$ACID$-Unification, Rewrite Reachability and Set Constraints

Siva Anantharaman, Paliath Narendran, *, Michael Rusinowitch

LIFO - Orlans (France), University at Albany–SUNY (USA), LORIA - Nancy (France)

Abstract. E-unification problems are central in automated deduction. In this paper, we consider theories that are extensions of the well-known AC or ACU, such as $ACID$ and $ACUIH^C$, and relate their study to set constraints or reachability problems on Minsky machine configurations. We first show that unification modulo the theory $ACUIH^C$ – i.e., ACU plus idempotence and a set of commuting homomorphisms – is undecidable. We next consider the $ACID$-unification problem, i.e., unification modulo AC plus idempotence for a given symbol '+', and distributivity of a given '$*$' w.r.t. '+'. It is shown that this problem is undecidable if equations of associativity-commutativity, or just associativity, on '$*$' are added on to $ACID$; but if '$*$' satisfies no additional identities other than distributivity, $ACID$-unification is shown to be NEXPTIME-decidable, and DEXPTIME-hard. All these conclusions remain valid also for the $ACUID$-unification problem, that is when a unit element for '+' which is null for '$*$', is added on to $ACID$.

Keywords: E-Unification, Complexity, Rewrite reachability, Minsky machine, Post correspondence problem, Set constraints, Tree automata, Dag automata, Labeled dag automata.

1 Introduction

The unification problems for the theories AC ("Associativity-Commutativity"), ACI ("AC plus Idempotence") and ACU ("AC with Unit element") have been studied in full detail in the past. On the other hand the theories $ACID$ ("ACI plus one Distributive symbol") or $ACUID$ ("ACID plus a Unit element U for addition, null for distribution") serve as a basis for studying set-constraints, but to our knowledge the unification problems over $ACID$ or $ACUID$ have not yet been looked into in detail.

Now any ground term over an $AC(U)ID$-signature defines a homomorphism w.r.t. the AC-operator (because of the distributivity), and this is the reason why we shall first look at the unification problem over the theory $ACUIH^C$, i.e., $ACUI$ with a family of commuting homomorphisms. We show this problem to be undecidable. (*Note*: in the case of a family of *non-commuting* homomorphisms, the unification problem has been proved DEXPTIME-complete in [4].) Our proof goes via reducing a reachability problem for commutative word rewriting (with constraints) to $ACUIH^C$- unification. We then show that the halting problem of Minsky counter machines can be reduced to this reachability problem for commutative word rewriting (with constraints). These results constitute Section 2.

We deduce as a rather immediate consequence that the $ACID$-unification problem is undecidable when the symbol '$*$', distributive w.r.t. the given ACI-symbol '+', is also

* Research supported in part by NSF grant CCR-9712396 and ONR grant N00014-01-1-0430.

assumed AC; this undecidability holds whether or not the ACI-symbol '+' admits a unit element U. If '$*$' is assumed only associative in addition to being distributive over '+', then $ACID$-unification is still undecidable: this is established by reduction from the modified Post correspondence problem. In the case where '$*$' does not satisfy other laws than distributivity, $ACID$-unification can be formulated as a problem of solvability *in terms of finite non-empty sets* for a particular class of set constraints. Its decidability can as such be deduced for instance via the results of [12], although no complexity bound can be derived this way. All these constitute Section 3.

In Section 4 we show that $ACID$-unification is NEXPTIME-decidable, by an approach based on labeled dag automata. And we establish a DEXPTIME lower bound for this problem in Section 5, where all these conclusions are shown to remain valid also for the $ACUID$-unification problem, i.e. when a unit element $U = 0$ for '+' which is 'null' w.r.t. '$*$', is added on to $ACID$; or equivalently, when we admit empty set values in the solution for the set variables. It might be of interest to note that for the particular class of set constraints expressing the general $AC(U)ID$-unification problem, if the solutions can be arbitrary sets – i.e., possibly infinite, or empty – then satisfiability is known to be NEXPTIME-decidable (cf. [2]) and DEXPTIME-hard (cf. [9]).

In an appendix, we establish a few results on the closure properties of dag automata and labeled dag automata, and also give some results concerning their languages.

