Alternating Two-Way AC-Tree Automata

Jean Goubault-Larrecq, Kumar Neeraj Verma

J. Goubault–Larrecq and K. N. Verma

Alternating Two–Way AC–Tree Automata

Research Report LSV–02–11, Sep. 2002

Laboratoire Spécification et Vérification

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

CENTRE NATIONAL DE LA RECHERCHE SCIENTIFIQUE

http://www.lsv.ens–cachan.fr/Publis/
Research Report LSV–02–11, Lab. Spécification et Vérification, CNRS & ENS de Cachan, France, Sep. 2002

Alternating Two-Way AC-Tree Automata*

Jean Goubault-Larrecq Kumar Neeraj Verma

LSV/CNRS UMR 8643 & ENS Cachan, 61 avenue du président-Wilson,
F-94235 Cachan Cedex {goubault|verma}@lsv.ens-cachan.fr
Phone: +33-1 47 40 75 68 Fax: +33-1 47 40 24 64

Abstract. We explore the notion of alternating two-way tree automata modulo the theory of finitely many associative-commutative (AC) symbols, some of them with a unit (AC1). This was prompted by questions arising in cryptographic protocol verification, in particular in modeling group key agreement schemes based on Diffie-Hellman-like functions, where the emptiness question for intersections of such automata is fundamental. We show that the use of conditional push clauses, or of alternation, leads to undecidability, already in the case of one AC or AC1 symbol, with only functions of arity zero. On the other hand, emptiness is decidable in the general case of many function symbols, including many AC or AC1 symbols, provided push clauses are unconditional and intersection clauses are final. To this end, extensive use of refinements of resolution is made.

1 Introduction

Automata and in particular tree automata are ubiquitous in verification [9]. One particular area where they, or extensions thereof, have proved useful is cryptographic protocol verification [33, 19, 22, 8], where they are used to represent inter alia the infinite sets of messages that a malicious Dolev-Yao intruder [13] may build by repeatedly decrypting and encrypting from past traffic. (We give a more detailed example of this later.) For now, let us say that messages in such models are represented as first-order terms (a.k.a., *trees*): if M and K are messages, then the term $c(M, K)$ denotes the result of encrypting M with key K. The equational theory of first-order terms is adequate for representing *ideal* cryptographic primitives, where $c(M, K) = c(M', K')$ only if $M = M'$ and $K = K'$ and where $c(c(\ldots c(M, K_1)\ldots), K_n) \neq M$ for any $n \geq 1$, in particular.

It is sometimes interesting to have a richer set of cryptographic primitives that obey additional algebraic laws. For example, it has been proposed [21] to use so-called *commutative hash functions* to implement dynamic authenticated dictionaries. Simply put, commutative hash functions are binary functions h that are associative and commutative (*AC*), such that anybody can build $h(M, N)$ from M and N, but noone can infer either M or N from $h(M, N)$ only. Another example comes from Diffie-Hellman key agreement [12], where principal A_1 chooses a secret M_1 and sends $e(M_1)$ to A_2 (where e is a unary function symbol denoting some way of encapsulating the secret), A_2 chooses another secret M_2 and sends $e(M_2)$ to A_1, and both compute $e(M_1 \oplus M_2)$ as their common key. It is required that any participant, whether honest or dishonest, can compute

* Partially supported by the ACI VERNAM, the RNTL project EVA and the ACI jeunes chercheurs "Sécurité informatique, protocoles cryptographiques et détection d'intrusions".

$e(M)$ from M, and $e(M \oplus M')$ from $e(M)$ and M', but no more. Concretely, $e(M)$ is implemented as $\alpha^M \bmod N$ for well-chosen numbers α and N, and $\oplus$ as multiplication: to get $e(M \oplus M') = \alpha^{M \cdot M'} \bmod N$ from $e(M) = \alpha^M \bmod N$, just raise the latter to M' mod N. Note that $\oplus$ has to be commutative for this to succeed. In general, extensions of this scheme to the important *group key agreement* problem, where a group of principals has to agree on a common key, unavailable to external eavesdroppers, and not created by any single principal in the group, as in the CLIQUES protocol suite [41], require $\oplus$ to be both associative and commutative; multiplication mod N certainly is.

If we are to extend the tree-automata based approaches to cryptographic protocol verification to such AC operators, we need suitably modified notions of automata recognizing terms modulo the theory of associativity and commutativity of $\oplus$: this is the topic of this paper. In addition, we shall see in Section 3 that cryptographic protocols are better represented through *two-way* tree automata, where transitions can not only construct but also *destruct* terms. We shall also see that the main security questions reduce to deciding emptiness of (intersections of) such automata.

Independently of cryptographic protocols, two-way automata modulo equational theories $\mathcal{E}$ are interesting per se; the cases $\mathcal{E} = \mathrm{A}$ and $\mathcal{E} = \mathrm{AC1}$ (AC with unit element) occur often, see below. We shall also investigate the *alternating* variants, as well as the cases with *several* AC or AC1 symbols, which are natural extensions.

Our import is a classification of alternating, two-way AC-tree automata, relative to the decidability of the emptiness, and in general the *intersection-emptiness* question: given finitely many alternating two-way AC-automata, is the intersection of their languages empty? We shall see that emptiness of alternating, two-way AC-automata is undecidable in general, but intersection-emptiness of two-way AC-automata with no alternation and only so-called *standard* push clauses is decidable. The decision algorithm is highly non-trivial, and rests partly on proof-theoretic tools.

The paper is organized as follows. In Section 2, we introduce a general clausal format for equational, alternating, two-way automata, possibly with equality constraints between brothers, and even a form of set constraints. Then, ordered resolution with splitting provides a streamlined decision procedure for the emptiness of such automata and the satisfiability of set constraints, in the non-equational case. This must have been known to a number of researchers, but we are not aware of this result in print. The rest of the paper deals with automata *modulo AC*, i.e., modulo the theory of finitely many AC symbols. We illustrate the relevance of this theory to the verification of cryptographic protocols in Section 3, using the example of group key agreement. This does not use alternation: this is fortunate, as emptiness is undecidable in the presence of alternation (Section 4), even with *constant-only* signatures, and just one AC symbol $+$. We show that the constant-only cases not covered by the results of Section 4 have decidable intersection-emptiness problems in Section 5; in fact, the resulting AC-automata recognize exactly the semi-linear sets. Leaving the constant-only case, we show that the intersection-emptiness of AC-automata on several free, AC and AC1 symbols is decidable in Section 6, provided the equational push clauses are *non-conditional*. In particular, this covers the case of the group key agreement example of Section 3.

Related Work. There is much literature on finite tree automata [9, 17]. Apart from the already cited use in cryptographic protocol verification, applications include approximat-

ing reachability sets for rewrite systems [18], disunification and inductive reducibility [30], unification under constraints [26], ground reducibility [10], automated inductive theorem proving [5], fast tree matching [28], automated model building in first-order logic [37], etc. These applications deal with automata on *finite* trees. We won't deal with automata on infinite trees [42], which are also fundamental, e.g. in temporal and program logics [14].

Two-way automata, a.k.a. *pushdown processes*, where transitions may not only construct but also destruct terms, are also classical. The relation with certain Horn sets was pioneered in [16], and refined in e.g., [7]. *Cartesian approximation* is the key to define upper approximations of various sets of ground atoms, e.g., *success sets*. This can be adapted to the AC case, yielding formulae in the decidable class of Section 6; for lack of space, details are left to the reader. Do not confuse pushdown processes with pushdown *automata* [39], which recognize the strictly larger class of context-free tree languages.

The idea of generalizing tree automata to recognize languages of terms *modulo an equational theory* is then natural, and a canonical choice of theory is that of one associative-commutative (AC) symbol $+$. This has been explored a number of times, e.g., [11, 34, 29, 35]. While not all notions of AC-automata coincide, there is always a common core. For example, the automata of [29] have additional sort restrictions, but are also extended with a rich constraints language; if we forget about the latter, and dismiss the sort restrictions, we get exactly the languages recognized by the non-alternating, non-two-way subclass of our automata. This subclass also coincides with the regular AC-automata of [35]. In general, it is easy to check that in the non-alternating, non-two-way case, our automata modulo $\mathcal{E}$ are exactly the $\mathcal{E}$-closures of the languages recognized by the same automaton modulo the empty theory; this is not the case in [35], except for regular $\mathcal{E}$-automata on a linear theory $\mathcal{E}$. The paper [29] establishes the standard properties of closure under intersection, union, complementation, and projection of the languages considered therein. For our two-way AC-automata, we only establish these closure properties in the constant-only subcase. In the general case, only the case of unions is trivial, and we concentrate on the difficult intersection-emptiness problem, leaving closure properties to future work.

As a final note, we shall make extensive use of resolution theorem proving techniques. A comprehensive reference is the handbook [38]. Using resolution techniques to decide subclasses of first-order logic formulas was pioneered by Joyner [25], and earlier by Maslov, see [15]. Standard refinements of resolution used in this area are hyperresolution and ordered refinements. Note that our decision algorithm of Section 6 is not based on resolution only. In this respect, this work is similar to early work such as [27], where resolution plus a number of other rules are used to decide the Gödel class.

Acknowledgments Thanks to H. Comon and L. Fribourg for many stimulating discussions, to A. Finkel, S. Lasota, and to the anonymous referees at LICS and CSL for helpful comments.

2 Alternating Two-Way $\mathcal{E}$-Automata

Fix a signature Σ of function symbols, each coming with a fixed arity, and let $\mathcal{E}$ be an equational theory. Unless told otherwise, in this paper we assume that Σ contains

finitely many binary symbols $+_1, \ldots, +_p$ and $\mathcal{E}$ is the theory stating that every $+_i$, $1 \leq i \leq p$, is associative and commutative (AC). A trivial case is when $\mathcal{E}$ is the empty theory; in this case we shall retrieve the standard notion of alternating two-way tree automata. We shall also be interested in the case where some symbols $+_i$ also have a unit 0_i; we say that $+_i$ is an AC1 symbol, then.

