized ways of encoding protocols [5]. Many extensions of tree automata have been suggested to address this problem, including multitree automata by Lugiez [19], two-way alternating tree automata by Verma [28], and equational tree automata by Ohsaki [24].

* Research supported by ONR Grant N00014-02-1-0715, NSF CAREER CCF-0448178, and NSF CCF-0429639

Equational tree automata are a natural mathematical extension of tree automata that recognize tree languages modulo an equational theory. Equational tree automata enjoy several nice properties. In particular, they are weakest extensions to tree automata that are closed under congruences. More precisely, when the equational theory is induced by only linear equations (i.e equations whose left- and right-hand sides are linear terms), such automata recognize exactly the congruence closure of regular languages (Lemma 2, [24]).

Checking properties of tree structures, however, often requires that the modeling language be closed under boolean operations and have efficient algorithms to check emptiness and inclusion. For example, when checking sufficient completeness, the main task is to check if the language of terms with defined functions is contained in the language of reducible terms. Thus, a sufficient completeness checker relies on a modeling language for trees for which checking inclusion is decidable. Since inclusion tests are most often implemented by complementation, intersection and a test for emptiness, these properties also are relevant for this problem. It is known that for *regular* equational tree automata with only associativity equations, the inclusion problem is undecidable. Moreover, this class of languages is not closed under intersection and complementation [23].

Motivated by this inadequacy in equational tree automata, Hendrix *et al.* proposed in [13] a further extension of tree automata, called *propositional tree automata*. These automata define a class of languages that is immediately closed under all the boolean operations via a straightforward, effective procedure for each operation. More importantly, they are the mathematically minimal extension in that the class of propositional tree automata accept the Boolean closure of languages recognizable by equational tree automata. The conservativeness of our extension leads to another desirable property: if the equational tree automata membership problem is decidable for a theory $\mathcal{E}$, then the membership problem for the propositional tree automata with $\mathcal{E}$ is decidable as well.

In [13], Hendrix *et al.* showed that the sufficient completeness problem for unconditional and left-linear membership rewrite systems modulo an equational theory can be reduced to the emptiness problem of propositional tree automata. Hence, one of the problems we investigate here is the emptiness problem modulo A- and AC-theories. Based on results for equational tree automata, we know that the problem is undecidable for propositional automata modulo A-theories. In this paper, we present a machine learning based semi-decision procedure, that is also a complete decision procedure under certain regularity conditions. We have found this algorithm effective in practice. Our algorithm has been implemented in a tree automata software library, called *CETA* [14], that can check the emptiness of propositional tree automata modulo associativity, commutativity, and identity. CETA is currently used for a next-generation sufficient completeness checker for Maude, and has already found a subtle bug in the built-in Maude specifications that can not be verified using the current checker.

This paper is organized as follows. In the next section, we define propositional tree automata. We show how this framework is closed under Boolean operations, and also investigate the recognition power relative to equational tree

automata. In Section 3, we consider the membership decision problem, and analyze the complexity results with the comparison to equational tree automata. In Sections 4 and 5, we explain our approach to the emptiness problem in detail. In Section 6, we show how our approach can be improved using ideas from machine learning. Finally, in Section 7, we conclude the paper by addressing the current software development project.

2 Preliminaries

We assume the reader is familiar with equational logic [6] and tree automata [7]. We use basic notations of rewriting from [4]. An *equational theory* is a pair $\mathcal{E} = (F, E)$ in which F is a finite set of function symbols, each with an associated *arity*, and E is a set of equations over the function symbols in F.

In the paper we are mainly interested in associative and/or commutative theory (A∪C-theories for short), that is equational theories whose equations in E are associativity and/or commutativity axioms for some of the binary function symbols. Given a binary function symbol $f \in F$, $f(f(x,y),z) = f(x,f(y,z))$ is an associativity (A) axiom, and $f(x,y) = f(y,x)$ is a commutativity (C) axiom. We use F_{A} to denote the symbols in F with an associativity axiom in E, and F_{C} to be the symbols with a commutativity axiom. Since commutativity alone does not essentially affect the expressive power of the languages (Theorem 3, [24]), we assume that each commutative symbol is associative, i.e. $F_{\mathrm{C}} \subseteq F_{\mathrm{A}}$. Furthermore we write AC to denote the set E consisting of *both* A *and* C axioms from $F_{\mathrm{A}} \cap F_{\mathrm{C}}$.

A *propositional tree automaton* (PTA) $\mathcal{A}$ is a tuple $(\mathcal{E}, Q, \phi, \Delta)$, consisting of the equational theory $\mathcal{E} = (F, E)$, a finite set Q of states disjoint from the symbols in F (i.e. $F \cap Q = \varnothing$), a propositional formula ϕ over Q, and a finite set Δ of transition rules whose shapes are in one of the following forms:

$$\begin{array}{cc} \text{(Regular)} & \text{(Monotone)} \\ f(p_1,\ldots,p_n) \to q & f(p_1,\ldots,p_n) \to f(q_1,\ldots,q_n) \end{array}$$

for some $f \in F$ with $\mathsf{arity}(f) = n$ and $p_1,\ldots,p_n,q,q_1,\ldots,q_n \in Q$. If a PTA only has regular rules, we say the PTA is *regular*; otherwise, it is *monotone*.

A move relation of $\mathcal{A} = (\mathcal{E}, Q, \phi, \Delta)$ is a rewrite relation over the set $\mathcal{T}(F \cup Q)$ of terms with respect to $\to_\Delta$ modulo $=_\mathcal{E}$, i.e. $s \to_\mathcal{A} t$ if there is a transition rule $l \to r \in \Delta$ and a context $C \in \mathcal{C}(F \cup Q)$ such that $s =_\mathcal{E} C[l]$ and $t =_\mathcal{E} C[\alpha]$. The reflexive-transitive closure of $\to_\mathcal{A}$ is denoted by $\to^*_\mathcal{A}$.

A term t is *accepted* by $\mathcal{A}$ if $t \in \mathcal{T}(F)$ and the complete set of states reachable from t, $\mathsf{reach}_\mathcal{A}(t) = \{\, \alpha \mid \exists \alpha \in Q\colon t \to^*_\mathcal{A} \alpha \,\}$, is a model of ϕ. Boolean formulas are evaluated using their standard interpretations:

$$P \models \alpha \;\text{ if } \alpha \in P,\;\; P \models \phi_1 \vee \phi_2 \;\text{ if } P \models \phi_1 \;\text{ or }\; P \models \phi_2,\;\; P \models \neg\,\phi \;\text{ if not}(\, S \models \phi\,)$$

As an example, we consider the PTA $\mathcal{A}$ with the propositional formula $\phi = \alpha \wedge \neg\beta$ and the transition rules

$$\mathsf{a} \to \alpha \qquad \mathsf{b} \to \beta \qquad \mathsf{f}(\alpha) \to \alpha \qquad \mathsf{f}(\beta) \to \beta \qquad \mathsf{f}(\alpha) \to \gamma \qquad \mathsf{f}(\beta) \to \gamma.$$

Then a is accepted by $\mathcal{A}$, because $\mathsf{reach}_\mathcal{A}(\mathsf{a}) = \{\,\alpha\,\}$ and $\{\,\alpha\,\} \models \alpha \wedge \neg\beta$. Similarly, $\mathsf{f}(\mathsf{a})$ is accepted as $\mathsf{reach}_\mathcal{A}(\mathsf{f}(\mathsf{a})) = \{\,\alpha,\gamma\,\}$ and $\{\,\alpha,\gamma\,\} \models \alpha \wedge \neg\beta$. However,

b and $\mathsf{f}(\mathsf{b})$ are not accepted, because $\mathsf{reach}_{\mathcal{A}}(\mathsf{b}) = \{\beta\}$ and $\{\beta\} \not\models \alpha \wedge \neg\beta$, and $\mathsf{reach}_{\mathcal{A}}(\mathsf{f}(\mathsf{b})) = \{\beta, \gamma\}$ and $\{\beta, \gamma\} \not\models \alpha \wedge \neg\beta$. Intuitively, the formula $\alpha \wedge \neg\beta$ means that $\mathcal{A}$ accepts any term reachable to the state α but unreachable to β.

Propositional tree automata are closed under Boolean operations: given $\mathcal{A} = (\mathcal{E}, Q_1, \phi_1, \Delta_1)$ and $\mathcal{B} = (\mathcal{E}, Q_2, \phi_2, \Delta_2)$, then by assuming $Q_1 \cap Q_2 = \varnothing$, the intersection $\mathcal{L}(\mathcal{A}) \cap \mathcal{L}(\mathcal{B})$ is accepted by the PTA $(\mathcal{E}, Q_1 \cup Q_2, \phi_1 \wedge \phi_2, \Delta_1 \cup \Delta_2)$. The complement of $\mathcal{L}(\mathcal{A})$ is accepted by $\mathcal{A}' = (\mathcal{E}, Q_1, \neg\phi_1, \Delta_1)$, where the formula ϕ_1 of $\mathcal{A}$ is replaced by $\neg\phi_1$. Therefore we have the following property for propositional tree automata.

Lemma 1. *The class of propositional tree automata is effectively closed under Boolean operations.* □

In the standard tree automata framework, the intersection of two tree automata may have the product of states, which is $|Q_1| \times |Q_2|$ state symbols, while the intersection of PTA $\mathcal{A}$ and $\mathcal{B}$ needs $|Q_1| + |Q_2|$ state symbols. In complimenting the PTA $\mathcal{A}$, the set of states is unchanged, so the number of state symbols is $|Q_1|$. But constructing the complement of a tree automaton, may require an exponential number of state symbols relative to the original.

It is also an easy lemma to show that the class of languages accepted by propositional tree automata under a certain equational theory is the smallest class of languages containing languages accepted by standard equational tree automata with the same equational theory and closed with respect to Boolean operations over the languages.

Lemma 2. *The class of tree languages accepted by PTA with an equational theory $\mathcal{E}$ corresponds precisely to the Boolean closure of tree languages accepted by equational tree automata sharing the equational theory $\mathcal{E}$.* □

One can observe that, given a term $t \in \mathcal{T}(F)$ and a propositional tree automaton $\mathcal{A}$, when $t \rightarrow_{\mathcal{A}}^{*} \alpha$ is decidable for any state α of $\mathcal{A}$, $\mathsf{reach}_{\mathcal{A}}(t)$ is effectively computable. This leads to the observation:

Lemma 3. *The membership problem for equational tree automata under an equational theory $\mathcal{E}$ is decidable if and only if the membership problem for propositional tree automata with $\mathcal{E}$ is decidable.* □

3 Decidability Results

As we showed in the previous section, the decidability of the membership problem of propositional tree automata depends upon that of equational tree automata with the usual definition of acceptance in terms of final states. From previous work [21, 23], we have the complexity results (in the next table) for regular and monotone cases with AC- or A-theory:

	regular A-TA	regular AC-TA	monotone A-TA	monotone AC-TA
complexity of membership	P-time	NP-complete	PSPACE-compl.	PSPACE-compl.

As an obvious observation, the membership problem for *propositional* regular AC-tree automata (abbreviated by MEM-PROP-REG-ACTA) seems harder than the problem for regular AC-tree automata. Here a propositional regular AC-tree automaton is a regular PTA over AC-theory, and a regular AC-tree automaton (regular AC-TA for short) corresponds to a regular PTA over AC-theory with a disjunction ϕ over atomic states as its propositional formula, i.e. $\phi = \alpha_1 \vee \cdots \vee \alpha_n$ for some $\alpha_1, \ldots, \alpha_n \in Q$.