2 *ACUI* with Commuting Homomorphisms

An $ACUIH^C$-signature consists of an AC-symbol denoted by '+' and assumed idempotent, a unit element for this '+' (denoted '0'), a finite family of unary symbols with additive homomorphism relations on each symbol, and commutativity relations between these unary symbols. It is generally assumed that a homomorphism maps 0 to 0, so to have more than just one ground term, we also assume given finitely many free constants. The idempotent AC symbol '+' can be viewed in this section as the set-union operation. The set of ground terms over such a given signature $ACUIH^C$ will be denoted by $\mathcal{G}$; a ground term is thus a sum (or set) of *commutative* strings over the symbols representing the commuting homomorphisms each applied itself to a ground term. The unification problem over such a signature reduces to solving linear equations over unknowns, with commutative strings over the unary homomorphism symbols as coefficients. For instance, if f, g are homomorphism symbols and a a free constant, then the unification problem $f(f^2ga + gX) =?= gf(fa + fX)$ reduces to solving the "linear equation"

$$f^3ga + fgX = gf^2a + gf^2X$$

for the 'unknown' X in $\mathcal{G}$. (Note: parentheses for the arguments of unary symbols are sometimes omitted.) The substitution $\{X \leftarrow fa\}$ is a solution here. For a more systematic presentation on how to formulate such E-unification problems in terms of solving linear equations over appropriate semi-rings, the interested reader may consult [3], [15].

We show in this section that the $ACUIH^C$-unification problem with free constants is undecidable. The proof is from a reduction of the reachability problem for configurations

of Minsky's counter machines. We briefly outline the ideas behind. As we just saw, the problem is that of solving linear equations of the form:

$$l_1X_1 + \ldots + l_nX_n + V \;=\; r_1X_1 + \ldots + r_nX_n + U$$

where the $X_i, 1 \leq i \leq n$ are the 'unknowns', U, V are sets of ground terms, and l_i, r_i are commutative strings over the homomorphism symbols, that we will also call 'ground terms' or 'ground strings' with no risk of confusion. Without any loss of generality, we shall *assume that the unit element '0' for the ACI-symbol '+' is neither in the set U, nor in V*. To this unification problem we associate a set of rewrite rules $l_i \to r_i,\ 1 \leq i \leq n$ and utilise the usual notion of rewrite steps (denoted '$\Longrightarrow$') on ground terms via these rules. The first step in our proof is to show that - provided the transitive closure of '$\Longrightarrow$' has no cycles - the linear equation of the type above has a solution if and only if V is a set of elements reachable from elements of the set U for the relation '$\Longrightarrow^*$'. The second step will be a reduction of the halting problem of Minsky machines with 2 counters to this reachability problem.

2.1 $ACUIH^C$-Unification Problem vs Rewrite Reachability

Before establishing our first step, a notational convention: for any binary relation '$\Longrightarrow$' between terms, we denote as usual by '$\Longrightarrow^+$' and '$\Longrightarrow^*$' its transitive closure and reflexive transitive closure respectively. If u, v are terms such that $u \Longrightarrow^* v$, we will say that v is reachable or derivable from u (or that u derives v) for the relation '$\Longrightarrow$'. For any given pair of sets of terms $S1, S2$, we will write $S1 \models\!\!\Longrightarrow^* S2$ iff:

i) for any element $v \in S2$, there exists a $u \in S1$ such that $u \Longrightarrow^* v$;

ii) for any element $u \in S1$, there exists a $v \in S2$ such that $u \Longrightarrow^* v$.

We will then say: the set $S2$ is a set of *reachable* (or *derivable*) elements from the set $S1$, for the relation '$\Longrightarrow$' that we omit to mention if clear from the context.

Proposition 1 *With the above notation, let*

$$(\&) \quad l_1X_1 + \ldots + l_nX_n + V \;=\; r_1X_1 + \ldots + r_nX_n + U$$

be an $ACUIH^C$-unification problem, $\{l_i \to r_i, i = 1..n\}$ the associated ground rewrite system, and '$\Longrightarrow$' the corresponding rewrite relation. If the problem (&) is solvable, **and** *if the relation $\Longrightarrow$ is acyclic* (i.e., $\Longrightarrow^+$ is irreflexive)*, then $U \models\!\!\Longrightarrow^* V$.*

Proof. We reason by induction on the minimal 'measure' $|X_1| + \ldots + |X_n|$ w.r.t. solutions $(X_1, ..., X_n)$ of the problem (&), for any unification problem with the same associated rewrite system (here, as usual, $|S|$ denotes the cardinality of the set S). If $|X_1| + ... + |X_n| = 0$ then $U = V$, and we have the trivial base case. So suppose $U \neq V$.