A (non-deterministic) *$\mathcal{E}$-tree automaton* $\mathcal{A}$ is a finite set of clauses of the form:
$$P(f(x_1, \ldots, x_n)) \Leftarrow P_1(x_1), \ldots, P_n(x_n) \quad (1)$$
where $A \Leftarrow B_1, \ldots, B_n$ is an implication "B_1 and $\ldots$ B_n imply A". Clauses (1) are called *pop clauses*, or ordinary tree automata transitions. Intuitively, this reads as "if x_1 is recognized at state P_1, and $\ldots$, and x_n is recognized at state P_n, then $f(x_1, \ldots, x_n)$ is recognized at state P". For a more detailed discussion why this really encodes automata, see e.g. [23].Here we shall always assume that the variables $x_1, \ldots, x_n$ are distinct; otherwise we get tree automata with equality constraints between brothers [4], which pose no problem in the non-equational case but would definitely in the AC case.

An *alternating $\mathcal{E}$-tree automaton* in addition has *intersection clauses* of the form:
$$P(x) \Leftarrow P_1(x), \ldots, P_n(x) \quad (2)$$
When $n = 1$, this is called an *ϵ-clause*.

A *two-way* automaton may also include so-called *push clauses* of the form:
$$P_i(x_i) \Leftarrow P(f(x_1, \ldots, x_n)), P_{i_1}(x_{i_1}), \ldots, P_{i_k}(x_{i_k}) \quad (3)$$
where $1 \leq i \leq n, 1 \leq i_1 < \ldots < i_k \leq n$. This intuitively means "if $f(x_1, \ldots, x_n)$ is recognized at P, and x_{i_1} at P_{i_1}, and $\ldots$ and x_{i_k} at P_{i_k}, then x_i is recognized at P_i". If $k = 0$, call this a *standard push clause*; if $k \neq 0$, call this a *conditional* push clause.

People familiar with classical automata theory tend to be puzzled by this definition of automata, and in particular by the fact that no definition of a *run* of a term against an automaton is given; we invite the puzzled reader to check that *positive hyper-resolution* derivations [6] (which are also unit derivations in the case of Horn clauses) are exactly *bottom-up runs* [17]: for every ground term $P(t)$, the positive hyper-resolution derivations of the unit clause $P(t)$ are exactly the runs of t that abut to state P, against the given tree automaton, considered bottom-up. On the other hand, *negative hyper-resolution* derivations are exactly the top-down runs. The theory of resolution theorem proving enables us to replace any complete deduction procedure (positive, negative hyper-resolution) by any other complete procedure; it seems that ordered resolution is the most powerful refinement of resolution in many practical cases.

A Tarskian *interpretation* modulo $\mathcal{E}$ is a tuple $(D, (I_f)_{f \text{ function symbol}}, (I_P)_{P \text{ predicate symbol}})$ where the *domain* D of I is a non-empty set, for every n-ary function symbol f, I_f is a function from D^n to D, and for every (unary) predicate symbol P, I_P is a subset of D. The interpretation of a term t in an environment ρ mapping variables to elements of D is $I[\![x]\!]\rho \hat{=} \rho(x)$, $I[\![f(t_1, \ldots, t_n)]\!]\rho \hat{=} I_f(I[\![t_1]\!]\rho, \ldots, I[\![t_n]\!]\rho)$. It is required that if s and t are equal modulo $\mathcal{E}$, then for every predicate P, for every ρ, $I[\![s]\!]\rho \in I_P$ if and only if $I[\![t]\!]\rho \in I_P$. The relation $I, \rho \models F$ is defined on atoms and literals F by $I, \rho \models P(t)$ iff $I[\![t]\!]\rho \in I_P$, $I, \rho \models -P(t)$ iff $I[\![t]\!]\rho \notin I_P$; let $I \models C$, where C is a clause, iff for every ρ, there is some literal L in C such that $I, \rho \models L$; let $I \models S$, where S is a set of clauses, iff $I \models C$ for every $C \in S$. S is *unsatisfiable* iff $I \models S$ for no Tarskian interpretation S. We write $S \models C$ to denote the condition "for every Tarskian interpretation I, if $I \models S$ then $I \models C$".

A *Herbrand interpretation* mod $\mathcal{E}$ is a set of equivalence classes mod $\mathcal{E}$ of ground atoms built on Σ. This is a special case of Tarskian interpretation, where D is fixed to be the set of all ground terms mod $\mathcal{E}$, and I_f maps $(t_1, \ldots, t_n)$ to the term $f(t_1, \ldots, t_n)$: then giving a collection of sets I_P of terms mod $\mathcal{E}$ for each P (a Tarskian interpretation) is equivalent to giving directly the set of all ground atoms mod $\mathcal{E}$ that are true in I (the Herbrand interpretation).

A *Herbrand model* mod $\mathcal{E}$ of a set S of clauses is a Herbrand model of every $C \in S$: a Herbrand interpretation I such that all ground instances of C contain some atom in I or some negation of an atom not in I. Every satisfiable Horn clause set, in particular every automaton, has a least Herbrand model mod $\mathcal{E}$: by standard arguments, any intersection of Herbrand models is indeed still a Herbrand model mod $\mathcal{E}$.

The set of terms (mod $\mathcal{E}$) *recognized* at *state* P in $\mathcal{A}$, a.k.a. the *language* $L_P(\mathcal{A})$ of $\mathcal{A}$ at P, is the set of all terms t such that $P(t)$ is in the least Herbrand model of $\mathcal{A}$. We say P is *empty* in $\mathcal{A}$ if and only if $L_P(\mathcal{A})$ is empty, and similarly for other properties. It is not hard to see that P is empty in $\mathcal{A}$ if and only if the set of clauses $\mathcal{A}$ plus the *query* clause $\bot \Leftarrow P(x)$ is satisfiable, where $\bot$ denotes false. Indeed, if P is empty then the least Herbrand model of $\mathcal{A}$ does not contain any ground atom of the form $P(t)$, hence makes $\bot \Leftarrow P(x)$ true. Conversely, if $\mathcal{A}$ plus $\bot \Leftarrow P(x)$ is satisfiable, then its least Herbrand model does not contain any ground atom of the form $P(t)$. Since every model of $\mathcal{A}$ plus $\bot \Leftarrow P(x)$ is also a model of $\mathcal{A}$, the least Herbrand model of $\mathcal{A}$ is included in that of $\mathcal{A}$ plus $\bot \Leftarrow P(x)$, hence does not contain any ground atom of the form $P(t)$ either; so P is empty in $\mathcal{A}$.

More generally, we may consider non-Horn clauses (we won't do this here). Call a *block* any clause of the form:

$$\bigvee_{i=1}^{m} \pm_i P_i(x) \tag{4}$$

where the signs $\pm_i$ are either $+$ or $-$ (negation). Note that a block has at most one free variable. Write e.g., $B(x)$ for a block with free variable x. Our clauses will be of two forms: *simple clauses* are blocks (4), and *complex clauses* are of the form:

$$\bigvee_{i=1}^{m} \pm_i P_i(f(x_1, \ldots, x_n)) \vee \bigvee_{j=1}^{n} B_j(x_j) \tag{5}$$

where $B_1(x_1), \ldots, B_n(x_n)$ are blocks, and $m \geq 1$. Note that f and the variables x_1, $\ldots$, x_n are the same for each i in (5), and that all the free variables of the clause occur in $f(x_1, \ldots, x_n)$. Complex clauses include pop clauses and push clauses alike; simple clauses include intersection clauses and query clauses $\bot \Leftarrow P(x)$.

It is arguably fair to call clauses (4) or (5) positive *set constraints* [3]. Indeed, in the empty theory, the elementary set constraints:

$$\begin{array}{cc} (a)\ X \subseteq Y & \\ (b)\ X \subseteq Y \cup Z & (c)\ X \cap Y \subseteq Z \\ (d)\ X \subseteq \complement Y & (e)\ \complement X \subseteq Y \\ (f)\ X \subseteq f(X_1, \ldots, X_n) & (g)\ f(X_1, \ldots, X_n) \subseteq X \\ (h)\ f_i^{-1}(X) \subseteq Y & \end{array}$$

where X, Y, Z, $\ldots$, are variables denoting unknown sets of terms, are notations for the respective clauses:

$$(a)\ \ -X(x) \vee +Y(x)$$

$$
\begin{array}{ll}
(b) & -X(x) \vee +Y(x) \vee +Z(x) \\
(c) & -X(x) \vee -Y(x) \vee +Z(x) \\
(d) & -X(x) \vee -Y(x) \\
(e) & +X(x) \vee +Y(x) \\
(f) & \begin{cases} -X(f(x_1, \ldots, x_n)) \vee +X_1(x_1) \\ \ldots \\ -X(f(x_1, \ldots, x_n)) \vee +X_n(x_n) \\ -X(g(x_1, \ldots, x_m)) \quad \text{(for all } g \neq f\text{)} \end{cases} \\
(g) & \bigvee_{i=1}^{n} -X_i(x_i) \vee +X(f(x_1, \ldots, x_n)) \\
(h) & -X(f(x_1, \ldots, x_n)) \vee +Y(x_i)
\end{array}
$$

Ordered resolution with (eager) splitting [15] terminates on clauses arising from alternating two-way automata (modulo the empty theory), and more generally on clauses of the form (4) and (5) , where the ordering is taken to be the size of atoms (exercise). In general, any refinement of resolution that is complete via semantic trees (in the sense of [25]) is still complete with eager splitting and subsumption. A clause C *subsumes* the clause D iff $C\sigma \subseteq D$ for some substitution σ (read the inclusion mod $\mathcal{E}$).

Splitting is a tableau rule. A *tableau* is a tree whose nodes are clause sets. Resolution steps extend a branch by adding below the node S a new node $S \cup \{C\}$, where C is a resolvent of some clauses in S. Splitting applies when S contains a clause $C_1 \vee \ldots \vee C_k$, where the subclauses C_i share no free variable, and produces k subbranches where this clause is replaced by $C_i, 1 \leq i \leq k$. (Think of a tableau as a *disjunction* of branches.) This is needed, e.g., resolving push and pop clauses on $P(f(x_1, \ldots, x_n))$ yields a splittable clause in general. This is needed because ordered resolvents of simple and complex clauses sometimes produce disjunctions of simple clauses, which can then be split into simple clauses again. So ordered resolution with eager splitting only produces simple and complex clauses again. This terminates because there are only finitely many of them—in fact, in non-deterministic (because of splitting) exponential time.