As the AC-TA membership problem is NP-complete and the AC-TA non-membership problem can be converted in linear-time to the PTA membership problem, the PTA membership problem cannot be in NP unless NP equals co-NP. We can show that MEM-PROP-REG-ACTA is in a higher complexity class.

Lemma 4. MEM-PROP-REG-ACTA *is in* Δ_2^P. □

In the following, we write $A \leqslant_T^P B$ if there is an algorithm M running polynomial-time for a problem A which can ask, during its computation, some membership questions about B, where each query for B is answered in a unit time. The relation $A \leqslant_m^P B$ is polynomial-time *many-to-one* reducibility, and it is defined as follows: $A \leqslant_m^P B$ if there exists a polynomial-time function $f\colon \Sigma^* \to \Gamma^*$ such that for each $x \in \Sigma^*$, $x \in A$ if and only if $f(x) \in B$.

Proof of Lemma 4. Let $\mathcal{A} = (\mathcal{E}, Q, \phi, \Delta)$ with $\mathcal{E} = (F, \mathrm{AC})$. We define the regular AC-tree automaton $\mathcal{B}_\mathcal{A}$ associated to $\mathcal{A}$. By assuming $\langle\,,\rangle$ is a fresh binary symbol, we let $\mathcal{B}_\mathcal{A} = (\mathcal{E}', P, \mathsf{p}_{\mathrm{acc}}, \Delta_\mathcal{A})$ where

$$\begin{aligned}
\mathcal{E}' &= (F \cup Q \cup \{\langle\,,\rangle\}, \mathrm{AC}) \\
P &= \{\mathsf{p}_\alpha,\ \mathsf{q}_\alpha \mid \alpha \in Q\} \cup \{\mathsf{p}_{\mathrm{acc}}\} \\
\Delta_\mathcal{A} &= \{\alpha \to \mathsf{q}_\alpha \mid \alpha \in Q\} \\
&\quad \cup \{\langle \mathsf{p}_\alpha, \mathsf{q}_\alpha\rangle \to \mathsf{p}_{\mathrm{acc}} \mid \alpha \in Q\} \\
&\quad \cup \{f(\mathsf{p}_\alpha, \mathsf{p}_\beta) \to \mathsf{p}_\gamma \mid f(\alpha, \beta) \to \gamma \in \Delta\}.
\end{aligned}$$

By construction, it is clear that for each $t \in \mathcal{T}(F)$ and $\alpha \in Q$, $t \to_\mathcal{A}^* q$ if and only if $\langle t, q\rangle \to_{\mathcal{B}_\mathcal{A}}^* \mathsf{p}_{\mathrm{acc}}$. One should note that $\mathcal{B}_\mathcal{A}$ can be constructed in quadratic-time to the size of $\mathcal{A}$, and the the membership problem for regular AC-tree automata (abbreviated by MEM-REG-ACTA) is NP-complete.

For the next step, we take the set $S = \{\alpha \in Q \mid \alpha \text{ appears in } \phi\}$. The computation of S can be deterministically done in the size of ϕ, denoted by $|\phi|$, which is the number of occurrences of Boolean variables and Boolean connectives in ϕ. Then, for every $\alpha \in S$ (e.g. in the lexicographic order), we test by using the oracle $\mathcal{L}(\mathcal{B}_\mathcal{A})$, whether $\langle t, \alpha\rangle \in \mathcal{L}(\mathcal{B}_\mathcal{A})$. If $\langle t, \alpha\rangle \in \mathcal{L}(\mathcal{B}_\mathcal{A})$ is true, α is assigned to 1; otherwise, α is 0. By letting this Boolean assignment to be the mapping $\tau\colon S \to \{1, 0\}$, it is easy to see $\tau(\phi) = 1$ if and only if $\mathsf{reach}_\mathcal{A}(t) \models \phi$. The output value of the Boolean circuit is computable in polynomial-time relative to $|\phi|$ [9].

The above algorithm runs totally in polynomial-time with respect to the size of $\mathcal{A}$. Therefore the deterministic algorithm with an oracle set in NP solves the original membership problem in polynomially bounded time.

As a corollary of the above proof, MEM-PROP-REG-ACTA is

– $\leqslant_m^P$-hard for NP (i.e. NP-hard),

– $\leqslant_m^P$-hard for coNP (i.e. coNP-hard).

One can observe that the problem determining, given a term t and a regular AC-tree automaton $\mathcal{A}$, whether $\mathcal{A}$ *does not* accept t is coNP-complete (abbreviated by INACCEPT-REG-ACTA). Because, if $L \in$ coNP then $(L)^c \in$ NP, and thus there exists a polynomial-time function f from $(L)^c$ to MEM-REG-ACTA such that $x \in (L)^c$ if and only if $f(x)$ is accepted by a regular AC-tree automaton $(F, Q, \alpha_1 \vee \cdots \vee \alpha_n, \Delta, \mathrm{AC})$ with $\alpha_1, \ldots, \alpha_n \in Q$. Then the reduction from L to MEM-PROP-REG-ACTA can be done by taking the propositional regular AC-tree automaton to be $(F, Q, \neg(\alpha_1) \wedge \cdots \wedge \neg(\alpha_n), \Delta, \mathrm{AC})$.

Moreover, MEM-PROP-REG-ACTA is $\leqslant_T^P$-equivalent to MEM-REG-ACTA, because $\leqslant_m^P$ is subsumed in $\leqslant_T^P$ and $\leqslant_T^P$ is transitive. Then, $\forall A \in \Delta_2^P : A \leqslant_T^P$ MEM-REG-ACTA, and thus, MEM-PROP-REG-ACTA $\leqslant_T^P$ MEM-REG-ACTA.

In case of monotone PTA over AC-theory, using the same construction as in the proof of Lemma 4, we can show that: the membership problem for monotone PTA over AC-theory is in $\mathrm{P}^{\mathrm{PSPACE}}$ (= PSPACE). Then, using the fact that the membership problem for monotone PTA over AC-theory is PSPACE-complete [21], we can obtain even a stronger result: the membership problem for monotone PTA over AC-theory (indicated by monotone AC-PTA in the table) is PSPACE-complete.

The previous proof technique can also be applied to A case. Therefore we obtain the following table of complexity results for sub-classes of propositional tree automata:

	regular A-PTA	regular AC-PTA	monotone A-PTA	monotone AC-PTA
complexity of membership	P-time	Δ_2^P	PSPACE-compl.	PSPACE-compl.

4 Emptiness Testing

We now turn out attention to the emptiness problem for PTA — given a PTA $\mathcal{A}$, does $\mathcal{L}(\mathcal{A}) = \varnothing$? This problem is computationally quite hard. Even in the free case, testing emptiness of a PTA is EXPTIME-complete. The tree automata universality problem, i.e. given a tree automaton $\mathcal{A}$ over a signature F, does $\mathcal{L}(\mathcal{A}) = \mathcal{T}(F)$?, is EXPTIME-complete (Theorem 14, [7]). This problem can be converted in linear time into the PTA emptiness problem of $(\mathcal{L}(\mathcal{A}))^c$.

In AC case, equational tree automata are known to be closed under Boolean operations [26], and the emptiness problem is decidable [23]. It follows that the class of regular PTA over AC-theory have a decidable emptiness problem. In contrast to the above, in A case (without commutativity axioms), the emptiness problem is undecidable:

Theorem 1. *The problem of checking whether $\mathcal{L}(\mathcal{A}) = \varnothing$ for regular PTA with a single associativity axiom is undecidable.*

Proof. Given a regular equational tree automaton $\mathcal{B}$ with a single associative symbol, it was shown in [24] to be undecidable whether $\mathcal{L}(\mathcal{B}) = \mathcal{T}(F)$. This problem is equivalent to checking $(\mathcal{L}(\mathcal{B}))^{\mathsf{c}} = \varnothing$. By Lemma 2, the language $(\mathcal{L}(\mathcal{B}))^{\mathsf{c}}$ is recognizable by a PTA with a single associative symbol. □

Despite the lack of decidability, we nevertheless are interested in developing semi-decision algorithms that work well in practice. This is motivated by the study about the sufficient completeness checking of order-sorted equational specifications, where we have found equational tree automata techniques to be quite useful [13]. In applications, thus far we have mainly been interested in regular PTA, so we will restrict our attention to regular PTA for the remainder of this section.

Our algorithm for checking emptiness computes the set of states reachable from terms. The idea of this algorithm is similar to the subset construction used in complementing regular tree automata in [7], but with extensions to handle associative and commutative symbols. Though having no guarantee to terminate for all cases, the algorithm finds an accepting term if a language accepted by an input PTA is non-empty, and it proves the emptiness if the accepting language is empty and the PTA satisfies certain regularity conditions.

Let $\equiv_{\mathcal{A}}$ be the equivalence relation over terms where $s \equiv_{\mathcal{A}} t$ iff. $\mathsf{reach}_{\mathcal{A}}(s) = \mathsf{reach}_{\mathcal{A}}(t)$. For tree automata, the correctness of subset construction typically relies on the fact that $\equiv_{\mathcal{A}}$ is a congruence with respect to contexts. i.e. $s \equiv_{\mathcal{A}} t$ implies $C[s] \equiv_{\mathcal{A}} C[t]$ for all contexts C. However, this fact *does not* hold in the case when the root of s or t is an A symbol f and the context C has s or t immediately within a term labeled by f. Due to this complication, our subset construction algorithm for A and AC symbols maintains additional information.

We first define the information our subset construction algorithm for the A and AC case will eventually compute.

Definition 1. *Given a PTA $\mathcal{A} = (\mathcal{E}, Q, \phi, \Delta)$ over the theory $\mathcal{E} = (F, E)$, let $\mathsf{derive}(\mathcal{A}) \subseteq \mathcal{P}(Q) \times F$ be the set $\mathsf{derive}(\mathcal{A}) = \{\, (\mathsf{reach}_{\mathcal{A}}(t), \mathrm{root}(t)) \mid t \in T_F \,\}$.*

One should remark that $\mathsf{derive}(\mathcal{A})$ is finite, however it is not always computable. Observe that $\mathcal{L}(\mathcal{A}) \neq \varnothing$ if and only if there is a pair $(P, f) \in \mathsf{derive}(\mathcal{A})$ such that $P \models \phi$. The undecidability of the emptiness problem of $\mathcal{L}(\mathcal{A})$ thus implies the membership question $(P, f) \in \mathsf{derive}(\mathcal{A})$ is not decidable either.

For the remainder of this section, let $\mathcal{A}$ be a PTA with an A∪C-theory. In this case, we can obtain $\mathsf{derive}(\mathcal{A})$ by iterative computation starting from the empty set $\mathsf{d}_{\mathcal{A}}(0) \triangleq \varnothing$. We then expand $\mathsf{d}_{\mathcal{A}}(0)$ to $\mathsf{d}_{\mathcal{A}}(1), \mathsf{d}_{\mathcal{A}}(2), \ldots$ in the inference rules (defined later) until completion. The mapping $\mathsf{d}_{\mathcal{A}}$ is simplified to d if $\mathcal{A}$ is obvious in the context.

Before describing the inference rules, we must give a few more definitions. We first extend $\mathsf{reach}_{\mathcal{A}}$ to allow sets of states $P_i \subseteq Q$ as constants appearing in terms. Precisely, the reachable states $\mathsf{reach}_{\mathcal{A}}(f(P_1, \ldots, P_n))$ for a *term with sets as constants* is the union of the reachable states for each term in $\mathcal{T}(F \cup Q)$ formed by choosing an element in each state, i.e.

$$\mathsf{reach}_{\mathcal{A}}(f(P_1, \ldots, P_n)) = \{\, \beta \in Q \mid (\exists\, \alpha_i \in P_i : 1 \leq i \leq n)\ f(\alpha_1, \ldots, \alpha_n) \rightarrow_{\mathcal{A}}^{*} \beta \,\}\,.$$

Moreover, for associative symbols $f \in (F_{\mathrm{A}} - F_{\mathrm{C}})$, we define a context-free grammar $G_{f,\mathsf{d}(i)}$ associated to d, to capture the variadic nature of associative symbols.