Case (a): $U \not\subset V$.

Let w be a string in $U \setminus V$. Without loss of generality we assume $w \in l_1X_1$, so $w = l_1w_1$ for some $w_1 \in X_1$; then write $X_1 = w_1 + Y_1$ with $w_1 \notin Y_1$ (and $w_1 \neq 0$).

Thus $r_1X_1 = r_1w_1 + r_1Y_1$ and $(Y_1, X_2, ..., X_n)$ has a strictly smaller measure than $(X_1, ..., X_n)$ and satisfies the unification problem:

$$l_1Y_1 + ... + l_nX_n + (V + l_1w_1) = r_1Y_1 + ... + r_nX_n + (U + r_1w_1)$$

By induction hypothesis therefore we have: $(U + r_1w_1) \models\!\!\Longrightarrow^* (V + l_1w_1)$.

This means that (i) every term in V is reachable from some term in $U + r_1w_1$ and hence from some term in $U + l_1w_1 = U$, and (ii) every term in U can derive some term in $V + l_1w_1$. Furthermore, r_1w_1 must derive elements in V, since $\Longrightarrow$ is assumed acyclic. Thus every term in U derives some term in V.

Case (b): $U \subset V$.

Let x be a term in V not in U; without loss of generality we assume $x \in r_1X_1$, i.e., $x = r_1x_1$ for some $x_1 \in X_1$. Let then $X_1 = x_1 + Z_1$ with $x_1 \notin Z_1$ (and $x_1 \neq 0$).

Thus $l_1X_1 = l_1x_1 + l_1Z_1$; now $(Z_1, X_2, ..., X_n)$ has a strictly smaller measure than $(X_1, ..., X_n)$ and satisfies the unification problem:

$$l_1Z_1 + ... + l_nX_n + (V + l_1x_1) = r_1Z_1 + ... + r_nX_n + (U + r_1x_1)$$

Again, by induction hypothesis we have $(U + r_1x_1) \models\!\!\Longrightarrow^* (V + l_1x_1)$. Now, since the relation $\Longrightarrow$ is assumed acyclic, it cannot be that $r_1x_1 \Longrightarrow^* l_1x_1$, so there must be some string in U *other than* r_1x_1 which reduces to l_1x_1, and hence to r_1x_1 as well. This implies then that any element of V is reachable from elements of U. Since U is a subset of V, the result follows. □

Remark 1. Let us point out that the above proposition ceases to be true if the rewrite relation is not acyclic. Here is a counter-example. We suppose given two commuting homomorphisms, and consider the following unification problem:

$$l_1X_1 + l_2X_2 + a^3b^3 + a^4b^2 + a^3b^2 = r_1X_1 + r_2X_2 + a^3b^3$$

where a, b are two homomorphism symbols, $l_1 = a^2b, r_1 = ab$ and $l_2 = a^2b^2, r_2 = a^3b^2$; the strings on a, b with no following variables are all assumed evaluated at some fixed constant. This problem does have a solution, namely: $X_1 = a^2b, X_2 = a$ (since '+' is idempotent). But neither a^4b^2 nor a^3b^2 can be reached from a^3b^3 via the rewrite rules. This is because each rule consuming a power of b brings back exactly the same power of b.

Proposition 2 *Let*

$$(\&) \quad l_1X_1 + \ldots + l_nX_n + V = r_1X_1 + \ldots + r_nX_n + U$$

be an $ACUIH^C$-unification problem, $\{l_i \to r_i, i = 1..n\}$ the associated ground rewrite system, '$\Longrightarrow$' the corresponding rewrite relation. Then the unification problem (&) is solvable **if** $U \models\!\!\Longrightarrow^* V$.

Proof. Let us define the *'distance'* between U and V (w.r.t. the relation $\Longrightarrow$) as the sum of the (minimal) number of rewrite steps needed for deriving the elements of V from the elements of U. We shall reason by induction on this distance. The case of distance 0 corresponds to the trivial base case $U = V$. So assume that this distance d is non-zero;

then there exist $u \in U$ and $v \in V$ such that $u \Longrightarrow^+ v$; without loss of generality we assume that $u = l_1 u'$ and $r_1 u' \Longrightarrow^* v$.