It is also easy to check that clauses (4) and (5) are exactly what is needed to write a definitional clausal form [2] of skolemized formulas from the monadic class [1]. This gives another proof that the monadic class is decidable, similar to [25]. In particular, this decides positive set constraints, as noticed by [3]. Using splitting is more efficient in practice than Joyner's condensing rule [25].

The same technique can be used, at least in principle, to deal with sets of clauses of the form (4) and (5) mod $\mathcal{E}$. For example, this works for positive set constraints on sets of normalizing λ-terms mod $\beta\eta$-conversion, yielding a non-deterministic doubly exponential-time algorithm for satisfiability [23]. It turns out that the situation with AC-set constraints, and even alternating two-way AC-automata is more complex, and rather different. In particular, in both the empty theory and the $\beta\eta$ theory cases, ordered resolution terminates, because only finitely many clauses can be generated: this yields decidability. Our argument in the AC case will be more involved (Section 6).

3 An Application in Cryptographic Protocol Verification

We give an example in the field of group key agreement schemes. To keep the exposition short, we only mention salient features exhibiting the role of AC tree-automata.

Consider the initial key agreement protocol IKA.1 [41] (formerly known as GDH.2), used to create an initial group key in the CLIQUES protocol suite. This works as follows; remember that we have a cryptographic hash function e, and an AC operation $\oplus$ with unit $\mathbf{0}$, typically implemented by $e(M) = \alpha^M \bmod N$, $\oplus$ being multiplication and $\mathbf{0}$ being 1. We also use a binary function $cons$, a constant nil to represent lists, and abbreviate $cons(M_1, cons(M_2, \ldots, cons(M_n, nil)\ldots))$ as $M_1; M_2; \ldots, M_n$. For simplicity, assume we have 3 members in the group, $\mathcal{M}_1, \mathcal{M}_2, \mathcal{M}_3$.

First, IKA.1 starts with an so-called *upflow* phase: $\mathcal{M}_1$ sends $\mathcal{M}_2$ the pair $e(\mathbf{0}); e(N_1)$, where N_1 is a fresh nonce; N_1 is modeled, as usual [33], as a new constant. Then $\mathcal{M}_2$ sends $e(N_2); e(N_1); e(N_1 \oplus N_2)$ to $\mathcal{M}_3$, where N_2 is another fresh nonce (modeled as another new constant $N_2 \neq N_1$). This is possible due to our assumptions that anybody can build $e(M)$ from M and $e(M \oplus M')$ from $e(M)$ and M'.

Once this is done, $\mathcal{M}_3$ starts the *downflow* phase, and broadcasts $e(N_2 \oplus N_3); e(N_1 \oplus N_3)$, from which all members can compute the group key $e(N_1 \oplus N_2 \oplus N_3)$. ($N_3$ is a third fresh nonce created by $\mathcal{M}_3$.)

All possible interleaved executions of the protocol can be described using Horn clauses mod AC1, and we claim that the resulting set of clauses is a two-way AC1-automaton. (The difference between AC and AC1 is inessential, as we can encode one into the other, see Corollary 4.) Let us write selected clauses from this set.

To model communication, take the standard Dolev-Yao approach [13]: every message sent is received by the intruder, every message received is from the intruder. For every configuration C reachable in an interleaved run of $\mathcal{M}_1, \mathcal{M}_2, \mathcal{M}_3$, create a fresh unary predicate symbol k_C, such that $k_C(M)$ holds if and only if the intruder can deduce M from the set of messages it has got from analyzing all communications before C was reached. Depending on the actual intruder model, one or several of the following clauses have to be written for each configuration C:

$$k_C(e(\mathbf{0})) \qquad \text{Intruder knows } e(\mathbf{0}) \tag{6}$$

$$k_C(e(x \oplus y)) \Leftarrow k_C(e(x)), k_C(y) \qquad \text{Intruder can exponentiate} \tag{7}$$

$$k_C(nil) \qquad \text{Intruder knows the empty list} \tag{8}$$

$$k_C(cons(x, y)) \Leftarrow k_C(x), k_C(y) \qquad \text{Intruder can build lists} \tag{9}$$

$$k_C(x) \Leftarrow k_C(cons(x, y)) \qquad \text{Intruder can read heads} \tag{10}$$

$$k_C(y) \Leftarrow k_C(cons(x, y)) \qquad \text{Intruder can read tails} \tag{11}$$

Starting from an initial configuration C_0, for which we assume that some predicate k_{C_0} has been defined—by some AC1-tree automaton—, execution may proceed by letting $\mathcal{M}_1$ send its upflow message to $\mathcal{M}_2$, letting the whole system progress to some new configuration C_1:

$$k_{C_1}(e(\mathbf{0}); e(N_1)) \qquad \text{Intruder gets } \mathcal{M}_1\text{'s message} \tag{12}$$

$$k_{C_1}(x) \Leftarrow k_{C_0}(x) \qquad \text{Intruder remembers past messages} \tag{13}$$

Again, (13) is optional. Including it means the intruder will be able to replay old messages. Excluding it as well as clauses (6)–(11) (with $C = C_1$) means the intruder acts as

a mere eavesdropper: if we make sure that no clause is given with head k_{C_0} (no message is known to the intruder initially), this is the *pure eavesdropper model*, where for every C, k_C holds of at most one message, which is the last message sent by some honest principal—a one-place buffer. Including all clauses (13) and (6)–(11) (with $C = C_1$) is the standard choice in Dolev-Yao models, where the intruder can replay, decompose and build messages at will.

Let us write what happens when the next action is $\mathcal{M}_2$ sending its own message to $\mathcal{M}_3$:

$$k_{C_2}(e(N_2); e(y); e(y \oplus N_2)) \Leftarrow k_{C_1}(x; e(y)) \quad (14)$$

(with possibly an extra clause $k_{C_2}(x) \Leftarrow k_{C_1}(x)$ if we wish to state that the intruder remembers past messages.) In other words, $\mathcal{M}_2$ reads the message from $\mathcal{M}_1$ by querying the intruder through k_{C_1}, then builds $e(N_2); e(y); e(y \oplus N_2)$, which it sends the intruder in the new configuration C_2.

The downflow message from $\mathcal{M}_3$ gives rise to the clause:

$$k_{C_3}(e(x \oplus N_3); e(y \oplus N_3)) \Leftarrow k_{C_2}(e(x); e(y); e(z)) \quad (15)$$

Now the secrecy requirement on, say, $\mathcal{M}_1$'s view of the group key is that

$$\bot \Leftarrow k_{C_3}(e(x); y), k_{C_1}(e(x \oplus N_1)), k_{C_2}(e(x \oplus N_1)), k_{C_3}(e(x \oplus N_1)) \quad (16)$$

Indeed, $\mathcal{M}_1$'s view of the group key is $e(x \oplus N_1)$, where the message broadcasted by $\mathcal{M}_3$ is $e(x); y$, i.e., where $k_{C_3}(e(x); y)$ holds (reminder: if this message is not forged, then $x = N_2 \oplus N_3$). Clause (16) states that this view $e(x \oplus N_1)$ was not known to the intruder in any configuration C_1, C_2, or C_3.

There are many other possible interleavings, which we do not mention. As we have said, our purpose here is not to actually verify this protocol, but to illustrate the application of AC and AC1-tree automata on a concrete example. All clauses but a few are automata clauses. E.g., clauses (8), (9) are pop clauses, clauses (10) and (11) are standard push clauses. Clause (13) is an ϵ-clause. The remaining ones can be rewritten in the form of automata clauses by introducing auxiliary states, a.k.a., predicate symbols. For example, (6) can be rewritten as two pop clauses, $k_C(e(x)) \Leftarrow P_C(x)$ and $P_C(\mathbf{0})$, where P_C is fresh. Clause (7) can be rewritten with both pop and push clauses:

$$Q_C(x) \Leftarrow k_C(e(x)) \qquad Q'_C(x \oplus y) \Leftarrow Q_C(x), k_C(y) \qquad k_C(e(z)) \Leftarrow Q'_C(z)$$

where Q_C and Q'_C are fresh. In general, any clause with linear left-hand and right-hand sides can be rewritten this way. The only clauses that cannot are (14) and (16). We can rewrite the latter as

$$R(z) \Leftarrow k_{C_3}(\mathit{cons}(z, z')) \qquad R'(x) \Leftarrow R(e(x)) \qquad R''(x \oplus N_1) \Leftarrow R'(x)$$
$$R'''(e(z)) \Leftarrow R''(z) \qquad \bot \Leftarrow R'''(z), k_{C_1}(z), k_{C_2}(z), k_{C_3}(z)$$

where R, R', R'', R''' are fresh and the last clause tests the intersection of languages defined by R''', k_{C_1}, …, k_{C_3}. While this last clause can be encoded using intersection and query clauses, it is really of a special form which will have its importance in Section 6 (*final intersection clauses*).

Clause (14) is more problematic, since it is not right-linear. The clauses

$$\mathit{valy}(y) \Leftarrow k_{C_1}(x; e(y)) \qquad k_{C_2}(x_1; x_2; x_3) \Leftarrow P_1(x_1), P_2(x_2), P_3(x_3)$$
$$P_1(e(N_2)) \qquad P_2(e(y)) \Leftarrow \mathit{valy}(y) \qquad P_3(e(y \oplus N_2)) \Leftarrow \mathit{valy}(y)$$

are actually equivalent in the Dolev-Yao model (where sending lists of elements or sending each element separately has the same net effect) and in the pure eavesdropper model (where any k_C recognizes at most one value anyway). As these clauses are both left and right-linear, we can transform them to push and pop automata clauses.

As we shall see, the resulting set of clauses falls into the decidable case of Section 6. In particular it avoids intersection clauses that are not final. The curious reader might want to know that the secrecy property fails, i.e., the empty clause is derivable from the above clauses, alternatively the intersection of states R''', $k_{C_1}, \ldots, k_{C_3}$ is not empty, i.e., there is an attack, in all models except the pure eavesdropper case [31]. Note that IKA.1 was indeed designed so as to be resistant only to pure eavesdroppers.