Definition 2. *Given a PTA* $\mathcal{A} = (\mathcal{E}, Q, \phi, \Delta)$ *with* $f \in F_{\mathrm{A}}$ *and set* $\mathsf{d}(i)$*, we define the* flattened grammar for f, $G_{f,\mathsf{d}(i)}(\_) = (\Sigma_{f,\mathsf{d}(i)}, Q, \_, R)$*, where*

- $\Sigma_{f,\mathsf{d}(i)} = \{\, P \mid \exists (P, g) \in \mathsf{d}(i) \colon g \neq f \,\}$,
- $R = \{\, \gamma := \alpha\beta \mid f(\alpha, \beta) \to \gamma \in \Delta \,\} \cup \{\, \gamma := P \mid P \in \Sigma_{f,D_i} \wedge \gamma \in P \,\}$.

In the paper, we write $\mathcal{L}(G(\alpha))$ to denote a language generated from α if G is (a mapping to) a grammar with a non-terminal symbol α. The *Parikh image* of the language $\mathcal{L}(G(\alpha))$ [25] is denoted by $\mathcal{S}(G(\alpha))$. Namely, $\mathcal{S}(G(\alpha)) = \{\, \#(w) \mid w \in \mathcal{L}(G(\alpha)) \,\}$, where $\# : \Sigma^* \to \mathbb{N}^{|\Sigma|}$ maps each string in Σ^* to the vector counting the number of occurrences of each terminal symbol. For a subset P ($\subseteq Q$) of non-terminals, let $\mathcal{L}(G(P))$ denotes the strings appearing only in the language generated from non-terminals $\alpha \in P$. The mapping for the Parikh image is denoted by $\mathcal{S}(G(P))$:

$$\mathcal{L}(G(P)) \;=\; \bigcap_{\alpha \in P} \mathcal{L}(G(\alpha)) \;-\; \bigcup_{\beta \in (Q-P)} \mathcal{L}(G(\beta))$$

$$\mathcal{S}(G(P)) \;=\; \bigcap_{\alpha \in P} \mathcal{S}(G(\alpha)) \;-\; \bigcup_{\beta \in (Q-P)} \mathcal{S}(G(\beta)).$$

As context-free grammars are not closed under intersection and complementation, it is essential to denote $\mathcal{L}(G(P))$. Besides, checking the emptiness of $\mathcal{L}(G(P))$ is undecidable. On the other hand, $\mathcal{S}(G(P))$ is a semi-linear set [25], because semi-linear sets are closed under Boolean operations, and moreover, the emptiness problem is decidable.

Now let us define the rules for computing a set $\mathsf{d}(i+1)$ given $\mathsf{d}(i)$ and starting with $\mathsf{d}(0) = \varnothing$:

$$(1)\quad f \notin F_{\mathrm{A}} \cup F_{\mathrm{C}} : \quad \frac{(P_1, f_1), \ldots, (P_n, f_n) \in \mathsf{d}(i)}{\mathsf{d}(i+1) = \mathsf{d}(i) \uplus \{\, (\, \mathsf{reach}_{\mathcal{A}}(f(P_1, \ldots, P_n)), f \,) \,\}}$$

$$(2)\quad f \in F_{\mathrm{A}} - F_{\mathrm{C}} : \quad \frac{P \subseteq Q \qquad \Sigma^{2+}_{f,\mathsf{d}(i)} \cap \mathcal{L}(G_{f,\mathsf{d}(i)}(P)) \neq \varnothing}{\mathsf{d}(i+1) = \mathsf{d}(i) \uplus \{\, (\, P, f \,) \,\}}$$

$$(3)\quad f \in F_{\mathrm{A}} \cap F_{\mathrm{C}} : \quad \frac{P \subseteq Q \qquad \mathbb{N}^{>\mathbf{1}} \cap \mathcal{S}(G_{f,\mathsf{d}(i)}(P)) \neq \varnothing}{\mathsf{d}(i+1) = \mathsf{d}(i) \uplus \{\, (\, P, f \,) \,\}}$$

In the first rule, we non-deterministically chose elements $(P_1, f_1), \ldots, (P_n, f_n)$ from $\mathsf{d}(i)$. These elements need not be distinct. In the second and third rules, we write $\Sigma^{2+}_{f,\mathsf{d}(i)}$ for the strings over $\Sigma_{f,\mathsf{d}(i)}$ containing at least two letters, and $\mathbb{N}^{>\mathbf{1}}$ for vectors over natural numbers whose elements sum up to at least 2. We use the disjoint union operator $\uplus$ to denote that the newly added elements must be *distinct* from the other elements in $\mathsf{d}(i)$.

It is relatively straightforward to show that by starting with $\mathsf{d}(0)$ and applying the rules for each operator until completion, we eventually have $\mathsf{derive}(\mathcal{A})$.

Theorem 2. *Let $\mathcal{A} = (\mathcal{E}, Q, \phi, \Delta)$ be a PTA with $\mathcal{E} = (F, E)$ containing only associativity and commutativity axioms (*A∪C*-theory). Every chain $\mathsf{d}(0), \mathsf{d}(1), \ldots$ obtained by applying the rules (1)–(3) until completion satisfies the following properties:*

- *the length k of the chain is $|\mathsf{derive}(\mathcal{A})|$, and*
- $\mathsf{d}(k) = \mathsf{derive}(\mathcal{A})$.

Proof. Appendix A. □

The undecidability of PTA with associative symbols in the regular case crops up in testing the emptiness of $\Sigma^{2+}_{f,\mathsf{d}(i)} \cap \mathcal{L}(G_{f,\mathsf{d}(i)}(P))$. The focus of the next section concerns how to solve this emptiness constraint. It is worth observing that this subset construction based approach can be generalized for the monotone case as well, but in this case, the grammar $G_{f,\mathsf{d}(i)}$ has to contain the additional rules of the form $\alpha\beta := \gamma\delta$ to account for monotone rules of the form $f(\gamma, \delta) \to f(\alpha, \beta)$.

5 Solving Language Equations for Associativity

Since at present the emptiness testing with monotone rules for associative symbols is beyond the goal of our project, we have developed an approach that is likely to work well in practice for the *regular* case with associative symbols. Our approach rests of an interactive semi-algorithm for each associative symbol $f \in F_{\mathrm{A}}$ which has access to the mapping $\mathsf{d}(i)$ as it is being generated and performs two actions simultaneously: (1) recursively enumerates pairs (P, f) not in $\mathsf{d}(i)$ for which $\Sigma^{2+}_{f,\mathsf{d}(i)} \cap \mathcal{L}(G_{f,\mathsf{d}(i)}(P))$ is non-empty; and (2) applies machine learning techniques to attempt construction of a family $\{\, \mathcal{M}_\alpha \,\}_{\alpha \in Q}$ of deterministic finite automata for which $\mathcal{L}(\mathcal{M}_\alpha) = \mathcal{L}(G_{f,\mathsf{derive}(\mathcal{A})}(\alpha))$ for all $\alpha \in Q$. If the first action succeeds, the semi-algorithm constructs the next $\mathsf{d}(i+1)$ from $\mathsf{d}(i)$. If the second action succeeds, we can decide for each subset of P states, the condition $\Sigma^{2+}_{f,\mathsf{d}(i)} \cap \mathcal{L}(G_{f,\mathsf{derive}(\mathcal{A})}(P)) = \varnothing$ in the rule (2). We then can either obtain $\mathsf{d}(i+1)$ or prove that the conditional rule for f can no longer be applied.

A naïve approach to the first action is quite simple. We recursively enumerate the strings in $\Sigma^{2+}_{f,\mathsf{d}(i)}$ in order of increasing length to form the infinite sequence $w_1, w_2, \ldots$, and parse each string w_i to get the complete set of states $P_i = \{\, \alpha \in Q \mid w \in \mathcal{L}(G_{f,\mathsf{d}(i)}(\alpha)) \,\}$. If $(P_i, f) \notin \mathsf{d}(i)$, then let $\mathsf{d}(i+1) = \{\, (P_i, f) \,\} \cup \mathsf{d}(i)$. Handling the second action is more complicated. First, observe that we can enumerate the set of finite automata in order of increasing length. Because recursively enumerable sets are closed under finite products, we can even enumerate finite families of automata $\{\, \mathcal{M}_\alpha \,\}_{\alpha \in Q}$. The difficult part then lies in checking whether $\mathcal{L}(\mathcal{M}_\alpha) = \mathcal{L}(G_{f,\mathsf{d}(i)}(\alpha))$ for all $\alpha \in Q$. It is well known that given a *single* finite automaton $\mathcal{M}$ and context-free grammar G, it is *undecidable* whether $\mathcal{L}(\mathcal{M}) = \mathcal{L}(G)$ (Theorem 8.12 (3), [15]). However, this result is just for a single automaton, and does not imply the undecidability of our problem. In fact, given a context-free grammar $G = (\Sigma, Q, \alpha_0, R)$ in Chomsky normal form, and a family of automata $\{\, \mathcal{M}_\alpha \,\}_{\alpha \in Q}$, the question whether $\mathcal{L}(\mathcal{M}_\alpha) = \mathcal{L}(G(\alpha))$ for all $\alpha \in Q$ is decidable.

The decidability of this problem is a direct consequence of Theorem 2.3 in [3]. Before explaining that result, however, it is necessary to shift our perspective of context-free grammars from viewing them as collections of *production rules* to viewing them as systems of *language equations*.

Definition 3. *Let $G = (\Sigma, Q, \alpha_0, R)$ be a context-free grammar. The* system of equations *generated by G is the family of equations $\{ \alpha = P_\alpha \}_{\alpha \in Q}$ in which for each non-terminal $\alpha \in Q$, P_α is the formula $P_\alpha = w_1 \mid \cdots \mid w_n$ where $\alpha = w_1, \ldots, \alpha = w_n$ are the production rules in R whose left-hand side equals α.*

Given a system of equations with non-terminals Q and terminals Σ, a *substitution* is a mapping $\theta : Q \to \mathcal{P}(\Sigma^*)$ associating each state $\alpha \in Q$ to a language $\theta(\alpha) \subseteq \Sigma^*$. A substitution θ can be *applied* to a language formula P, yielding a language $P\theta \subseteq \Sigma^*$ which is defined using the axioms:

$$P\theta = \begin{cases} \{ a \} & \text{if } P = a \text{ for some } a \in \Sigma, \\ \theta(\alpha) & \text{if } P = \alpha \text{ for some } \alpha \in Q, \\ S\theta \cup T\theta & \text{if } P = (S \mid T), \\ \{ st \mid s \in S\theta \ \wedge \ t \in T\theta \} & \text{if } P = (S \,.\, T). \end{cases}$$

We may assume associativity of $\mid$ and $.$ in the above definition. Here $S \,.\, T$ denotes the concatenation of S and T. A substitution $\theta : Q \to \mathcal{P}(\Sigma^*)$ is a *solution* to the system of equations $\{ \alpha = P_\alpha \}_{\alpha \in Q}$ if and only if $\theta(\alpha) = P_\alpha\theta$ for all $\alpha \in Q$. It is known that each system of equations generated by G has a *least solution*, namely $\theta_{\mathcal{L}} : \alpha \mapsto \mathcal{L}(G(\alpha))$, and $\theta_{\mathcal{L}}(\alpha) \subseteq \psi(\alpha)$ for all solutions $\psi : Q \to \mathcal{P}(\Sigma^*)$ and $\alpha \in Q$. For grammars in Chomsky normal form, we can use the following theorem to help check whether an arbitrary solution is the least solution. Note that this is an easy consequence of Theorem 2.3 in [3].