Now write $U = u' + U'$ with $u' \notin U'$; then the sets $(r_1 u' + U')$ and V are such that $(r_1 u' + U') \models\!\!\Longrightarrow^* V$, and the distance (w.r.t. $\Longrightarrow$) from $(r_1 u' + U')$ to V is strictly smaller than d. So, by induction hypothesis there must exist sets $X'_1, \ldots, X'_n$ such that

$$l_1 X'_1 + \ldots + l_n X'_n + V = r_1 X'_1 + \ldots + r_n X'_n + (r_1 u' + U')$$

Now, setting $X_1 = X'_1 + u'$ and $X_j = X'_j$ for all $j > 1$, we get a solution for (&). □

Putting these two propositions together, we get

Theorem 1. *Let*

$$(\&) \quad l_1 X_1 + \ldots + l_n X_n + V = r_1 X_1 + \ldots + r_n X_n + U$$

be a given $ACUIH^C$-unification problem, $\{l_i \to r_i, i = 1..n\}$ the associated ground rewrite system, and '$\Longrightarrow$' the corresponding rewrite relation. If the relation $\Longrightarrow$ is acyclic, then the problem (&) is solvable if and only if $U \models\!\!\Longrightarrow^* V$.

Corollary 1.1: Let $a_1, \ldots, a_m$ be commuting homomorphisms. Then the equation

$$a_1 X_1 + \ldots + a_m X_m + \epsilon = X_1 + \ldots + X_m + W$$

is solvable if and only if W consists of strings over the $a_i, i = 1..m$, evaluated on some constants. □

In particular if a, b, c are commuting homomorphisms, then the equation: $a(x_1)+b(x_2)+\epsilon = x_1 + x_2 + W$ forces W to consist entirely of terms not containing c.

2.2 $ACUIH^C$-Unification in Terms of Admissible Reachability

Definition 1 *A marked rewrite rule over the set $\mathcal{G}$ is a rule of the form $l \to r\ [a]$, where the* mark *a is a given ground term, and l, r are strings such that l does* not contain the *mark a* as a factor. *A ground term not having the mark a as a factor is said to be* compatible *with the marked rewrite rule $l \to r\ [a]$. A rewrite step using the rewrite rule $l \to r\ [a]$ is said to be* admissible *iff the string rewritten under this step is compatible with the rule.*

A rewrite rule with no attached mark may be seen as marked with the empty string; any ground string is compatible with such a rewrite rule. Given a set of marked rewrite rules over $\mathcal{G}$, we shall be henceforth denoting by $\Longrightarrow$ the associated relation of admissible reduction, and by $\models\!\!\Longrightarrow^*$ the relation of *admissible reachability* (or derivability) between sets of terms, defined as in the previous subsection but here w.r.t. this restricted notion $\Longrightarrow$ of *admissible reduction.*

Theorem 2. *Let*

$$(\&) \quad l_1 X_1 + \ldots + l_n X_n + V = r_1 X_1 + \ldots + r_n X_n + U$$

be a given $ACUIH^C$-unification problem. Let $\{l_i \to r_i\}$ be the associated rewrite system, each rule respectively marked *with either a constant $a_i, i = 1..n$, or the empty string; and let '$\Longrightarrow$' be the associated* admissible rewrite *relation, that we assume to be acyclic. We consider the following constraint on the X_i :*

($\mathcal{C}$) if $l_i \to r_i$ is marked with a_i, then
X_i does not contain terms in which the mark a_i occurs.

Then the the problem (&) is solvable under constraint $\mathcal{C}$ if and only if *we have $U \models\!\!\Longrightarrow^* V$.*

Proof. The proof of Theorem 1 based on propositions 1, 2, goes through verbatim; we mention here the points to check, for completeness. (We are assuming each rewrite rule $l_i \to r_r$ is marked with an a_i, which is a constant or the empty string.)

Proposition 1, case (a): if $(X_1, \ldots, X_n)$ is a solution satisfying constraint $\mathcal{C}$, $w = l_1 w_1$, and $X_1 = w_1 + Y_1$, then $(Y_1, X_2, \ldots, X_n)$ continues to satisfy the constraint $\mathcal{C}$, so induction goes through. The same reasoning holds also for case (b).