4 Undecidability Results

Modulo AC or AC1, we have the following undecidability results.

First, Lemma 1 shows the power of conditional push clauses, which allow us to simulate two-counter machines [32]. For this we do not even need intersection clauses. We get an undecidable problem even without intersection clauses. —i.e., with *non-deterministic* automata.

Lemma 1. *Every r.e. set is effectively representable as the language $L_{q_0}(\mathcal{A})$, where $\mathcal{A}$ is a non-deterministic two-way AC (resp. AC1) automaton with one AC (resp. AC1) symbol and two constants. In particular, emptiness is undecidable for this class.*

Proof. For every r.e. set E, there is a two-counter machine M (with counters R_1, R_2) such that M accepts, starting with $R_1 = 0$ and $R_2 \in E$. It then suffices to encode configurations of M that lead to acceptance using two-way AC (resp. AC1) automata. This is done by encoding the values m, n of the registers by terms $(m{+}1)a_1{+}(n{+}1)a_2$ (or by terms $ma_1 + na_2$ in the AC1 case). Recall that a two-counter machine [32] is a finite labeled transition system with an initial state q_0, a final (acceptance) state q_f, and transitions $q \xrightarrow{a} q'$ where a may be Inc R_i, Dec R_i or Zero R_i, $i \in \{1, 2\}$. Inc R_i increments R_i, Dec R_i checks whether R_i is ≥ 1, and if so decrements R_i, and Zero R_i checks whether $R_i = 0$.

A *configuration* of the machine M is a triple (q, m, n) where q is a state, $m, n \in \mathbb{N}$ are the values of R_1 and R_2 respectively. This configuration is encoded as the ground atom $q(\overline{m,n})$, where $\overline{m,n} \hat{=} (m{+}1)a_1{+}(n{+}1)a_2$, and for each $k \geq 1$, ka_i is $a_i{+}\ldots{+}a_i$ (k summands). This if for the AC case; in the AC1 case, we would code $\overline{m,n} \hat{=} ma_1 + na_2$; this is left to the reader.

Introduce the following clauses, where is_a_1, is_a_2, $r_{0,0}$, $is\_a_1^+$ and $is\_a_2^+$, and $state$ are predicate symbols distinct from all states: $is\_a_i(a_i)$ (is_a_i recognizes just a_i), $r_{0,0}(x{+}y) \Leftarrow is\_a_1(x), is\_a_2(y)$ ($r_{0,0}$ recognizes $\overline{0,0}$), $is\_a_i^+(a_i)$ and $is\_a_i^+(x{+}y) \Leftarrow is\_a_i^+(x), is\_a_i(y)$ (is_a_i recognizes all $ka_i, k \geq 1$), $state(x{+}y) \Leftarrow is\_a_1^+(x), is\_a_2^+(y)$ ($state$ recognizes all $\overline{m,n}, m, n \in \mathbb{N}$). We translate the machine M as follows:

1. Acceptance: $q_f(x) \Leftarrow state(x)$.
2. $q \xrightarrow{a} q', a =$ Inc R_i: $\quad q(y) \Leftarrow q'(x+y), is\_a_i(x), state(y)$.
3. $q \xrightarrow{a} q', a =$ Dec R_i: $\quad q(x+y) \Leftarrow is\_a_i(x), q'(y)$.

4. $q \xrightarrow{a} q', a = \text{Zero } R_i$: $\begin{cases} q_{aux}(y) \Leftarrow q'(x+y), is\_a_i(x), is\_a^+_{3-i}(y) \\ q(x+y) \Leftarrow q_{aux}(x), is\_a_i(y) \end{cases}$

where q_{aux} is a fresh predicate, one for each q.

Let $\mathcal{A}$ be the set of clauses as above. It is easy to check, first, that, if (q, m, n) is a configuration of M that leads to acceptance, i.e., to some configuration (q_f, m', n'), then $q(\overline{m,n})$ is deducible from $\mathcal{A}$ by unit resolution; second, that all unit clauses deducible from $\mathcal{A}$ by unit resolution are of the form $is\_a_i(a_i), r_{0,0}(a_1 + a_2), is\_a_i^+(ka_i)$ $(k \geq 1)$, $state(\overline{m,n})$ $(m, n \in \mathbb{N})$, or $q(\overline{m,n})$, where (q, m, n) leads to acceptance in M, or $q_{aux}((m+1)a_1)$ (resp. $q_{aux}((n+1)a_2)$), where $m, n \in \mathbb{N}$, and $(q, m, 0)$ (resp. $(q, 0, n)$) leads to acceptance in M.

Assuming without loss of generality that the only transition out of q_0 is Zero R_1, it follows that $q_0(\overline{m,n})$ is in the least Herbrand model of $\mathcal{A}$ iff $m = 0$ and (q_0, m, n) leads to acceptance in M, iff $m = 0$ and $n \in E$. □

This uses *conditional* push clauses in an essential way. We shall see in later sections that, on the opposite, *standard* push clauses can be effectively removed from AC-automata. Another problem stems from intersection clauses:

Lemma 2. *Every r.e. set is effectively representable as the language $L_{q_0}(\mathcal{A})$, where $\mathcal{A}$ is an alternating AC (resp. AC1) automaton with one AC (resp. AC1) symbol and four constants. In particular, emptiness is undecidable for this class.*

Proof. We use an encoding similar to [24]. —except that the direction of computation is reversed, as in Lemma 1 (otherwise we would only get the second part of the Lemma). Again we encode two-counter machines. However, content n of register R_i is coded as any term $n^1 a_i^1 + n^2 a_i^2$, with $n^1 \geq n^2 \geq 1$, and $n = n^1 - n^2$. In the sequel, write this term $[n^1, n^2]_i$. Incrementing R_i will be done by incrementing n^1, while decrementing will be achieved by incrementing n^2. The encoding is not one-to-one: the same value n may be coded by several terms $[n^1, n^2]_i$. We take care of this by requiring that any predicate q meant to recognize $\overline{m,n}$ recognizes every representative of it.

Introduce the following clauses, where $is\_a_1^1, is\_a_2^1, is\_a_1^2, is\_a_2^2, r_{0,0}, \ldots$, are predicate symbols distinct from all states, and $j \in \{1, 2\}$:

	recognizes:
$is\_a_i^j(a_i^j)$	just a_i^j
$zero_i(x+y) \Leftarrow is\_a_i^1(x), is\_a_i^2(y)$	
$zero_i(x+y) \Leftarrow one_i(x), is\_a_i^2(y)$	$[n,n]_i, n \geq 1$
$one_i(x+y) \Leftarrow is\_a_i^1(x), zero_i(y)$	$[n+1,n]_i, n \geq 1$
$r_{0,0}(x+y) \Leftarrow zero_1(x), zero_2(y)$	$\overline{0,0}$
$nn_i(x) \Leftarrow zero_i(x)$	
$nn_i(x+y) \Leftarrow is\_a_i^1(x), nn_i(y)$	$[n^1,n^2]_i, n^1 \geq n^2 \geq 1$
$state(x+y) \Leftarrow nn_1(x), nn_2(y)$	$\overline{m,n}, m, n \geq 0$
$st_1^+(x+y) \Leftarrow is\_a_1^1(x), state(y)$	$\overline{m+1,n}, m, n \geq 0$
$st_2^+(x+y) \Leftarrow is\_a_2^1(x), state(y)$	$\overline{m,n+1}, m, n \geq 0$
$st_1^0(x+y) \Leftarrow zero_1(x), nn_2(y)$	$\overline{0,n}, n \geq 0$
$st_2^0(x+y) \Leftarrow nn_1(x), zero_2(y)$	$\overline{m,0}, m \geq 0$

Also, with each state q, associate two fresh predicate symbols q_1^+ and q_2^+, distinct from each other, from every state, and from every predicate introduced above. Add

the intersection clauses $q_i^+(x) \Leftarrow q(x), st_i^+(x)$ for $i \in \{1,2\}$; q_i^+ recognizes every configuration recognized by q such that R_i is not zero. We translate the machine M as follows:

1. Acceptance: $q_f(x) \Leftarrow state(x)$.
2. $q \xrightarrow{a} q', a =$ Inc R_i: $q(x+y) \Leftarrow is\_a_i^2(x), q'^+_i(y)$.
3. $q \xrightarrow{a} q', a =$ Dec R_i: $q(x+y) \Leftarrow is\_a_i^1(x), q'(y)$.
4. $q \xrightarrow{a} q', a =$ Zero R_i: $q(x) \Leftarrow q'(x), st_i^0(x)$.

The rest of the proof is as for Lemma 1. □

By a remark in [24], three constants actually suffice for this Lemma.

Note also that even without intersection clauses and conditional push clauses, but provided we allow for non-Horn clauses, satisfiability is again undecidable. Indeed, replace the intersection clauses $q(x) \Leftarrow q_1(x), q_2(x)$ with the two clauses $-P(z) \vee -q_1(z)$ and $+P(x) \vee +q(x) \vee -q_2(x)$, with P fresh.

5 Deciding The Constant-Only Case

We warm up by solving the case of one AC symbol $+$, and finitely many constants symbols $a_1, \ldots, a_p$—the so-called *constant-only* case. This will turn out to be the core of the general problem in Section 6.

Because of the negative results of Section 4, we must restrict the format of clauses. Without loss of generality we may assume that we use clauses of the following form:

$$P(x+y) \Leftarrow P_1(x), P_2(y) \quad (17) \qquad P(x) \Leftarrow P_1(x+y) \quad (20)$$

$$P(a_i) \quad (18) \qquad \bot \Leftarrow P_1(x), \ldots, P_k(x) \quad (21)$$

$$P(x) \Leftarrow P_1(x) \quad (19) \qquad \bot \Leftarrow P(u) \quad (22)$$

where x and y are distinct variables, and u is a closed term. Clauses (17) are *pop clauses*, (18) *base clauses*, (19) *ϵ-clauses*, (20) standard *push clauses*, (21) *final intersection clauses*, or *query clauses* when $k = 1$, and (22) *test clauses*.