Theorem 3. *If G is a context-free grammar in Chomsky normal form, there is a* unique *solution θ to the system of equations generated by G in which $\epsilon \notin \theta(\alpha)$ for any $\alpha \in Q$.* □

In the theorem ϵ denotes the empty string. The solution θ in the previous theorem is the least solution, since G does not contain a production rule of the form $\alpha := \beta$, and so $\epsilon \notin \mathcal{L}(G(\alpha))$ for any $\alpha \in Q$.

Given a context-free grammar in Chomsky normal form G and a family of finite automata $\{ \mathcal{M}_\alpha \}_{\alpha \in Q}$, we can use Theorem 3 to check whether $\mathcal{L}(\mathcal{M}_\alpha) = \mathcal{L}(G(\alpha))$ for all $\alpha \in Q$.

Theorem 4. *Let G be a context-free grammar in Chomsky normal form with non-terminals Q. If $\mathcal{L}(G(\alpha))$ is regular for all $\alpha \in Q$, there is a constructable set of finite automata $\{ \mathcal{M}_\alpha \}_{\alpha \in Q}$ for which $\mathcal{L}(\mathcal{M}_\alpha) = \mathcal{L}(G(\alpha))$.*

Proof. We recursively enumerate the families of finite automata $\{ \mathcal{M}_\alpha \}_{\alpha \in Q}$ and check if $\mathcal{L}(\mathcal{M}_\alpha) = \mathcal{L}(G(\alpha))$ for each $\alpha \in Q$. If we let $\psi : Q \to \mathcal{P}(\Sigma^*)$ be the substitution $\alpha \mapsto \mathcal{L}(\mathcal{M}_\alpha)$, then the problem of checking whether $\mathcal{L}(\mathcal{M}_\alpha) = \mathcal{L}(G(\alpha))$ for all $\alpha \in Q$ reduces to deciding whether ψ is the unique solution satisfying Theorem 3. For each equation $\alpha = P_\alpha$, we can construct the automaton

$\mathcal{M}_{P_\alpha}$ with $\mathcal{L}(\mathcal{M}_{P_\alpha}) = P_\alpha\psi$ due to the effective closure of regular languages under union and concatenation. Moreover, one can check whether $\mathcal{L}(\mathcal{M}_\alpha) = \mathcal{L}(\mathcal{M}_{P_\alpha})$ for each $\alpha \in Q$ using the standard approaches for testing the equivalence of finite automata. So clearly we can check whether ψ is a solution. But it is also trivial to check whether $\epsilon \notin \mathcal{L}(\mathcal{M}_\alpha)$ for each $\alpha \in Q$. Thus it is decidable whether ψ satisfies the conditions in Theorem 3. If it does then $\psi(\alpha)$ must equal $\mathcal{L}(G(\alpha))$ for each $\alpha \in Q$. □

The key problem discussed in the section is determining whether the language $\mathcal{L}(G(\alpha))$ is regular for each non-terminal $\alpha \in Q$. One would expect this problem to be undecidable. Surprisingly, despite searching several texts, we could not find a decidability result for this problem. If $\mathcal{L}(G(\alpha))$ is regular for each non-terminal α, Theorem 4 shows that we can always show that by generating an equivalent family of finite automata. The other case is not so clear. Undecidability results for context-free languages such as Greibach's theorem (Sec. 8.7 in [15]) do not apply since they concern single context-free languages and this property concerns every non-terminal in a grammar. Theorem 4's result itself relied heavily upon the assumption that every non-terminal generates a regular language. The same approach does not work to construct a finite automata corresponding to a single non-terminal in G due to the undecidability of the equivalence problem for context-free grammars and regular languages.

6 Angluin's Algorithm

Though technically sound, if one were to implement the semi-algorithm using the naïve approach outlined above, the efficiency would likely be less than desired. Enumerating finite automata in order of increasing size takes exponential time relative to the size of the automaton. Each family of finite automata would need to be checked for equivalence, and this also takes exponential time. Unfortunately, we don't see a way to improve the exponential time required to check equivalence, but by applying techniques from learning theory, we decrease the number of equivalence queries we make so that if the algorithm eventually succeeds, we will have only required a polynomial number of queries relative to the size of the accepting family of automata eventually found.

A well-known algorithm in machine learning is Angluin's algorithm [1] for learning regular languages with oracles. For an arbitrary language L, this algorithm attempts to construct a finite automaton $\mathcal{M}$ such that $\mathcal{L}(\mathcal{M}) = L$ by asking questions to two oracles: a *membership* oracle that answers whether a string $u \in \Sigma^*$ is in L; an *equivalence* oracle that answers whether $\mathcal{L}(\mathcal{M}) = L$ and if not, provides a *counterexample* string $u \in \Sigma^*$ in the symmetric difference of L and $\mathcal{L}(\mathcal{M})$, i.e. $u \in L \oplus \mathcal{L}(\mathcal{M})$ with $L \oplus \mathcal{L}(\mathcal{M}) = (L - \mathcal{L}(\mathcal{M})) \cup (\mathcal{L}(\mathcal{M}) - L)$. Angluin's algorithm will terminate only if L is regular. However, given the appropriate oracles, one can attempt to apply it with any language, even languages not known to be regular. Due to space limitation of the paper, we roughly sketch below how Angluin's algorithm works. Readers are recommended to consult [17] for further details.

First we recall the definition of *Nerode's right congruence*: given a language $L \subseteq \Sigma^*$, the equivalence relation $\sim_L$ over Σ^* is the relation such that for $u, v \in \Sigma^*$, $u \sim_L v$ if and only if for all $w \in \Sigma^*$, $uw \in L \iff vw \in L$. It is known that a language L is regular if and only if the number of equivalence classes $|\Sigma^*/\!\sim_L|$ is finite. Angluin's algorithm maintains a data structure that stores two constructs: (1) a finite set $S \subseteq \Sigma^*$ of strings, each belonging to a distinct equivalence class in $\Sigma^*/\!\sim_L$, and (2) a finite set $D \subseteq \Sigma^*$ of distinguishing strings which in conjunction with the membership oracle, allows the algorithm to classify an arbitrary string into one of the known equivalence classes.

Initially, $S = \{ \epsilon \}$ and $D = \varnothing$. Using the membership oracle in conjunction with S and D, the algorithm constructs a deterministic finite automaton $\mathcal{M}$ such that $\mathcal{L}(\mathcal{M}) = L$ when $S = \Sigma^*/\!\sim_L$. The algorithm then queries the equivalence oracle which either succeeds and we are done, or returns a counterexample which can be analyzed to reveal at least one additional equivalence class representative in $\Sigma^*/\!\sim_L$ that is not in S. If L is regular, eventually the algorithm will learn all of the equivalence classes in $\Sigma^*/\!\sim_L$. If L is not regular, $\Sigma^*/\!\sim_L$ must be infinite and so the algorithm will not terminate.

Given a finite family of regular languages $\{ L_\alpha \}_{\alpha \in Q}$, Angluin's algorithm can be easily generalized to simultaneously learn a finite family of automata $\{ \mathcal{M}_\alpha \}_{\alpha \in Q}$ such that $\mathcal{L}(\mathcal{M}_\alpha) = L_\alpha$ for all $\alpha \in Q$. In this version, there must be a membership oracle for each language L_α, and an equivalence oracle which given a family $\{ \mathcal{M}_\alpha \}_{\alpha \in Q}$, returns true if $L_q = \mathcal{L}(\mathcal{M}_\alpha)$ for all $\alpha \in Q$, or a pair (α, u) where $\alpha \in Q$, and u is a counterexample in $L_\alpha \oplus \mathcal{L}(\mathcal{M}_\alpha)$. The generalized algorithm will terminate when L_α is regular for each $\alpha \in Q$.

In the context of this paper, we use Angluin's algorithm in conjunction with the flattened grammar $G_{f,\mathsf{d}(i)}$ with terminals $\Sigma_{f,\mathsf{d}(i)}$ and non-terminals Q. The algorithm attempts to construct of a family of finite automata $M = \{ \mathcal{M}_\alpha \}_{\alpha \in Q}$ for which $\mathcal{L}(\mathcal{M}_\alpha) = \mathcal{L}(G_{f,\mathsf{d}(i)}(\alpha))$. If the process succeeds, we can easily determine whether $\Sigma^{2+} \cap \mathcal{L}(G_{f,\mathsf{d}(i)}(P)) = \varnothing$ for each pair $(P, f) \notin \mathsf{d}(i)$ using standard techniques for finite automata. If we discover that $\Sigma^{2+} \cap \mathcal{L}(G_{f,\mathsf{d}(i)}(P)) \neq \varnothing$, we set $\mathsf{d}(i+1) = \mathsf{d}(i) \uplus \{ (P, f) \}$ and repeat the process for $\mathsf{d}(i+1)$.

To apply Angluin's algorithm, we need to provide the membership oracles and equivalence oracle needed given a context-free grammar G with non-terminals Q and terminals Σ. The membership oracle for each non-terminal $\alpha \in Q$ can be provided by a context-free language parser that parses a string $u \in \Sigma^*$ and returns true if $u \in \mathcal{L}(G(\alpha))$. Given the family $\{ \mathcal{M}_\alpha \}_{\alpha \in Q}$, our equivalence oracle forms the mapping $\theta : \alpha \mapsto \mathcal{L}(\mathcal{M}_\alpha)$ and checks if it is the solution to the equations generated by G satisfying Theorem 3. In case that θ is not the solution, the equivalence oracle must analyze the mapping to return a counterexample. The algorithm we use is presented in Fig. 1. We can show the correctness of the equivalence oracle by stating the following theorem:

Theorem 5. *Given a family a context-free grammar G in Chomsky normal form with non-terminals Q and terminals Σ, and a family $\{ \mathcal{M}_\alpha \}_{\alpha \in Q}$ of finite automata over Σ, the algorithm* check_equiv *in Fig. 1*

- *returns* true *if $\mathcal{L}(G(\alpha)) = \mathcal{L}(\mathcal{M}_\alpha)$ for all $\alpha \in Q$; and otherwise,*

PROCEDURE check_equiv

INPUT $G(\_) = (\Sigma, Q, \_, P)$: a context-free grammar
$\{\mathcal{M}_\alpha\}_{\alpha \in Q}$: a family of finite automata over Σ

OUTPUT true or (α, u) for some $\alpha \in Q$ and $u \in \Sigma^*$

```
let θ be the substitution α ↦ L(M_α);
for each α ∈ Q do
  if ε ∈ L(M_α) then return (α, ε);

  if L(M_α) ≠ P_α θ then
    choose u ∈ L(M_α) ⊕ P_α θ
    if u ∈ L(M_α) ⊕ L(G(α)) then return (α, u)
    else
      for each α := βγ ∈ P and u = st do
        if s ∈ L(M_β) ⊕ L(G(β)) then return (β, s);
        if t ∈ L(M_γ) ⊕ L(G(γ)) then return (γ, t)
      od;
od;
return true
```

Fig. 1. Checking language equivalence

– *returns a pair* (β, w) *such that* $w \in \mathcal{L}(G(\beta)) \oplus \mathcal{L}(\mathcal{M}_\beta)$.