Proposition 2: the distance from U to V is defined now over admissible rewrite steps. If $u = l_1 u' \Longrightarrow r_1 u'$ admissibly, then u' doesn't contain the mark a_1 on the rule $l_1 \to r_1\,[a_1]$; so if $(X'_1, X'_2, \ldots, X'_n)$ is a solution satisying constraint $\mathcal{C}$ for the linear equation at the inductive step, then the solution $(X'_1 + u', X'_2, \ldots, X'_n)$ for the initial equation also satisfies the constraint $\mathcal{C}$. □

We will show in the next section that *admissible reachability is undecidable*. From this and the above theorem, it will follow that solving an $ACUIH^C$-linear equation (&) with marked associated rewrite rules, and under constraint $\mathcal{C}$, is undecidable. But it follows from Corollary 1.1 that the constraint $\mathcal{C}$ is itself expressible in terms of linear equations over the $ACUIH^C$-signature: indeed the equation

$$a_1 X_1 + \ldots + a_{j-1} X_{j-1} + \epsilon + a_{j+1} X_{j+1} + \ldots + a_n X_n = X_1 + X_2 + \ldots + X_n$$

forces the component X_j of the solution to be free from terms containing a_j, if the marks a_i are all distinct. In other words, we will be able to deduce the undecidability of the $ACUIH^C$-unification problem.

2.3 Minsky Counter Machines and Admissible Reachability

A Minsky machine with two counters C_1, C_2 storing non-negative integer values, executes *programs* which are finite lists of *instructions* labeled with the natural numbers from 1 to L, each of which has one of the following forms (where $l \in \{1, \ldots, L-1\}$, $l' \in \{1, \ldots, L\}$, $l \neq l'$, and i is 1 *or* 2):

(i) l: ADD 1 to C_i and GOTO $l + 1$;

(ii) l: If $C_i \neq 0$
then SUBTRACT 1 from C_i and GOTO $l + 1$;
else GOTO l';

(iii) L: STOP.

We shall agree that in any given program $\mathcal{P}$ there is exactly one instruction 'STOP' labeled L. A *configuration* of such a 2-counter machine, at any given stage of a computation following such a program, can then be defined as a triple (l, C_1, C_2) where l is the (label of the) next instruction to execute, and $C_i, i = 1, 2$, are the current integer values of the two counters. A configuration of the form (L, xxx, yyy) (resp. $(1, xxx, yyy)$) will be labeled *final* (resp. *start* or *initial*). A program by definition *starts* on an initial configuration, and *halts*, by definition again, if it reaches a configuration which is final (in which case, it has no more instructions to execute). The following result on Minsky's 2-counter machines is classical, and will well serve our purposes, after some minor adaptations.

Theorem 3. *(Minsky[14]) For every partial recursive function f on natural numbers there exists a program $\mathcal{Q}$ such that applied with $(1, 2^d, 0)$ as starting configuration, $\mathcal{Q}$ halts with the final configuration $(L, 2^{f(d)}, 0)$ if $f(d)$ is defined on d, and does* not *halt otherwise.*

In particular it is undecidable in general if an arbitrarily chosen program $\mathcal{P}$ will halt, when applied with an arbitrarily given starting configuration $(1, 2^d, 0)$.

We introduce a homomorphism symbol h_l for every instruction $l \in 1..L$; add two more homomorphisms $h_{c_i}, i = 1, 2$, for the two counters. Any given machine configuration can then be seen as a (commutative) string $h_l\, h_{c_1}^{m_1}\, h_{c_2}^{m_2}$, if l is the current instruction to execute and $m_i, i = 1, 2$, are the current (non-negative) integer values of the two counters. We then associate the following system of *marked* rewrite rules to each type of machine instructions:

i) (increment C_1 or C_2) For any $j \in 1..(L-1)$,
$h_j \to h_{c_1} h_{j+1}$, $\quad h_j \to h_{j+1} h_{c_2}$

ii) (conditional subtraction on C_i) For any $j \in 1..(L-1)$,
$h_{c_1} h_j \to h_{j+1}$, $\quad h_j h_{c_2} \to h_{j+1}$

ii') (counter C_1 or C_2 must be empty) For any $j \in 1..(L-1)$, $k \in 1..L$, with $j \neq k$,
$h_j \to h_k\ [h_{c_1}]$, $\quad h_j \to h_k\ [h_{c_2}]$