Definition 1. *An* AC^0-automaton *is a finite set of pop clauses (17), of base clauses (18), and of ϵ-clauses (19).*

By standard marking techniques, it is decidable whether any given state P of an AC^0-automaton $\mathcal{A}$ is empty in $\mathcal{A}$. Things get more complex in the presence of final intersection clauses. First, note that ground terms in the constant-only case are finite linear combinations $\sum_{i=1}^p n_i a_i$, with $n_i \in \mathbb{N}$ and $\sum_{i=1}^p n_i \geq 1$: equivalently, non-zero p-tuples of natural numbers. Recall that a set of p-tuples of natural numbers is *linear* if and only if it can be written $\{a + b_1 + \ldots b_k | k \in \mathbb{N}, b_1, \ldots, b_k \in B\}$ where $+$ is taken componentwise, $a \in \mathbb{N}^p$ and B is a finite subset of $\mathbb{N}^p$. A *semi-linear* set is a finite union of linear sets. The semi-linear sets are exactly the sets definable in Presburger arithmetic [20]. Also, the commutative image of any context-free language is semi-linear, and in fact effectively so: this is Parikh's Theorem [36]. Now observe that if we read clauses (17), (18), (19) modulo associativity (A) instead of mod AC, what we get is exactly a context-free grammar: (17) is usually written $P \to P_1, P_2$, (18) is $P \to a_i$, (19) is $P \to P_1$. Call a set *AC^0-recognizable* iff it is $L_P(\mathcal{A})$ for some AC^0-automaton $\mathcal{A}$. Then:

Lemma 3 (Parikh). *For every AC^0-automaton $\mathcal{A}$, $L_P(\mathcal{A})$ is an effective semi-linear set. The AC^0-recognizable sets are the semi-linear sets of non-zero tuples of integers.*

The results of Section 4 imply that sets recognized by AC-automata with conditional push clauses or even just intersection transitions are in general *not* semi-linear. Nonetheless, any finite intersection of semi-linear sets is semi-linear, so:

Lemma 4. *The satisfiability of sets of clauses (17), (18), (19), (21), (22) is decidable.*

Proof. Let $\mathcal{A}$ be all non-test, non-final intersection clauses in the given set S, $\bot \Leftarrow P_1^i(x), \ldots, P_{n_i}^i(x)$ be the final intersection clauses in S, and $\bot \Leftarrow P^j(u_j)$ be the test clauses in S. By Lemma 3 the languages $L_{P_j^i}(\mathcal{A})$ and $L_{P^j}(\mathcal{A})$ are effectively semi-linear. Then S is unsatisfiable iff for some i, $L_{P_1^i}(\mathcal{A}) \cap \ldots \cap L_{P_{n_i}^i}(\mathcal{A}) \neq \emptyset$, or for some $j, u_j \in L_{P^j}(\mathcal{A})$, which is effectively decidable. □

In passing, the connection between automata mod A and context-free grammars shows that the emptiness problem for A-automata with one final intersection clause is undecidable, even without push clauses. This is by reduction from the emptiness problem for intersection of context-free languages. In particular, A-recognizable sets are not closed under intersection.

The AC case is tamer. By Lemma 4, AC^0-recognizable languages are closed under intersection, union, complementation, and projection.

We shall spend the rest of this section proving that satisfiability is also decidable in the presence of standard push clauses (20). First, this is the case provided we don't ask for emptiness of intersections, i.e., if clauses (21) are restricted to query clauses ($k = 1$). This is Lemma 5 below.

Definition 2 (Two-way AC). *A* two-way AC^0-automaton *is a finite set of pop clauses (17), of base clauses (18), of ϵ-clauses (19), and of standard push clauses (20).*

For any ground term $t \hat{=} \sum_{i=1}^p n_i a_i$, its *length* is $|t| \hat{=} \sum_{i=1}^p n_i$. Write $t + \langle n \rangle$ ("t plus n things") the term $t + x_1 + \ldots + x_n$, where $x_1, \ldots, x_n$ are variables that occur nowhere else, $n \in \mathbb{N}$. Let $\langle n+1 \rangle \hat{=} x + \langle n \rangle$ where x occurs nowhere else.

Lemma 5. *The satisfiability of sets of clauses (17), (18), (19), (20), (22), and query clauses (21) with $k = 1$, is decidable.*

In particular, the following problems are decidable: 1. whether $L_P(\mathcal{A})$ is empty, and 2. whether the ground term u is in $L_P(\mathcal{A})$, where $\mathcal{A}$ is a two-way AC^0-automaton.

Proof. The second part of the Lemma follows from the first part by adding one query clause $\bot \Leftarrow P(x)$ (problem 1.), or one test clause $\bot \Leftarrow P(u)$ (problem 2.) to $\mathcal{A}$. Let $\bot \Leftarrow P^1(u_1), \ldots, \bot \Leftarrow P^q(u_q)$ be the test clauses in the given set of clauses. Define $s \prec t$ iff $s + u = t$ for some ground term u, $s \preceq t$ iff $s \prec t$ or $s = t$. We use negative hyperresolution with eager splitting and forward subsumption, which is a sound and complete resolution strategy. We need to generalize the format of query and test clauses to:

$$\bot \Leftarrow P(\langle n+1 \rangle) \qquad (23) \qquad\qquad \bot \Leftarrow P(t + \langle n \rangle) \qquad (24)$$

where $n \in \mathbb{N}$, $t \preceq u_j$ for some j, $1 \leq j \leq q$. The case of a query clause $\bot \Leftarrow P(x)$ is

(23) when $n = 0$, that of a test clause $\bot \Leftarrow P(u)$ is (24) when $t = u, n = 0$. Negative hyperresolution is the binary resolution rule, where one premise is constrained to be a negative clause. We claim that: $(*)$ the only negative clauses we shall ever have are of the form (23) and (24). Note that factoring is not needed, because all our clauses are Horn (see e.g., [15], Definition 4.1). We prove claim $(*)$ by induction on the length of a negative hyperresolution derivation with splitting. Resolving a clause (23) or (24) with a clause (18), if possible at all, yields the empty clause. Resolving against (19) or (20) again yields a clause (23) or (24). It remains to examine the two cases where we resolve against a pop clause (17), say $P(x + y) \Leftarrow P_1(x), P_2(y)$.

Case 1: with a clause (23), say $\bot \Leftarrow P(\langle n + 1\rangle)$. Close examination of all the AC-most general unifiers between $\langle n + 1\rangle$ and $x + y$ show that they all map x to $\langle n_1 + 1\rangle$ and y to $\langle n_2 + 1\rangle$, where $n_1, n_2 \in \mathbb{N}$ and $n = n_1 + n_2 + 1$ (unless $n = 0$, where $n_1 = n_2 = 0$), and that every such substitution $\{x \mapsto \langle n_1 + 1\rangle, y \mapsto \langle n_2 + 1\rangle\}$ is thus obtained. So the resolvents are of the form $\bot \Leftarrow P_1(\langle n_1 + 1\rangle), P_2(\langle n_2 + 1\rangle)$ with $n = n_1 + n_2 + 1$. Then, these split as two clauses of the form (23).

Case 2: with a clause (24), say $\bot \Leftarrow P(t+\langle n\rangle)$ with $t \preceq u_j$ for some $j, 1 \leq j \leq q$. The binary resolvents are of the form $\bot \Leftarrow P_1(t_1+\langle n_1\rangle), P_2(t_2+\langle n_2\rangle)$ with $t_1+t_2 = t$, $n_1+n_2 = n$, or $\bot \Leftarrow P_1(t+\langle n_1\rangle), P_2(\langle n_2 + 1\rangle)$ with $n_1+n_2+1 = n$ (unless $n = 0$, where $n_1 = n_2 = 0$), or $\bot \Leftarrow P_1(\langle n_1 + 1\rangle), P_2(t+\langle n_2\rangle)$ with $n_1+n_2+1 = n$ (unless $n = 0$, where $n_1 = n_2 = 0$). There are finitely many such resolvents, and they all split as two clauses of the form (23) or (24). This finishes to prove $(*)$.

We now show that negative hyperresolution with splitting and forward subsumption terminates. Otherwise, there is an infinite branch, containing an infinite sequence of clauses (23) or (24) since there are only finitely many other clauses. So there is an infinite subsequence of clauses $\bot \Leftarrow P(\langle n_k + 1\rangle), k \in \mathbb{N}$, with the same predicate P, or $\bot \Leftarrow P(t + \langle n_k\rangle), k \in \mathbb{N}$, with the same P and the same t. (In the latter case, this is because there are only finitely many t with $t \preceq u_j$ for some j.) Since $\leq$ is a well-ordering on $\mathbb{N}$, there are two indices $k < k'$ such that $n_k \leq n_{k'}$. Then the kth clause in the subsequence subsumes the k'th, which is impossible. □

Call a two-way AC^0-automaton $\mathcal{A}$ *normal* iff no state of $\mathcal{A}$ is empty. Lemma 5 allows us to compute empty states; then we may safely remove all clauses with an empty state in their body. Therefore:

Lemma 6. *There is an effective procedure transforming any two-way AC^0-automaton $\mathcal{A}$ into a normal one $\mathcal{B}$, such that for every non-empty state P of $\mathcal{A}$, $L_P(\mathcal{A}) = L_P(\mathcal{B})$.*

Call a state P in $\mathcal{A}$ *bounded* iff $L_P(\mathcal{A})$ is finite; equivalently, if there is an upper bound on the lengths $|t|$ of terms t recognized at P in $\mathcal{A}$. Call P *unbounded* otherwise.