Proof sketch. Termination of this procedure is straightforward, and it is easy to verify that when a pair is returned at a return statement, it is indeed a counterexample. The non-trivial part of this theorem is that if the outer loop terminates without returning a counterexample, check_equiv returns true and $\mathcal{L}(G(\alpha)) = \mathcal{L}(\mathcal{M}_\alpha)$ is guaranteed. We obtain this property by showing that if $\mathcal{L}(\mathcal{M}_\alpha) \neq \mathcal{L}(\mathcal{G}(\alpha))$ for some $\alpha \in Q$ and the outer loop is executed, the body of the loop is guaranteed to return a pair. See Appendix A.5 for details. □

When equipped with context-free language parsers as membership oracles and chec_equiv as an equivalence oracle, Angluin's algorithm accomplishes the same goal as the simple enumeration-based algorithm used to prove Theorem 4. However, the approach in this section is much more efficient. In searching for a solution, the enumeration algorithm used in Theorem 4 checks equivalence of every family of finite automata in order of increasing size. The total number of equivalence checks will be exponential relative to the size of the final output. Since each equivalence check itself takes exponential time, the enumeration algorithm double exponential relative to the size of the final output. In contrast, Angluin's algorithm makes a number of oracle queries that is polynomial [1] to the size of the final output. The equivalence oracle itself takes exponential time, and so the total time of the new algorithm is a single exponential relative to the size of the final output.

7 Concluding Remarks

The tree automata techniques developed in this paper are not only for theoretical use. The emptiness checking procedure explained in the previous two sections has been implemented in the CETA library [14]. This software provides the function for emptiness checking with not only associativity and commutativity axioms, but identity axioms as well. The identity axiom for a function symbol f with a unit symbol c is the equations of the forms $f(c, x) = x$ and $f(x, c) = x$. In CETA, identity axioms in a propositional tree automaton are converted into the rewrite rules $f(c, x) \to x$ and $f(x, c) \to x$ in conjunction with a specialized Knuth-Bendix style completion procedure modulo associativity and commutativity that preserves the set of reachable states for each term.

Though still a prototype, CETA has been integrated to work with the reachability analysis tool ACTAS [22], as well as the next generation sufficient completeness tool for Maude. In future project we plan to apply the new ACTAS for the tree automata based verification of infinite state systems including network protocols. In the Maude sufficient completeness tool, we use CETA by posing the sufficient completeness problem of an equational specification as a PTA emptiness problem. Sufficient completeness is a property of equational specifications that guarantees that enough equations have been specified so that defined operations are fully specified on all relevant data. We already experienced that CETA is useful in this context, as it allowed the checker to find a subtle bug in Maude involving lists formed from an associative operator, and also to verify the correctness of the bug-fix by proving that the language accepted by the corresponding tree automaton is empty, where the automaton often contains a theory with associativity.

References

1. D. Angluin: *Learning Regular Sets from Queries and Counterexamples*, Information and Computation 75, pp. 87–106, Elsevier, 1987.
2. A. Armando, D. Basin, Y. Boichut, Y. Chevalier, L. Compagna, J. Cuellar, P. Hankes Drielsma, P.-C. Heám, O. Kouchnarenko, J. Mantovani, S. Mödersheim, D. von Oheimb, M. Rusinowitch, J. Santiago, M. Turuani, L. Viganò and L. Vigneron: *The AVISPA Tool for the Automated Validation of Internet Security Protocols and Applications*, Proc. of 17th CAV, Edinburgh (UK), LNCS 3576, pp. 281–285, Springer-Verlag, 2005.
3. J. Autebert, J. Berstel and L. Boasson: *Context-Free Languages and Push-Down Automata*, *Handbook of Formal Languages* 1, pp. 111–174. Springer-Verlag, 1997.
4. F. Baader and T. Nipkow: *Term Rewriting and All That*, Cambridge University Press, 1998.
5. Y. Boichut, P.-C. Heám and O. Kouchnarenko: *Automatic Verification of Security Protocols Using Approximations*, technical report RR-5727, INRIA, October 2005.
6. A. Bouhoula, J.P. Jouannaud and J. Meseguer: *Specification and Proof in Membership Equational Logic*, TCS 236, pp. 35–132, Elsevier, 2000.
7. H. Comon, M. Dauchet, R. Gilleron, F. Jacquemard, D. Lugiez, S. Tison and M. Tommasi: *Tree Automata Techniques and Applications*, incomplete draft, 2005. Available at `http://www.grappa.univ-lille3.fr/tata`

8. P. Devienne, J.-M. Talbot and S. Tison: *Set-Based Analysis for Logic Programming and Tree Automata*, Proc. of 4th SAS, Paris (France), LNCS 1302, pp. 127–140, Springer-Verlag, 1997.
9. D.-Z. Du and K. Ko: *Theory of Computational Complexity*, John Wiley and Sons, 2000.
10. J.P. Gallagher and G. Puebla: *Abstract Interpretation over Non-Deterministic Finite Tree Automata for Set-Based Analysis of Logic Programs*, Proc. of 4th PADL, Portland (USA), LNCS 2257, pp. 243–261, Springer-Verlag, 2002.
11. T. Genet and F. Klay: *Rewriting for Cryptographic Protocol Verification*, Proc. of 17th CADE, Pittsburgh (USA), LNCS 1831, pp. 271–290, Springer-Verlag, 2000.
12. S. Ginsburg: *The Mathematical Theory of Context-Free Languages*, McGraw-Hill, 1966.
13. J. Hendrix, H. Ohsaki and J. Meseguer: *Sufficient Completeness Checking with Propositional Tree Automata*, technical report UIUCDCS-R-2005-2635, Department of Computer Science, University of Illinois at Urbana-Champaign, 2005. Available at `http://texas.cs.uiuc.edu/`
14. J. Hendrix: *CETA: A Library for Equational Tree Automata*, Department of Computer Science, University of Illinois at Urbana-Champaign, 2006. Software available under GPL license at `http://texas.cs.uiuc.edu/ceta/`
15. J.E. Hopcroft and J.D. Ullman: *Introduction to Automata Theory, Languages, and Computation*, Addison-Wesley Publishing Company, 1979.
16. H. Hosoya, J. Vouillon and B.C. Pierce: *Regular Expression Types for XML*, Proc. of 5th ICFP, Montreal (Canada), SIGPLAN Notices 35(9), pp. 11–22, ACM, 2000.
17. M. Kearns and U. Vazirani: *An Introduction to Computational Learning Theory*, MIT Press, 1994.
18. N. Klarlund and A. Møller: *MONA Version 1.4 User Manual*, BRICS Notes Series NS-01-1, Department of Computer Science, University of Aarhus, 2001.
19. D. Lugiez: *Multitree Automata That Count*, TCS 333, pp. 225–263, Elsevier, 2005.
20. M. Nederhof: *Practical Experiments with Regular Approximation of Context-Free Languages*, Computational Linguistics 26(1), pp. 17–44, 2000.
21. H. Ohsaki, J.-M. Talbot, S. Tison and Y. Roos: *Monotone AC-Tree Automata*, Proc. of 12th LPAR, Montego Bay (Jamaica), LNAI 3855, pp. 337–351, Springer-Verlag, 2005.
22. H. Ohsaki and T. Takai: *ACTAS : A System Design for Associative and Commutative Tree Automata Theory*, Proc. of 5th RULE, Aachen (Germany), ENTCS 124, pp. 97–111, Elsevier, 2005.
23. H. Ohsaki and T. Takai: *Decidability and Closure Properties of Equational Tree Languages*, Proc. of 13th RTA, Copenhagen (Denmark), LNCS 2378, pp. 114–128, Springer-Verlag, 2002.
24. H. Ohsaki: *Beyond Regularity: Equational Tree Automata for Associative and Commutative Theories*, Proc. of 15th CSL, Paris (France), LNCS 2142, pp. 539–553, Springer-Verlag, 2001.
25. R. Parikh: *On Context-Free Languages*, JACM 13(4), pp. 570–581, 1966.
26. H. Seidl, T. Schwentick and A. Muscholl: *Numerical Document Queries*, Proc. of 22nd PODS, San Diego (USA), pp. 155–166, ACM, 2003.
27. I. Yagi, Y. Takata and H. Seki: *A Static Analysis Using Tree Automata for XML Access Control*, Proc. of 3rd ATVA, Taipei (Taiwan), LNCS 3707, pp. 234–247, Springer-Verlag, 2005.
28. K.N. Verma: *Two-Way Equational Tree Automata for AC-Like Theories: Decidability and Closure Properties*, Proc. of 14th RTA, Valencia (Spain), LNCS 2706, pp. 180–197, Springer-Verlag, 2003.

A Proofs

In this section we suppose $\mathcal{A} = (\mathcal{E}, Q, \phi, \Delta)$ to be a PTA with $\mathcal{E} = (F, E)$ containing only associativity and commutativity axioms. Before proving Theorem 2, we need to introduce a number of lemmata, related to free, associative, and associative and commutative symbols.

A.1 Free Symbols

It is straightforward to show by induction on proof strutures, the following lemma about terms whose root is a free symbol:

Lemma 5. *Given terms $f(t_1, \ldots, t_n), u \in \mathcal{T}(F \cup Q)$, if f is a free symbol and $f(t_1, \ldots, t_n) =_{\mathcal{E}} u$ then u must have the form $f(u_1, \ldots, u_n)$ where $t_i =_{\mathcal{E}} u_i$ for all $i \leq n$.* □

This lemma then leads to the following:

Lemma 6. *Given terms $f(t_1, \ldots, t_n), u \in \mathcal{T}(F \cup Q)$, if f is a free symbol and $f(t_1, \ldots, t_n) \rightarrow^*_{\mathcal{A}} u$ then either: (1) u is a state in Q or (2) it has the form $f(u_1, \ldots, u_n)$ where $t_i \rightarrow^*_{\mathcal{A}} u_i$ for all $i \leq n$.*

Proof. If $f(t_1, \ldots, t_n) \rightarrow_{\mathcal{A}} u$, then there is a chain with the form

$$f(t_1, \ldots, t_n) =_{\mathcal{E}} C_1[l_1] \rightarrow_{\mathcal{A}} C_1[\alpha_1] =_{\mathcal{E}} C_2[l_2] \rightarrow_{\mathcal{A}} \cdots\cdots C_n[\alpha_n] =_{\mathcal{E}} u.$$

Our proof is by induction on this chain. In the base case, $f(t_1, \ldots, t_n) =_{\mathcal{E}} u$. By Lemma 5, u must have a form satisfying (2).