Lemma 7. *The set $B_\mathcal{A}$ of bounded states, in any given normal two-way AC^0-automaton $\mathcal{A}$, is effectively computable. Moreover, for every $P \in B_\mathcal{A}$ we can effectively compute upper bounds on $|t|, t \in L_P(\mathcal{A})$.*

Proof. Let B be any set of predicate symbols (states) in $\mathcal{A}$. We build a system C_B of inequality constraints on the set of variables x_P^B, $P \in B$ as follows. For each clause in $\mathcal{A}$, generate false if the head predicate (on the left of $\Leftarrow$) is in B but some predicate in the body (on the right) is not in B; generate true if the head predicate is not in B;

otherwise, all predicates are in B, and for each clause:
– of the form (17), generate $x^B_P \geq x^B_{P_1} + x^B_{P_2}$;
– of the form (18), generate $x^B_P \geq 1$;
– of the form (19), generate $x^B_P \geq x^B_{P_1}$;
– of the form (20), generate $x^B_P \geq x^B_{P_1} - 1$.
Finally, generate $x^B_P \geq 1$ for each $P \in B$. If every $P \in B$ is bounded and every $P \notin B$ is unbounded, and x^B_P is the maximal length of terms recognized at $P \in B$ in $\mathcal{A}$ (this exists because no state is empty in $\mathcal{A}$), then all inequalities in C_B are satisfied. Conversely, if C_B is satisfied, then every $P \in B$ is bounded, and any satisfying valuation maps x^B_P to an upper bound on the lengths of terms t recognized at P in $\mathcal{A}$; this is by induction of the lengths of unit resolution derivations of $P(t)$. So the set B_0 of bounded states is the unique maximal set B such that the system of inequalities above is satisfiable. This can be built by enumerating all subsets B of states and taking the largest for which C_B is satisfiable. Satisfiability of C_B can be decided by, e.g., Section 6 of [40]. □

Let $A \subseteq \{a_1, \ldots, a_p\}$. The *projection on* A of a set S of terms is $\pi_A(S) \hat{=} \{\sum_{a_i \in A} n_i a_i | \sum_{i=1}^p n_i a_i \in S\}$. This may include the (fictitious) empty sum $\mathbf{0}$. To remain in the realm of AC-terms, write $\pi^+_A(S) \hat{=} \pi_A(S) \setminus \{\mathbf{0}\}$. A state P is *bounded on* A (in $\mathcal{A}$) iff $\pi_A(L_P(\mathcal{A}))$ is bounded. The *bounded support* of P is the maximal $A \subseteq \{a_1, \ldots, a_p\}$ such that P is bounded on A. We may compute automata recognizing projections $\pi_A(L_P(\mathcal{A}))$; that $\mathbf{0}$ is not an AC-term merely complicates the statement of the Lemma:

Lemma 8. *There is an effective procedure that, given a finite set A of constants, given any two-way AC^0-automaton $\mathcal{A}$, produces another $\mathcal{B}$ such that $L_P(\mathcal{B}) = \pi^+_A(L_P(\mathcal{A}))$ for every P in $\mathcal{A}$ (we write $\mathcal{B}$ as $\pi^+_A(\mathcal{A})$), and computes the set $\mathcal{P}$ of all predicates in $\mathcal{A}$ such that $\mathbf{0} \in \pi_A(\mathcal{A})$.*

Proof. The idea is to replace every base clause $P(a)$ with $a \notin A$ by $P(\mathbf{0})$—except we do not have a zero. Instead, let $\mathcal{P}$ be the smallest set of states such that: (a) if $\mathcal{A}$ contains a base clause $P(a)$ with $a \notin A$, then $P \in \mathcal{P}$; (b) if $\mathcal{A}$ contains an ϵ-clause $P(x) \Leftarrow P_1(x)$ with $P_1 \in \mathcal{P}$, then $P \in \mathcal{P}$; and (c) if $\mathcal{A}$ contains a pop clause $P(x+y) \Leftarrow P_1(x), P_2(y)$ with $P_1, P_2 \in \mathcal{P}$, then $P \in \mathcal{P}$. This is easily computable. Now build $\mathcal{B}$ by generating:
– for each pop clause $P(x+y) \Leftarrow P_1(x), P_2(y)$ of $\mathcal{A}$, this clause, plus $P(x) \Leftarrow P_1(x)$ if $P_2 \in \mathcal{P}$, plus $P(x) \Leftarrow P_2(x)$ if $P_1 \in \mathcal{P}$;
– for each base clause $P(a)$ with $a \in A$, this clause, otherwise none;
– for each ϵ or push clause, this clause.
By induction on the length of a unit resolution derivation of $P(\sum_{i=1}^p n_i a_i)$ from $\mathcal{A}$, it is easy to show that either $\sum_{a_i \in A} n_i = 0$ and $P \in \mathcal{P}$, or $\sum_{a_i \in A} n_i \geq 1$ and $P(\sum_{a_i \in A} n_i a_i)$ is derivable in $\mathcal{B}$. Conversely, by induction on a proof of $P \in \mathcal{P}$—using rules (a), (b), (c)—it is clear that $\mathbf{0} \in \pi_A(L_P(\mathcal{A}))$; and by induction on the length of a unit resolution derivation of $P(t)$ in $\mathcal{B}$, $P(t')$ is derivable in $\mathcal{A}$ for some $t' \succeq t$ (in the case of ϵ-clauses generated from pop clauses of $\mathcal{A}$, this is because, for every $P_2 \in \mathcal{P}$, since $\mathbf{0} \in \pi_A(L_{P_2}(\mathcal{A}))$, in particular $L_{P_2}(\mathcal{A})$ is not empty); in particular, if $t = \sum_{a_i \in A} n_i a_i$ is recognized at P in $\mathcal{B}$, then t is indeed the projection of some term recognized at P in $\mathcal{A}$. □

By starting from $A = \emptyset$, and adding constants a_i while $L_P(\pi_A(\mathcal{A}))$ is bounded, using Lemma 7 and Lemma 8, we get:

Corollary 1. *There is an effective procedure that, given a normal two-way automaton $\mathcal{A}$, computes the bounded support of each state P of $\mathcal{A}$.*

From all this we finally deduce:

Theorem 1. *There is an effective procedure transforming any two-way AC^0-automaton $\mathcal{A}$ into a AC^0-automaton $\mathcal{B}$ such that for every P in $\mathcal{A}$, $L_P(\mathcal{A}) = L_P(\mathcal{B})$.*

Proof. By Lemma 6 we may assume that $\mathcal{A}$ is normal. Using Corollary 1, for each push clauses $P(x) \Leftarrow P_1(x+y)$ compute the bounded support A of P_1 in $\mathcal{A}$. By definition the terms recognized at P_1 are of the form $\sum_{a_i \in A} n_i a_i + \sum_{a_i \notin A} n_i a_i$ where there are only finitely many values for the first summand, and where the n_is, $a_i \notin A$, have unbounded values. If $A = \{a_1, \ldots, a_p\}$, P_1 is bounded. Enumerate the sums $\sum_{i=1}^{p} n_i a_i$ (except $\mathbf{0}$) that are less than some element of $L_{P_1}(\mathcal{A})$ w.r.t. $\prec$, using Lemma 5, and produce an AC-automaton $\mathcal{A}_1$ built on fresh states, recognizing this set at say q_1. Then replace the push clause $P(x) \Leftarrow P_1(x+y)$ by the clause $P(x) \Leftarrow q_1(x)$.

If P_1 is unbounded, then enumerate the sums $\sum_{a_i \in A} n_i a_i$ (except $\mathbf{0}$) by enumerating the elements of $\pi_A^+(L_{P_1}(\mathcal{A}))$, using Lemma 8 and Lemma 5, and produce an AC-automaton $\mathcal{A}_1$ recognizing this set at q_1, built on fresh states. Similarly, produce an AC-automaton $\mathcal{A}_2$ recognizing all (non-zero) terms $\sum_{a_i \notin A} n_i a_i$ (with $n_i \in \mathbb{N}$ arbitrary) at q_2, again built on fresh states. Then replace the push clause $P(x) \Leftarrow P_1(x+y)$ by the clauses $P(x+y) \Leftarrow q_1(x), q_2(y)$, plus $P(x) \Leftarrow q_1(x)$, plus $P(x) \Leftarrow q_2(x)$ if $\mathbf{0} \in \pi_A(L_{P_1}(\mathcal{A}))$ (the latter is decidable by Lemma 8). □

In particular, for every two-way AC^0-automaton $\mathcal{A}$, $L_P(\mathcal{A})$ is an effective semilinear set. So the languages of two-way AC^0-automata are closed under intersection, union, complementation and projection. Also, by Theorem 1 and Lemma 4:

Corollary 2. *The satisfiability of sets of clauses of the form (17)–(22) is decidable.*

This was the main result of this section. Later, we shall need the following corollary:

Corollary 3. *Given a set $\mathcal{A}$ of clauses of the form (17)–(21), and a disjunction $C \hat{=} B_1(x_1) \vee \ldots \vee B_n(x_n)$ of blocks, where the x_is are pairwise distinct, it is decidable whether $\mathcal{A} \models C$.*

Proof. Equivalently, whether $\mathcal{A} \models (\forall x_1 \cdot B_1(x_1)) \vee \ldots \vee (\forall x_n \cdot B_n(x_n))$. Let $B_k(x_k)$ be $\pm_{k1} P_{k1}(x_k) \vee \ldots \vee \pm_{km_k} P_{km_k}(x_k)$. Skolemize by creating n new constants $b_1, \ldots, b_n$. Then $\mathcal{A} \models C$ iff $\mathcal{A}$ union the clauses $\mp_{kj} P_{kj}(b_k)$, $1 \leq k \leq n$, $1 \leq j \leq m_k$, is unsatisfiable, where $\mp_{kj}$ is the sign opposite to $\pm_{kj}$. We conclude by Corollary 2. □

Note that $\models$ is entailment in *Tarskian semantics*, where terms take their values in some arbitrary non-empty domain D (see [6]). This is required because skolemization is valid in Tarskian, not Herbrand semantics. Tarskian entailment implies Herbrand entailment, because Herbrand interpretations are special cases of interpretations in the sense of Tarski, but not conversely.

6 Deciding The General Case

We turn to deciding emptiness of AC-automata with push clauses, based on unrestricted signatures Σ containing binary symbols $+_1, \ldots, +_p$ that are AC. Call the other function symbols in Σ *free*. Zero-ary function symbols are constants. Considering the undecidability results of Section 4, the largest class of AC-automata for which we can reasonably hope to decide the emptiness problem is as follows.