In the inductive case, there is a context C and rule $f(\alpha_1, \ldots, \alpha_n) \rightarrow \alpha \in \Delta$ such that $f(t_1, \ldots, t_n) =_{\mathcal{E}} C[f(\alpha_1, \ldots, \alpha_n)]$ and $C[\alpha] \rightarrow^*_{\mathcal{A}} u$. If the context C is the empty context $\Box$, then $C[\alpha] = \alpha$. Since no rule in Δ or equation in E can apply to α, it follows that $u = \alpha$. Otherwise, C is not the empty context. It follows by Lemma 5 that $\text{root}(C[l]) = f$. It also follows that $C[\alpha]$ must have the form $f(v_1, \ldots, v_n)$ with $t_i =_{\mathcal{E}} v_i$ or $t_i \rightarrow_{\mathcal{A}} v_i$ for all $i \leq n$. Together this implies that the conditions of our lemma are satisfied for $C[\alpha]$ so by induction u has one of the required forms. □

Lemma 7. *Given a term $f(t_1, \ldots, t_n) \in \mathcal{T}(F)$ where $f \in F$ a free symbol, If we let $P_i = \mathsf{reach}_{\mathcal{A}}(t_i)$ for all $i \leq n$, then*

$$\mathsf{reach}_{\mathcal{A}}(f(t_1, \ldots, t_n)) = \mathsf{reach}_{\mathcal{A}}(f(P_1, \ldots, P_n)).$$

Proof. If $q \in \mathsf{reach}_{\mathcal{A}}(f(t_1, \ldots, t_n))$, then $f(t_1, \ldots, t_n) \rightarrow^*_{\mathcal{A}} q$. As $f(t_1, \ldots, t_n) \neq_{\mathcal{E}} q$, there must be a term $u \in \mathcal{T}(F)$ where $f(t_1, \ldots, t_n) \rightarrow^*_{\mathcal{A}} u \rightarrow_{\mathcal{A}} q$. By Lemma 6, u must have the form $f(u_1, \ldots, u_n)$ where $t_i \rightarrow^*_{\mathcal{A}} u_i$ for all $i \leq n$. Since u must also match the left-hand side of a rule at the right position, then there must be a rule of the form $f(p_1, \ldots, p_n) \rightarrow q$ where $u = f(p_1, \ldots, p_n)$ and

$t_i \to_{\mathcal{A}}^* p_i$ for all $i \leq n$. But this implies that $q \in \mathsf{reach}_{\mathcal{A}}(f(P_1, \ldots, P_n))$. Thus $q \in \mathsf{reach}_{\mathcal{A}}(f(t_1, \ldots, t_n)$ implies $q \in \mathsf{reach}_{\mathcal{A}}(f(P_1, \ldots, P_n))$, i.e.

$$\mathsf{reach}_{\mathcal{A}}(f(t_1, \ldots, t_n)) \subseteq \mathsf{reach}_{\mathcal{A}}(f(P_1, \ldots, P_n)).$$

On the other hand, as $P_i = \mathsf{reach}_{\mathcal{A}}(t_i)$, $t_i \to_{\mathcal{A}}^* p_i$ for each $p_i \in P_i$ and $i \leq n$. If $f(p_1, \ldots, p_n) \to q \in \Delta$, then $f(t_1, \ldots, t_n) \to_{\mathcal{A}}^* q$. So

$$\mathsf{reach}_{\mathcal{A}}(f(P_1, \ldots, P_n)) \subseteq \mathsf{reach}_{\mathcal{A}}(f(t_1, \ldots, t_n)).$$

□

A.2 Associative Symbols

We now turn our attention to proving several lemmata dealing with associative symbols. In the subsection, let $\mathsf{d}(i)$ be a subset of $\mathcal{P}(Q) \times F$, and let f be an associative symbol. We first define contexts that are considered *maximal* relative to a given set of terms.

Definition 4. *Given terms $t_1, \ldots, t_n \in \mathcal{T}(F)$, a context C with n-holes is called a* maximal f-context *for the term $C[t_1, \ldots, t_n]$ when $C \in \mathcal{T}(\{f, \Box_1, \ldots, \Box_2\})$ and $\mathrm{root}(t_i) \neq f$ for all $i \leq n$.*

First, the following lemma is an easy consequence of the fact that the only has axiom in $\mathcal{E}$ involving f is a single associativity axiom.

Lemma 8. *Given terms $C[t_1, \ldots, t_n], u \in \mathcal{T}(F)$ with $n \geq 2$ and C a maximal f-context, if $C[t_1, \ldots, t_n] \to_{\mathcal{A}}^* u$, then either u is a state in Q or u has the form $f(u_1, u_2)$ where for some index $i < n$, there are contexts C_1, C_2 such that $C_1[t_1, \ldots, t_i] \to_{\mathcal{A}}^* u_1$ and $C_2[t_{i+1}, \ldots, t_n] \to_{\mathcal{A}}^* u_2$.* □

Lemma 9. *Let $t_1, \ldots, t_n$ be terms in $\mathcal{T}(F)$ such that $(\mathrm{root}(t_j), \mathsf{reach}_{\mathcal{A}}(t_j)) \in \mathsf{d}(i)$ and $\mathrm{root}(t_j) \neq f$ for all $j \leq n$. For each state $\alpha \in Q$, if $C[t_1, \ldots, t_n] \to_{\mathcal{A}}^* \alpha$ for a maximal f-context C, then $\alpha :=_{G_{f,\mathsf{d}(i)}} \mathsf{reach}_{\mathcal{A}}(t_1) \ldots \mathsf{reach}_{\mathcal{A}}(t_n)$.*

Proof. We prove this by induction on n. In the base case, $n = 1$ and so the context $C = \Box$. Thus $C[t_1, \ldots, t_n] = t_1$. By the definition of $G_{f,\mathsf{d}(i)}$, since $(\mathrm{root}(t_1), \mathsf{reach}_{\mathcal{A}}(t_1)) \in \mathsf{d}(i)$ and $\mathrm{root}(t_1) \neq f$, there is a production rule $\alpha := \mathsf{reach}_{\mathcal{A}}(t_1)$ in $G_{f,\mathsf{d}(i)}$. Thus $\alpha :=_{G_{f,\mathsf{d}(i)}} \mathsf{reach}_{\mathcal{A}}(t_1)$.

In the inductive case, $n \geq 2$ and so C is non empty. So if $C[t_1, \ldots, t_n] \to_{\mathcal{A}}^* u$, there must be a final rewrite step

$$C[t_1, \ldots, t_n] \to_{\mathcal{A}}^* f(\beta, \gamma) \to_{\mathcal{A}} \alpha$$

where $f(\beta, \gamma) \to \alpha$ is a rule in $\mathcal{A}$. By Lemma 8, it follows that for some $j < n$, there are contexts C_1, C_2 such that $C_1[t_1, \ldots, t_j] \to_{\mathcal{A}}^* \beta$ and $C_2[t_{j+1}, \ldots, t_n] \to_{\mathcal{A}}^* \gamma$. By induction

$$\beta :=_{G_{f,\mathsf{d}(i)}} \mathsf{reach}_{\mathcal{A}}(t_1), \ldots, \mathsf{reach}_{\mathcal{A}}(t_j), \text{ and}$$
$$\gamma :=_{G_{f,\mathsf{d}(i)}} \mathsf{reach}_{\mathcal{A}}(t_{j+1}), \ldots, \mathsf{reach}_{\mathcal{A}}(t_n).$$

Thus by the definition of $G_{f,\mathsf{d}(i)}$, $\alpha :=_{G_{f,\mathsf{d}(i)}} \mathsf{reach}_{\mathcal{A}}(t_1) \ldots \mathsf{reach}_{\mathcal{A}}(t_n)$. □

Lemma 10. *Let $t_1, \ldots, t_n$ be terms in $\mathcal{T}(F)$ such that $(\mathrm{root}(t_j), \mathsf{reach}_{\mathcal{A}}(t_j)) \in \mathsf{d}(i)$ and $\mathrm{root}(t_j) \neq f$ for all $j \leq n$. For each state $\alpha \in Q$, if $\alpha :=_{G_{f,\mathsf{d}(i)}} \mathsf{reach}_{\mathcal{A}}(t_1), \ldots, \mathsf{reach}_{\mathcal{A}}(t_n)$, then $C[t_1, \ldots, t_n] \to^*_{\mathcal{A}} \alpha$ for each maximal f-context C.*

Proof. We prove this by induction on n. In the base case, $n = 1$, and so $C[t_1, \ldots, t_n] = t_1$. From the definition of $G_{f,\mathsf{d}(i)}$, $\alpha \in \mathsf{reach}_{\mathcal{A}}(t_1)$. Thus by the definition of $\mathsf{reach}_{\mathcal{A}}$, $t_1 \to^*_{\mathcal{A}} \alpha$.

In the inductive case, $n \geq 2$ and so C is non empty. Since $\alpha :=_{G_{f,\mathsf{d}(i)}} \mathsf{reach}_{\mathcal{A}}(t_1) \ldots \mathsf{reach}_{\mathcal{A}}(t_n)$, there must be a production rule $\alpha := \beta\gamma$ in $G_{f,\mathsf{d}(i)}$ such that for some $j < n$,

$$\beta :=_{G_{f,\mathsf{d}(i)}} \mathsf{reach}_{\mathcal{A}}(t_1), \ldots, \mathsf{reach}_{\mathcal{A}}(t_j), \text{ and}$$
$$\gamma :=_{G_{f,\mathsf{d}(i)}} \mathsf{reach}_{\mathcal{A}}(t_{j+1}), \ldots, \mathsf{reach}_{\mathcal{A}}(t_n).$$

By induction for contexts C_1, C_2,

$$C_1[t_1, \ldots, t_j] \to^*_{\mathcal{A}} \beta \qquad \text{and} \qquad C_2[t_{j+1}, \ldots, t_n] \to^*_{\mathcal{A}} \gamma.$$

Since $\alpha := \beta\gamma$ is in $G_{f,\mathsf{d}(i)}$, there must be a rule $f(\beta, \gamma) \to \alpha$ in $\mathcal{A}$. Thus,

$$C[t_1, \ldots, t_n] =_{\mathcal{E}} f(C_1[t_1, \ldots, t_j], C_2[t_{j+1}, \ldots, t_n]) \to^*_{\mathcal{A}} \alpha.$$

□

The following corollary is immediate from the definitions of $\mathsf{reach}_{\mathcal{A}}$ and $\mathcal{L}(G_{f,\mathsf{d}(i)}(P))$ using Lemmata 9 and 10.

Corollary 1. *Let $t_1, \ldots, t_n$ be terms in $\mathcal{T}(F)$ such that $(\mathrm{root}(t_j), \mathsf{reach}_{\mathcal{A}}(t_j)) \in \mathsf{d}(i)$ and $\mathrm{root}(t_j) \neq f$ for all $j \leq n$. For each set $P \subseteq Q$,*

$$\mathsf{reach}_{\mathcal{A}}(C[t_1, \ldots, t_n]) = P \iff (\mathsf{reach}_{\mathcal{A}}(t_1), \ldots, \mathsf{reach}_{\mathcal{A}}(t_n)) \in \mathcal{L}(G_{f,\mathsf{d}(i)}(P)).$$

□

A.3 Associative and Commutative Symbols

We now turn our attention to proving several lemmata dealing with associative and commutative symbols. In the subsection, let $\mathsf{d}(i)$ be a subset of $\mathcal{P}(Q) \times F$, let f be an associative and commutative symbol, and let $C[t_1, \ldots, t_n] \in \mathcal{T}(F)$ be a term such that $n \geq 1$, $\mathrm{root}(t_i) \neq f$ for all $i \leq n$, and $C \in \mathcal{T}(\{f, \Box_1, \ldots, \Box_n\})$ is a context only containing the AC symbol f with n holes. We begin with two definitions related to removing the equations from a tree automaton.

Definition 5. *Given a PTA $\mathcal{A} = (\mathcal{E}, Q, \phi, \Delta)$ with an associative and commutative symbol $f \in F_{\mathrm{A}} \cap F_{\mathrm{C}}$, let $\mathcal{A}_{f,\mathrm{A}}$ denote the PTA formed from $\mathcal{A}$ by removing the commutativity axiom $f(x, y) = f(y, x)$ from the equational theory used in A, i.e., $\mathcal{A}_{f,\mathrm{A}} = (\mathcal{E}_{f,\mathrm{a}}, Q, \phi, \Delta)$ where $\mathcal{E}_{f,\mathrm{a}} = (F, E - \{ f(x, y) = f(y, x) \})$.* □

Definition 6. *Given a PTA $\mathcal{A} = (\mathcal{E}, Q, \phi, \Delta)$, let $\mathcal{A}_\varnothing$ denote the PTA formed by removing all equations from $\mathcal{E}$, i.e., $\mathcal{A}_\varnothing = (\mathcal{E}_\varnothing, Q, \phi, \Delta)$ where $\mathcal{E}_\varnothing = (F, \varnothing)$.* □

We first note the following observation that is an obvious consequence of Lemma 2 in [24] since associativity and commutativity equations are linear.

Lemma 11. *For each term $t \in \mathcal{T}(F)$ and state $\alpha \in Q$, if $t \to^*_{\mathcal{A}} \alpha$, then there is a term $u \in \mathcal{T}(F)$ such that $t =_{\mathcal{E}} u$ and $u \to^*_{\mathcal{A}_\varnothing} \alpha$.* □

We next note the following lemma which is an easy consequence of the fact that f is associative and commutative.

Lemma 12. *If $C[t_1, \ldots, t_n] =_{\mathcal{E}} u$, then u must be of the form $C'[u_1, \ldots, u_n]$ with C' a maximal context containing the AC symbol f and n holes and $u_i =_{\mathcal{E}} t_{\pi(i)}$ for each i where $\pi : [1, n] \to [1, n]$ is a permutation.* □

Now we are able to prove the main technical result of this subsection. This is the AC counterpart to Lemma 9 and Lemma 10 in the previous subsection.

Lemma 13. *For each state $\alpha \in Q$,*

$$C[t_1, \ldots, t_n] \to^*_{\mathcal{A}} \alpha \iff \#(\mathsf{reach}_{\mathcal{A}}(t_1), \ldots, \mathsf{reach}_{\mathcal{A}}(t_n)) \in \mathcal{S}(G_{f,\mathsf{d}(i)}(\alpha)).$$

Proof. If $C[t_1, \ldots, t_n] \to^*_{\mathcal{A}} \alpha$, then by Lemma 11, there is a term $u \in \mathcal{T}(F)$ such that $C[t_1, \ldots, t_n] =_{\mathcal{E}} u$ and $u \to^*_{\mathcal{A}_\varnothing} \alpha$. By Lemma 12, u must have the form $C'[u_1, \ldots, u_n]$ where $u_i =_{\mathcal{E}} t_{\pi(i)}$ for some permutation $\pi : [1, n] \to [1, n]$. Since $u \to^*_{\mathcal{A}_\varnothing} \alpha$, clearly $C'[u_1, \ldots u_n] \to^*_{\mathcal{A}_{f,\mathrm{A}}} \alpha$. Thus by Lemma 9,

$$\alpha :=_{G_{f,\mathcal{P}(Q)\times F}} \mathsf{reach}_{\mathcal{A}_{f,\mathrm{A}}}(u_1) \ldots \mathsf{reach}_{\mathcal{A}_{f,\mathrm{A}}}(u_n).$$

As $\mathsf{reach}_{\mathcal{A}_{f,\mathrm{A}}}(u_i) \subseteq \mathsf{reach}_{\mathcal{A}}(t_{\pi(i)}$ for all $i \leq n$,

$$\alpha :=_{G_{f,\mathsf{d}(i)}} \mathsf{reach}_{\mathcal{A}}(t_{\pi(1)} \ldots \mathsf{reach}_{\mathcal{A}}(t_{\pi(n)}).$$

It then follows that $\mathsf{reach}_{\mathcal{A}}(t_1) \ldots \mathsf{reach}_{\mathcal{A}}(t_n) \in \mathcal{S}(G_{f,\mathsf{d}(i)}(\alpha))$.

On the other hand, if $\#(\mathsf{reach}_{\mathcal{A}}(t_1), \ldots, \mathsf{reach}_{\mathcal{A}}(t_n)) \in \mathcal{S}(G_{f,\mathsf{d}(i)}(\alpha))$, then there must be a permutation $\pi : [1, n] \to [1, n]$ such that

$$\alpha :=_{G_{f,\mathsf{d}(i)}} \mathsf{reach}_{\mathcal{A}}(t_{\pi(1)}, \ldots, \mathsf{reach}_{\mathcal{A}}(t_{\pi(n)}).$$

It then follows by Lemma 10 that $C[t_\pi(1), \ldots, t_\pi(n)] \to^*_{\mathcal{A}_{f,\mathrm{A}}} \alpha$. As $C[t_1, \ldots t_n] =_{\mathcal{E}} C[t_\pi(1), \ldots t_\pi(n)]$ and $\to^*_{\mathcal{A}_{f,\mathrm{A}}} \subseteq \to^*_{\mathcal{A}}$, $C[t_1, \ldots, t_n] \to^*_{\mathcal{A}} \alpha$. □

The following corollary is immediate from the definitions of $\mathsf{reach}_{\mathcal{A}}$ and $\mathcal{S}(G_{f,\mathsf{d}(i)}(P))$ using Lemma 13.

Corollary 2. *For each set $P \subseteq Q$,*

$$\mathsf{reach}_{\mathcal{A}}(C[t_1, \ldots, t_n]) = P \iff \#(\mathsf{reach}_{\mathcal{A}}(t_1), \ldots, \mathsf{reach}_{\mathcal{A}}(t_n)) \in \mathcal{S}(G_{f,\mathsf{d}(i)}(P)).$$

□

A.4 Putting It Together

Now that we have proven most of the preliminary results, we are ready to begin addressing Theorem 2. This theorem is most easily seen as the consequence of a couple results. First, it helps to make the following fairly obvious observations:

Lemma 14. *Let* $\mathsf{d}(i) \subseteq \mathsf{derive}(\mathcal{A})$*. For each pair* $(P, f) \in \mathsf{d}(i)$*, there is a term* t *such that* $\mathrm{root}(t) = f$ *and* $\mathsf{reach}_{\mathcal{A}}(t) = P$*.*

Proof. By the assumption that $\mathsf{d}(i) \subseteq \mathsf{derive}(\mathcal{A})$ and the definition of $\mathsf{derive}(\mathcal{A})$. □

Lemma 15. *Given an symbol* $f \in F$*, and a set* $\mathsf{d}(i) \subseteq \mathsf{derive}(\mathcal{A})$*, if* u *is a string in* $\Sigma^*_{f,\mathsf{d}(i)}$ *with length* n*, then there are terms* $t_1, \ldots, t_n \in T(F)$ *where* $u = \mathsf{reach}_{\mathcal{A}}(t_1), \ldots, \mathsf{reach}_{\mathcal{A}}(t_n)$ *and* $\mathrm{root}(t_i) \neq f$ *for all* $i \leq n$*.*

Proof. Let $u = P_1 \ldots P_n$. By the definition of $\Sigma_{f,\mathsf{d}(j)}$, there must be a function symbol $g_j \neq f$ for each $j \leq n$ such that $(P_j, g_j) \in \mathsf{d}(i)$. As $\mathsf{d}(i) \subseteq (\mathcal{A})$, there must be terms $t_1, \ldots, t_n \in T(F)$ such that $\mathsf{reach}_{\mathcal{A}}(t_j) = P_j$ and $\mathrm{root}(t_j) = g_j$ for each $j \leq n$. □

We now are ready to begin proving the key two lemmata required to show Theorem 2.

Lemma 16. *If* $\mathsf{d}(i) \subseteq \mathsf{derive}(\mathcal{A})$*, and* $\mathsf{d}(i+1)$ *is obtained by applying one of the rules (1) – (3) to* $\mathsf{d}(i)$*, then* $\mathsf{d}(i+1) \subseteq \mathsf{derive}(\mathcal{A})$*.*

Proof. We prove this by considering separately each of the possible rules (1) – (3) that may be used to form $\mathsf{d}(i+1)$.

First we consider the case where rule (1) is used with a free symbol $f \in F$:

$$f \notin F_{\mathrm{A}} \cup F_{\mathrm{C}} : \quad \frac{\{\,(P_1, f_1), \ldots, (P_n, f_n)\,\} \in \mathsf{d}(i)}{\mathsf{d}(i+1) = \mathsf{d}(i) \uplus \{\,(\,\mathsf{reach}_{\mathcal{A}}(f(P_1, \ldots, P_n)), f\,)\,\}}.$$

By Lemma 14, for each pair $(P_j, f_j) \in \mathsf{d}(i)$ with $j \leq n$, there is a term $t_j \in \mathcal{T}(F)$ such that $\mathsf{reach}_{\mathcal{A}}(t_j) = P_j$. By Lemma 7, $(\mathsf{reach}_{\mathcal{A}}(f(P_1, \ldots, P_n)), f) = (\mathsf{reach}_{\mathcal{A}}(f(t_1, \ldots, t_n)), f)$. It follows that $(\mathsf{reach}_{\mathcal{A}}(f(t_1, \ldots, t_n)), f) \in \mathsf{derive}(\mathcal{A})$, and thus $\mathsf{d}(i+1) \subseteq \mathsf{derive}(\mathcal{A})$.

Now we consider the case where rule (2) is used with an associate symbol $f \in F$:

$$f \in F_{\mathrm{A}} - F_{\mathrm{C}} : \quad \frac{P \subseteq Q \qquad \Sigma^{2+}_{f,\mathsf{d}(i)} \cap \mathcal{L}(G_{f,\mathsf{d}(i)}(P)) \neq \varnothing}{\mathsf{d}(i+1) = \mathsf{d}(i) \uplus \{\,(\,P, f\,)\,\}}.$$

As $\Sigma^{2+}_{f,\mathsf{d}(i)} \cap \mathcal{L}(G_{f,\mathsf{d}(i)}(P)) \neq \varnothing$, there must be a string $u \in \Sigma^*_{f,\mathsf{d}(i)}$ such that $|u| \geq 2$ and $u \in \mathcal{L}(G_{f,\mathsf{d}(i)}(P))$. Let $n = |u|$. By Lemma 15, there must be terms $t_1, \ldots, t_n \in \mathcal{T}(F)$ such that $u = \mathsf{reach}_{\mathcal{A}}(t_1), \ldots, \mathsf{reach}_A A(t_n)$ and $\mathrm{root}(t_i) \neq f$ for all $i \leq n$. Let C be a context formed from f with n holes. As $u \in \mathcal{L}(G_{f,\mathsf{d}(i)}(P))$, by Cor. 1, $\mathsf{reach}_{\mathcal{A}}(C[t_1, \ldots, t_n]) = P$. Since $n \geq 2$, $C \neq \Box$ and

thus $\mathrm{root}(C[t_1, \ldots, t_n) = f$. It then follows that $(P, f) \in \mathsf{derive}(\mathcal{A})$, and thus $\mathsf{d}(i+1) \subseteq \mathsf{derive}(\mathcal{A})$.