Definition 3 (Two-Way AC). *A* non-alternating standard-two-way AC-automaton $\mathcal{A}$ *on Σ is a finite set of clauses of one of the following forms:*

- free *pop clauses (1) where f is a free symbol,*
- $+_i$ pop *clauses* $P(x +_i y) \Leftarrow P_1(x), P_2(y)$*,*
- free *push clauses (3) (possibly conditional) with* $i \notin \{i_1, \ldots, i_k\}$*,*
- ϵ*-clauses* $P(x) \Leftarrow P_1(x)$ *(19),*
- standard $+_i$ push *clauses* $P(x) \Leftarrow P_1(x +_i y)$ *(20).*

We do allow conditional push clauses on free function symbols, while push clauses must be standard on AC symbols. To cope with intersection-emptiness, we add final intersection clauses (21) to non-alternating standard-two-way AC-automata.

Note the additional restriction $i \notin \{i_1, \ldots, i_k\}$ on free push clauses. This is required for technical purposes: otherwise the free push/free pop entry in Figure 1 below would be wrong. More precisely, if we did not require this, it would be possible to emulate intersection clauses $P(x) \Leftarrow P_1(x), P_2(x)$ by creating one pop clause $q(f(x)) \Leftarrow P_1(x)$ and one push clause $P(x) \Leftarrow q(f(x)), P_2(x)$ (provided q is fresh, both are equivalent), leading to undecidability.

We base our study on ordered resolution, where the literals resolved upon in each premise must be maximal with respect to some stable ordering $\succ$, with eager splitting. Specifically, let $||t||$ denote the *size* of t, and define $s \succ t$ if and only if $||s\sigma|| > ||t\sigma||$ for every ground substitution σ. Ordered resolution with eager splitting is complete, but does *not* yield a terminating decision algorithm straight out of the box. However we take it as a starting point. First examine the shape of possible resolvents generated in an ordered resolution derivation. Since our clauses are Horn, we don't need factoring.

The first kind of clauses that crop up are the *general $+_i$ transitions*:

$$P(x_1 +_i \ldots +_i x_k) \Leftarrow P_1(x_1 +_i \langle n_1 \rangle_i), \ldots, P_k(x_k +_i \langle n_k \rangle_i) \tag{25}$$

where $k \geq 1$, the x_js are pairwise distinct, $n_1, \ldots, n_k \in \mathbb{N}$, and if $k = 1$ then $n_1 \geq 1$. The notations $t +_i \langle n \rangle_i$ and $\langle t \rangle_i$ extend the unsubscripted notations by using $+_i$ for $+$.

Then we also get *general $+_i$ final clauses*:

$$\bot \Leftarrow P_1(t_1), \ldots, P_k(t_k) \tag{26}$$

where $t_1, \ldots, t_k$ are terms built on variables and the sole function symbol $+_i$, and if some t_j is a variable x, then x is free in some non-variable term $t_{j'}$.

Finally, we shall get *free final clauses* the negative complex clauses (5) where f is a free function.

	free push	free pop	free final	ϵ	final inter.	$+_i$ trans.	$+_i$ final
free push	–	$\bigvee\{\epsilon,$[1] final inter.$\}$	–	free push	–	–	–
free pop		–	free final/[1] $\bigvee$final inter.	free pop	free final/[1] $\bigvee$final inter.	–	–
free final			–	free final	–	–	–
ϵ				ϵ	final inter.	$+_i$ trans.	$+_i$ final
final inter.					–	$\bigvee\{$final inter., $+_i$ final$\}$	–
$+_{i'}$ trans.						$\bigvee\{+_i$ trans.,[2] ϵ, final inter., $+_i$ final$\}$	$\bigvee\{$final inter.,[2] $+_i$ final.$\}$
$+_{i'}$ final							–

[1] provided the free function symbols in each clause coincide.
[2] provided $i = i'$

Fig. 1. What resolution with splitting generates

Lemma 9. *The possible resolvents between clauses considered above are as summarized in Figure 1. – denotes no possible resolvent, A/B means this generates clauses of kind A or clauses of kind B, $\bigvee\{A_1, \ldots, A_n\}$ means this generates disjunctions of variable-disjoint clauses of kind A_1, …, A_n; $\bigvee A$ abbreviates $\bigvee\{A\}$. (Some entries are not filled because of symmetry.) $+_i$ trans. abbreviates "general $+_i$ transition", $+_i$ final abbreviates "general $+_i$ final clause", and so on.*

Note that standard $+_i$ push clauses are the special case of general $+_i$ transitions where $k = 1$ and $n_1 = 1$; and that $+_i$ pop clauses are the special case $k = 2, n_1 = n_2 = 0$.

Proof. Most of it is standard and boring verification. The difficult cases are at the bottom-right of the array.

The $+_{i'}$ trans./$+_i$ final case works as follows: taking the notations of (26) and (25), we compute the mgus of t_j (from the former) with $x_1 +_i \ldots +_i x_k$. By the side-conditions on $+_{i'}$ and since t_j must be maximal, t_j is not a variable, so is headed by $+_{i'}$. For unification to succeed, then, $i = i'$. On the other hand any mgu will map every free variable of t_j and of $x_1, \ldots, x_k$ to $+_i$-sums of fresh variables. The resulting resolvent is then a disjunction of terms built on variables and $+_i$ alone. This always splits in $+_i$ final clauses and final intersection clauses (for those variable t_js that do not occur in non-variable $t_{j'}$s).

Instead of dealing with the other cases, we deal with the technically trickiest, the $+_{i'}$ trans./$+_i$ trans. case. Again by maximality considerations, $i = i'$. For simplicity, write just $+$ instead of $+_i$. Without loss of generality, we resolve:

$$P(x_1 + \ldots + x_k) \Leftarrow P_1(x_1 + \langle n_1 \rangle), \ldots, P_k(x_k + \langle n_k \rangle)$$
$$P_1(y_1 + \ldots + y_\ell) \Leftarrow Q_1(y_1 + \langle m_1 \rangle), \ldots, Q_k(y_\ell + \langle m_\ell \rangle)$$

We may assume that $n_1 \geq 1$, otherwise the argument of P_1, x_1, is smaller than $x_1 + \ldots + x_k$, so is not maximal—unless $k = 1$, but then the side-condition of $+$ transitions requires $n_1 \geq 1$. The most general unifiers of $y_1 + \ldots + y_\ell$ with $x_1 + \langle n_1 \rangle$ map x_1 to $\sum_{j \in J} y'_j$, y_j to $y'_j + \langle n'_j \rangle$ if $j \in J$ and to $\langle n'_j \rangle$ otherwise, where J ranges over the non-empty subsets of $\{1, \ldots, \ell\}$, and the integers n'_j, $1 \leq j \leq \ell$ are such that $\sum_{j=1}^{\ell} n'_j = n_1$, with $n'_j \geq 1$ whenever $j \notin J$; here the y'_js are fresh pairwise distinct variables. The corresponding resolvent is

$$P\left(\sum_{j \in J} y'_j + x_2 + \ldots + x_k\right) \Leftarrow \bigwedge_{j \in J} Q_j(y'_j + \langle n'_j + m_j \rangle), \bigwedge_{j \notin J} Q_j(\langle n'_j + m_j \rangle), \\ P_2(x_2 + \langle n_2 \rangle), \ldots, P_k(x_k + \langle n_k \rangle)$$

The terms $Q_j(\ldots)$, $j \notin J$, split out of this as final intersection clauses. What remains is a general $+_i$ transition, unless $k = 1$, J is a singleton $\{j\}$, and $n'_j + m_j = 0$, where we get an ϵ-clause. □

Consider a refutation π of some non-alternating standard-two-way AC-automaton with final intersection clauses by ordered resolution and splitting, from a set of clauses S; assume π *minimal size*. Recall this is a *tableau* proof, organized as a tree of clause sets. Then every (split part of a) resolvent is either the empty clause or is used later on in the refutation. Call general $+_i$ transitions and general $+_i$ final clauses the *uncontrollable* clauses: all the others are finitely many by construction; call them *controllable*. If π contains an uncontrollable clause, then since S only contains controllable clauses, there are two controllable clauses whose resolvent splits as a disjunction of clauses, at least one of which is uncontrollable. Looking at Figure 1, uncontrollable clauses in some premise of the resolution rule only yield uncontrollable clauses again (based on the same AC function symbol $+_i$), or ϵ or final intersection clauses. In a refutation, all branches are closed; also, every uncontrollable clause is used, and the empty clause is controllable; so working our way downwards along each branch (think of a tableau as expanding downwards) we produce sequences of uncontrollable clauses on the same symbol $+_i$ that eventually contribute some ϵ or final intersection clause C_B, on each branch B. Collect C_Bs as a disjunction $\bigvee_B C_B$. Working our way upwards now, again we only find uncontrollable clauses on the same function symbol $+_i$, until we reach ϵ, final intersection clauses, or standard $+_i$ push clauses, or $+_i$ pop clauses (we stop there even though the latter two are uncontrollable by our definition). These clauses form a subset of $(S'/+_i)$ for some node S' of the tableau, where $(S'/+_i)$ denotes the subset of S' consisting of ϵ and final intersection clauses, as well as standard $+_i$ push clauses and $+_i$ pop clauses (with the *same* $+_i$). On the other hand, $\bigvee_B C_B$ is by construction a logical consequence of $(S'/+_i)$. Corollary 3 provides an effective way of deciding this. (Please note that this works precisely because $\bigvee_B C_B$ is a consequence of $(S'/+_i)$ in *Tarskian* semantics, not just in Herbrand semantics. See the remark after Corollary 3.) Also, up to condensing, there are only finitely many disjunctions $\bigvee_B C_B$.

So we may instead infer the C_Bs using the following effective deduction rule:

$$\frac{S'}{\bigvee_{i=1}^{p} C_i} \text{ if } (S'/+_i) \models \bigvee_{i=1}^{p} C_i \tag{27}$$

where S' is the current node (clause set) on the current branch, the C_is are pairwise distinct up to renaming, and each C_i is either an ϵ-clause or a final intersection clause. (Read this rule just like the resolution rule: the conclusion has to be added to the current set of clauses.) Corollary 3 provides an effective way of checking the logical consequence required above.

Theorem 2. *Call* restricted *ordered resolution the rule where ordered resolvents are only retained if they are disjunctions of variable-disjoint controllable clauses, standard $+_i$ push clauses and $+_i$ pop clauses. Then restricted ordered resolution with splitting and rule (27) is sound, complete, and terminates on non-alternating standard-two-way AC-automata with final intersection clauses.*

In particular, the emptiness of the intersection of languages defined by non-alternating standard-two-way AC-automata is decidable.

Proof. Soundness is obvious. For completeness, repeat the argument above, this time taking a minimal tableau proof using restricted ordered resolution with splitting and (27). Termination follows from the fact that there are only finitely many controllable, standard $+_i$ push clauses and $+_i$ pop clauses on a given signature. □

By making some $+_i$ symbols be AC1, with respective units $\mathbf{0}_i$, we get so-called *standard-two-way mixed AC/AC1-automata*:

Corollary 4. *The emptiness of the intersection of languages defined by standard-two-way mixed AC/AC1-automata $\mathcal{A}$ is decidable.*

Proof. Build a new non-alternating standard-two-way AC (not mixed)-automaton $\mathcal{B}$ from $\mathcal{A}$ by replacing each $+_i$ pop clause $P(x +_i y) \Leftarrow P_1(x), P_2(y)$ where $+_i$ is AC1, by the clauses $P(x +_i y) \Leftarrow P_1(x), P_2(y)$, plus $P(x) \Leftarrow P_1(x), P_2(\mathbf{0}_i)$ and $P(y) \Leftarrow P_1(\mathbf{0}_i), P_2(y)$ (understand the latter as splitting into one ϵ-clause and a free push clause on $\mathbf{0}_i$ each); and each $+_i$ push clause $P(x) \Leftarrow P_1(x +_i y)$ by $P(x) \Leftarrow P_1(x +_i y)$ plus the ϵ-clause $P(x) \Leftarrow P_1(x)$. We leave it as an easy exercise to show that if $P(t)$ is derivable from $\mathcal{B}$, then $P(t)$ is derivable in $\mathcal{A}$; and conversely, if $P(t)$ is derivable in $\mathcal{A}$ then for some t' obtained from t by using the equations $x +_i \mathbf{0}_i = x$, $P(t')$ is derivable in $\mathcal{B}$. □

7 Conclusion

We have classified alternating two-way mixed AC/AC1-automata according to the decidability of the intersection-emptiness question. Essentially, alternation or conditional push clauses lead to undecidability. On the other hand we were able to give a decision algorithm for standard-two-way mixed AC/AC1-automata without alternation, and with only standard push clauses on equational symbols.

A question spurred by the cryptographic protocol-related motivations in the introduction is: although conditional push clauses (3) where f is AC lead to undecidability, do we get a decidable class when additionally $i \notin \{i_1, \ldots, i_k\}$ in (3)? This is still open. We let the reader check that Petri nets are easily encoded in AC^0-automata with such conditional push clauses; hence the latter recognize strictly more sets than the semi-linear sets, so the techniques of Section 5 don't adapt to this case. These automata would be useful, e.g. in modeling the A-GDH.2 protocol, the authenticated version of IKA.1 mentioned in Section 3, which requires conditional push clauses.

References

1. W. Ackermann. *Solvable Cases of the Decision Problem*. Studies in Logic and the Foundations of Mathematics. North-Holland, Amsterdam, 1954.
2. M. Baaz, U. Egly, and A. Leitsch. *Normal Form Transformations*, chapter 5, pages 273–333. Volume I of Robinson and Voronkov [38], 2001.
3. L. Bachmair, H. Ganzinger, and U. Waldmann. Set constraints are the monadic class. In *Proc. 8th Annual IEEE Symposium on Logic in Computer Science (LICS'93)*, pages 75–83. IEEE Computer Society Press, 1993.
4. B. Bogaert and S. Tison. Equality and disequality constraints on direct subterms in tree automata. In *Proceedings of Symp. on Theoretical Aspects of Computer Science (STACS '92)*, pages 161–172. Springer-Verlag LNCS 577, 1992.
5. A. Bouhoula and J.-P. Jouannaud. Automata-driven automated induction. In *LICS'97*, 1997.
6. C.-L. Chang and R. C.-T. Lee. *Symbolic Logic and Mechanical Theorem Proving*. Computer Science Classics. Academic Press, 1973.
7. W. Charatonik and A. Podelski. Set-based analysis of reactive infinite-state systems. In B. Steffen, editor, *TACAS'98*, pages 358–375. Springer-Verlag LNCS 1384, 1998.
8. H. Comon, V. Cortier, and J. Mitchell. Tree automata with one memory, set constraints and ping-pong protocols. In *ICALP'2001*, pages 682–693. Springer-Verlag LNCS 2076, 2001.
9. H. Comon, M. Dauchet, R. Gilleron, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi. Tree automata techniques and applications. `www.grappa.univ-lille3.fr/tata`, 1997.
10. H. Comon and F. Jacquemard. Ground reducibility is EXPTIME-complete. In *LICS'97*, Varsaw, June 1997. IEEE Comp. Soc. Press.
11. B. Courcelle. On recognizable sets and tree automata. In M. Nivat and H. Aït-Kaci, editors, *Resolution of Equations in Algebraic Structures*. Academic Press, 1989.
12. W. Diffie and M. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, IT-22(6):644–654, 1976.
13. D. Dolev and A. C. Yao. On the security of public key protocols. *IEEE Transactions on Information Theory*, IT-29(2):198–208, 1983.
14. E. A. Emerson and C. S. Jutla. The complexity of tree automata and logics of programs (extended abstract). In *29th FOCS*, pages 328–337, 1988.
15. C. Fermüller, A. Leitsch, U. Hustadt, and T. Tammet. *Resolution Decision Procedures*, chapter 25, pages 1791–1849. Volume II of Robinson and Voronkov [38], 2001.
16. T. Frühwirth, E. Shapiro, M. Y. Vardi, and E. Yardeni. Logic programs as types for logic programs. In *LICS'91*, 1991.
17. F. Gécseg and M. Steinby. Tree languages. In G. Rozenberg and A. Salomaa, editors, *Handbook of Formal Languages*, volume 3, pages 1–68. Springer Verlag, 1997.
18. T. Genet. Decidable approximations of sets of descendants and sets of normal forms. In *RTA'98*, pages 151–165. Springer Verlag LNCS 1379, 1998.

19. T. Genet and F. Klay. Rewriting for cryptographic protocol verification. In *17th CADE*, pages 271–290. Springer Verlag LNCS 1831, 2000.
20. S. Ginsburg and E. H. Spanier. Semigroups, Presburger formulas and languages. *Pacific Journal of Mathematics*, 16(2):285–296, 1966.
21. M. T. Goodrich, R. Tamassia, and A. Schwerin. Implementation of an authenticated dictionary with skip lists and commutative hashing. In *DISCEX II*, volume 2, pages 68–82. IEEE Computer Society Press, 2001.
22. J. Goubault-Larrecq. A method for automatic cryptographic protocol verification. In *FMPPTA'2000, 15th IPDPS Workshops*, pages 977–984. Springer-Verlag LNCS 1800, 2000.
23. J. Goubault-Larrecq. Higher-order positive set constraints. In J. Bradfield, editor, *CSL'02*. Springer Verlag LNCS, 2002. To appear, available as LSV Research Report.
24. O. H. Ibarra, J. Su, Z. Dang, T. Bultan, and R. A. Kemmerer. Counter machines and verification problems. *Theoretical Computer Science*, 2001. To appear.
25. W. H. Joyner Jr. Resolution strategies as decision procedures. *Journal of the ACM*, 23(3):398–417, July 1976.
26. Y. Kaji, T. Fujiwara, and T. Kasami. Solving a unification problem under constrained substitutions using tree automata. *Journal of Symbolic Computation*, 23(1):79–117, 1997.
27. B. Kallick. A decision procedure based on the resolution method. *Information Processing*, 68:269–275, 1969.
28. P. Li. Pattern matching in trees. Master's Thesis CS-88-23, University of Waterloo, May 1988.
29. D. Lugiez. A good class of tree automata. Application to inductive theorem proving. In *ICALP'98*, pages 409–420. Springer-Verlag LNCS 1443, 1998.
30. D. Lugiez and J. L. Moysset. Tree automata help one to solve equational formulae in AC-theories. *Journal of Symbolic Computation*, 18(4):297–318, 1994.
31. J. Millen and G. Denker. CAPSL and muCAPSL. *Journal of Telecommunications and Information Technology*, 2002. To appear.
32. M. L. Minsky. Recursive unsolvability of Post's problem of "tag" and other topics in the theory of Turing machines. *Annals of Mathematics, Second Series*, 74(3):437–455, 1961.
33. D. Monniaux. Abstracting cryptographic protocols with tree automata. In *SAS'99*, pages 149–163. Springer-Verlag LNCS 1694, 1999.
34. J. Niehren and A. Podelski. Feature automata and recognizable sets of feature trees. In *TAPSOFT'93*, pages 356–375. Springer-Verlag LNCS 668, 1993.
35. H. Ohsaki. Beyond regularity: Equational tree automata for associative and commutative theories. In *CSL'01*, pages 539–553. Springer-Verlag LNCS 2142, 2001.
36. R. J. Parikh. On context-free languages. *Journal of the ACM*, 13(4):570–581, 1966.
37. N. Peltier. Tree automata and automated model building. *Fundamenta Informaticae*, 30(1):59–81, 1997.
38. J. A. Robinson and A. Voronkov, editors. *Handbook of Automated Reasoning*. North-Holland, 2001.
39. K. M. Schimpf and J. H. Gallier. Tree pushdown automata. *Journal of Computer and System Sciences*, 30(1):25–40, 1985.
40. R. Shostak. Deciding linear inequalities by computing loop residues. *Journal of the ACM*, 28(4):769–779, Oct. 1981.
41. M. Steiner, G. Tsudik, and M. Waidner. Key agreement in dynamic peer groups. *IEEE Transactions on Parallel and Distributed Systems*, 11(8):769–780, 2000.
42. W. Thomas. Automata on infinite objects. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, chapter 4, pages 133–191. Elsevier Science, 1990.