Finally we consider the case where rule (3) is used with an AC symbol $f \in F$:

$$f \in F_{\mathrm{A}} \cap F_{\mathrm{C}} : \quad \frac{P \subseteq Q \qquad \mathbb{N}^{>\mathbf{1}} \cap \mathcal{S}(G_{f,\mathsf{d}(i)}(P)) \neq \varnothing}{\mathsf{d}(i+1) = \mathsf{d}(i) \uplus \{ (P, f) \}}.$$

As $\mathbb{N}^{>\mathbf{1}} \cap \mathcal{S}(G_{f,\mathsf{d}(i)}(P)) \neq \varnothing$, there must be a string $u \in \Sigma_{f,\mathsf{d}(i)}$ with length at least 2 such that $\#(u) \in \mathcal{S}(G_{f,\mathsf{d}(i)}(P))$. Let n be the length of u. By Lemma 15, there must be terms $t_1, \ldots, t_n \in \mathcal{T}(F)$ such that $u = \mathsf{reach}_{\mathcal{A}}(t_1), \ldots, \mathsf{reach}_A A(t_n)$ and $\mathrm{root}(t_i) \neq f$ for all $i \leq n$. Let C be a context formed from f with n holes. As $u \in \mathcal{S}(G_{f,\mathsf{d}(i)}(P))$, by Cor. 2, $\mathsf{reach}_{\mathcal{A}}(C[t_1, \ldots, t_n]) = P$. Since $n \geq 2$, $C \neq \Box$ and thus $\mathrm{root}(C[t_1, \ldots, t_n) = f$. It then follows that $(P, f) \in \mathsf{derive}(\mathcal{A})$, and thus $\mathsf{d}(i+1) \subseteq \mathsf{derive}(\mathcal{A})$. □

Lemma 17. *If $\mathsf{d}(i) \subseteq \mathcal{P}(Q) \times F$ and none of the rules (1) – (3) can be applied to $\mathsf{d}(i)$, then for all $t \in \mathcal{T}(F)$, $(\mathsf{reach}_{\mathcal{A}}(t), \mathrm{root}(t)) \in \mathsf{d}(i)$.*

Proof. We prove this by noetherian induction on the subterm relation. Specifically, we try to prove that $(\mathsf{reach}_{\mathcal{A}}(t), \mathrm{root}(t)) \in \mathsf{d}(i)$ assuming that if $s \in \mathcal{T}(F)$ is a subterm of t, then $(\mathsf{reach}_{\mathcal{A}}(s), \mathrm{root}(s)) \in \mathsf{d}(i)$. By the restrictions placed on the axioms of $\mathcal{E}$, $\mathrm{root}(t)$ must be either a free symbol, an associative symbol, or an associative-commutative symbol. We consider each of these possibilities separately.

If $\mathrm{root}(t) \in F$ is free, we can assume that t is of the form $f(t_1, \ldots, t_n)$. Observe that if f is a constant, then $n = 0$. By induction $(\mathsf{reach}_{\mathcal{A}}(t_j), \mathrm{root}(t_j)) \in \mathsf{d}(i)$ for all $j \leq n$. This would suggest that we could apply rule (1) using $(\mathsf{reach}_{\mathcal{A}}(t_j), \mathrm{root}(t_j)$ in place of the pair (P_j, f_j) appearing at the top of the rule if $(\mathsf{reach}_{\mathcal{A}}(f(P_1, \ldots, P_n)), f) \notin \mathsf{d}(i)$. However, by assumption the rule cannot be applied, and by Lemma 7, $\mathsf{reach}_{\mathcal{A}}(f(P_1, \ldots, P_n)) = \mathsf{reach}_{\mathcal{A}}(f(t_1, \ldots, t_n))$. As $t = f(t_1, \ldots, t_n)$, we have that $(\mathsf{reach}_{\mathcal{A}}(t), \mathrm{root}(t)) \in \mathsf{d}(i)$.

If $\mathrm{root}(t)$ is associative and not commutative, then we can assume that t is of the form $C[t_1, \ldots, t_n]$ where $C \in T(\{f, \Box_1, \ldots, \Box_n\})$ is a maximal f-context only containing the associative symbol $\mathrm{root}(t)$ with n holes, and $\mathrm{root}(t_i) \neq \mathrm{root}(t)$ for all $i \leq n$. Let $f = \mathrm{root}(t)$, and let $P = \mathsf{reach}_{\mathcal{A}}(C[t_1, \ldots, t_n])$. By our induction hypothesis, $(\mathsf{reach}_{\mathcal{A}}(t_i), \mathrm{root}(t_i)) \in \mathsf{d}(i)$ for each $i \leq n$, and therefore $\mathsf{reach}_{\mathcal{A}}(t_i) \in \Sigma_{f,\mathsf{d}(i)}$ (since $\mathrm{root}(t_i) \neq f$). By Cor. 1, $\mathsf{reach}_{\mathcal{A}}(t_1), \ldots, \mathsf{reach}_{\mathcal{A}}(t_n) \in \mathcal{L}(G_{f,\mathsf{d}(i)}(P))$. Since $\mathrm{root}(C[t_1, \ldots, t_n]) = f$, $C \neq \Box$, and $n \geq 2$. Thus,

$$\mathsf{reach}_{\mathcal{A}}(t_1), \ldots, \mathsf{reach}_{\mathcal{A}}(t_n) \in \Sigma^{2+}_{f,\mathsf{d}(i)} \cap \mathcal{L}(G_{f,\mathsf{d}(i)}(P)).$$

This would imply that rule (2) could be applied to form $\mathsf{d}(i+1) = \mathsf{d}(i) \uplus \{ (P, f) \}$ if $(P, f) \notin \mathsf{d}(i)$. Since by assumption the rule cannot be applied, $(P, f) \in \mathsf{d}(i)$.

Finally, if $\mathrm{root}(t)$ is associative and commutative, then we can assume that t is of the form $C[t_1, \ldots, t_n]$ where $C \in T(\{f, \Box_1, \ldots, \Box_n\})$ is a maximal context only containing the AC symbol $\mathrm{root}(t)$ with n holes, and $\mathrm{root}(t_i) \neq \mathrm{root}(t)$ for all $i \leq$

n. Let $f = \mathrm{root}(t)$, and let $P = \mathsf{reach}_{\mathcal{A}}(C[t_1, \ldots, t_n])$. By our induction hypothesis, $(\mathsf{reach}_{\mathcal{A}}(t_i), \mathrm{root}(t_i)) \in \mathsf{d}(i)$ for each $i \leq n$, and therefore $\mathsf{reach}_{\mathcal{A}}(t_i) \in \Sigma_{f,\mathsf{d}(i)}$ (since $\mathrm{root}(t_i) \neq f$). By Cor. 2, $\#(\mathsf{reach}_{\mathcal{A}}(t_1), \ldots, \mathsf{reach}_{\mathcal{A}}(t_n)) \in \mathcal{S}(G_{f,\mathsf{d}(i)}(P))$. Since $\mathrm{root}(C[t_1, \ldots, t_n]) = f$, $C \neq \Box$, and $n \geq 2$. Thus,

$$\#(\mathsf{reach}_{\mathcal{A}}(t_1), \ldots, \mathsf{reach}_{\mathcal{A}}(t_n)) \in \mathbb{N}^{>\mathbf{1}} \cap \mathcal{S}(G_{f,\mathsf{d}(i)}(P)).$$

This would imply that rule (3) could be applied to form $\mathsf{d}(i+1) = \mathsf{d}(i) \uplus \{\, (P, f) \,\}$ if $(P, f) \notin \mathsf{d}(i)$. Since by assumption no rule can be applied, $(P, f) \in \mathsf{d}(i)$.

Finally we can conclude with the proof of Theorem 2:

Theorem 2. *Let $\mathcal{A} = (\mathcal{E}, Q, \phi, \Delta)$ be a PTA with $\mathcal{E} = (F, E)$ containing only associativity and commutativity axioms (*A∪C*-theory). Every chain $\mathsf{d}(0), \mathsf{d}(1), \ldots$ obtained by applying the rules (1)–(3) until completion satisfies the following properties:*

- *the length k of the chain is $|\mathsf{derive}(\mathcal{A})|$, and*
- $\mathsf{d}(k) = \mathsf{derive}(\mathcal{A})$.

Proof. Since $|\mathcal{P}(Q) \times F|$ is finite and each application of a rule (1) – (3) to a set $\mathsf{d}(i)$ results in a set $\mathsf{d}(i+1)$ with one additional element, any chain $\mathsf{d}(0)\,\mathsf{d}(1)\,\mathsf{d}(2)\ldots$ obtained by applying the rules until completion must be finite. Let $\mathsf{d}(0)\,\mathsf{d}(1)\ldots\mathsf{d}(k)$ be a chain resulting from applying the rules until termination. As $\mathsf{d}(0) = \varnothing$, $\mathsf{d}(0) \subseteq \mathsf{derive}(\mathcal{A})$. By Lemma 16, $\mathsf{d}(i) \subseteq \mathsf{derive}(\mathcal{A})$ implies $\mathsf{d}(i+1) \subseteq \mathsf{derive}(\mathcal{A})$. Therefore by induction on i, $\mathsf{d}(i) \subseteq \mathsf{derive}(\mathcal{A})$ for all $i \leq k$. In particular, $\mathsf{d}(k) \subseteq \mathsf{derive}(\mathcal{A})$. However as no rule can be applied to $\mathsf{d}(k)$, Lemma 17 implies that $\mathsf{derive}(\mathcal{A}) \subseteq \mathsf{d}(k)$. Thus $\mathsf{d}(k) = \mathsf{derive}(\mathcal{A})$. Moreover, since $d(0) = \varnothing$ and each step in the chain $\mathsf{d}(i)$ to $\mathsf{d}(i+1)$ adds a single new element, $k = |\mathsf{d}(k)| = |\mathsf{derive}(\mathcal{A})|$. □

A.5 Proof of Theorem 5

It is easy to verify that the procedure terminates and that the pair returned by each return statement is indeed a counterexample. The non-trivial part of this theorem is that if the outer loop terminates without returning a pair, check_equiv should return true. This property is obtained by showing that if $\mathcal{L}(\mathcal{M}_\alpha) \neq \mathcal{L}(\mathcal{G}(\alpha))$ for some $\alpha \in Q$ executed by the outer loop, then the body of the loop is guaranteed to return a pair.

The string $u \in \Sigma^*$ chosen in the body is in the symmetric difference of $\mathcal{L}(\mathcal{M}_\alpha)$ and $P_\alpha\theta$. If $u \in \mathcal{L}(\mathcal{M}_\alpha) \oplus \mathcal{L}(\mathcal{G}(\alpha)$ (or vice versa), then the body returns (α, u). Otherwise, if $u \in \mathcal{L}(\mathcal{M}_\alpha) \iff u \in \mathcal{L}(\mathcal{G}(\alpha))$, then then $u \in P_\alpha\theta \oplus \mathcal{L}(\mathcal{G}(\alpha))$.

Let ψ be the substitution $\alpha \mapsto \mathcal{L}(G(\alpha))$. Since ψ is a solution to the equations generated by G, $\mathcal{L}(G(\alpha)) = \psi(\alpha) = P_\alpha\theta$. So $u \in P_\alpha\theta \oplus P_\alpha\psi$. We will show that the inner for loop must return a value when $u \in P_\alpha\theta - P_\alpha\psi$ — the proof in the other case when $u \in P_\alpha\psi - P_\alpha\theta$ is similar.

If the rules in G whose left-hand-side is α are $\alpha := \beta_1\gamma_1, \ldots, \alpha := \beta_n\gamma_n$, then P_α is of the form $P_\alpha = \beta_1\gamma_1 \mid \cdots \mid \beta_n\gamma_n$. So $u \in P_\alpha\theta$ implies that $u \in (\beta_i\gamma_i)\theta$ for

some i. Likewise, as $u \notin P_\alpha\psi$ and $\beta_i\gamma_i\psi \subseteq P_\alpha\psi$, it easily follows that $u \notin (\beta_i\gamma_i)\psi$. Since $u \in (\beta_i\gamma_i)\theta$, we can partition it into strings $s, t \in \Sigma^*$ such that $u = st$, $s \in \theta(\beta_i)$, and $t \in \theta(\gamma_i)$. In addition, since $u = st$ and $u \notin (\beta_i\gamma_i)\psi$, either $s \notin \psi(\beta_i)$ or $t \notin \psi(\gamma_i)$. Thus by the definition of ψ, there is a rule $\alpha := \beta_i\gamma_i$ in P and strings $s, t \in \Sigma^*$ such that $u := st$ and either $s \in \mathcal{L}(\mathcal{M}_{\beta_i}) - \mathcal{L}(G(\beta_i))$ or $t \in \mathcal{L}(\mathcal{M}_{\gamma_i}) - \mathcal{L}(G(\gamma_i))$. A similar argument in this case where $u \in P_\alpha\psi - P_\alpha\theta$ shows that the inner loop will always return a pair when executed.