Any given instruction transforms the current configuration-string by applying a unique well-determined rewrite rule from this set. Note that the last two sets of rewrite rules are *marked*, and therefore can be applied for reduction only on strings which are compatible: the first among these can rewrite only strings with no first counter, and the second only the strings with no second counter. Also note that any sequence of admissible rewrite steps starting from the configuration string $h_1\, h_{c_1}^{2^d}$ corresponds to a (correct) sequence of machine instructions and vice versa. The final statement of the above theorem of Minsky on 2-counter machines can then be formulated as: Given two arbitrary strings $s = h_1\, h_{c_1}^{2^d}$ and $f = h_L\, h_{c_1}^{2^m}$ over our $ACUIH^C$-signature, it is undecidable in general if the latter is reachable from the former via admissible rewrite steps using the marked rules of the above rewrite system.

The proof that $ACUIH^C$-unification is undecidable is almost complete now, thanks to the result of our previous section; remains the hitch that this result made an assumption

of acyclicity on the rewrite relation, and this does not hold for the above rewrite system. In order to get the property we introduce a *third* counter C_3. The idea is that initially counter C_3 has value 1, that every instruction of type (i) or (ii) also increments C_3 by 2, instruction of type (iii) increments C_3 by 1 and goes to a new instruction where the only thing done is decrementing C_3 by 2 as long as possible, until it stops. In other words, our Minsky machine with *three* counters $C_i, i = 1..3$, storing non-negative integer values, executes *programs* which are finite lists of at least three *instructions* labeled with the natural numbers from 1 to $L+2$, each of which has one of the following forms (where $l \in \{1, \ldots, L-1\}$, $l' \in \{1, \ldots, L\}$, $l \neq l'$, and i is 1 *or* 2):

(i) l: ADD 1 to C_i, ADD 2 to C_3, and GOTO $l+1$;

(ii) l: If $C_i \neq 0$
then SUBTRACT 1 from C_i, ADD 2 to C_3, and GOTO $l+1$;
else ADD 2 to C_3 and GOTO l';

(iii) L: PAUSE; ADD 1 to C_3 and GOTO L+1;

(iv) L+1: IF $C_3 > 2$
then DECREMENT C_3 by 2 and GOTO $L+1$;
else GOTO $L+2$;

(v) L+2: STOP.

Such a program $\mathcal{P}'$ is said to be extended (in a natural manner) from a 2-counter machine program $\mathcal{P}$ as described earlier. We assume that any such program $\mathcal{P}'$ has exactly one instruction 'STOP' labeled $L+2$. Following the same lines of thinking as above, we can visualize these 3-counter machine programs as executions of rewrite steps applying rules from the following system:

i) (increment C_1 or C_2) For any $j \in 1..(L-1)$,
$h_j \to h_{c_1} h_{j+1} h_{c_3}^2$, $h_j \to h_{j+1} h_{c_2} h_{c_3}^2$

ii) (conditional subtraction on C_i) For any $j \in 1..(L-1)$,
$h_{c_1} h_j \to h_{j+1} h_{c_3}^2$, $h_j h_{c_2} \to h_{j+1} h_{c_3}^2$

ii') (counter C_1 or C_2 must be empty) For any $j \in 1..(L-1)$, $k \in 1..L$, with $j \neq k$,
$h_j \to h_k h_{c_3}^2\ [h_{c_1}]$, $h_j \to h_k\, h_{c_3}^2\ [h_{c_2}]$

iii) (pause) $h_L \to h_{L+1}\, h_{c_3}$

iv) (conditional decrement on C_3, or STOP)
$h_{L+1}\, h_{c_3}^2 \to h_{L+1}$, $h_{L+1} \to h_{L+2}\ [h_{c_3}]$

It is immediate that this rewrite system is acyclic, and that no right-hand-side is a substring of any of the marks (note: the marks here are h_{c_1}, h_{c_2} and h_{c_3}). The following fact is equally obvious: Our earlier 2-counter machine run on a program $\mathcal{P}$ starting from an initial configuration $h_1\, h_{c_1}^{2^d}$ halts, with as final configuration a string $h_L\, h_{c_1}^{2^m}$, **if and only if** our current 3-counter machine run on the extended program $\mathcal{P}'$ from the initial configuration $h_1\, h_{c_1}^{2^d}\, h_{c_3}$ halts, with final configuration $h_{L+2}\, h_{c_1}^{2^m}$.

We can now formulate our undecidability result, whose proof is a direct consequence of our second theorem 2 combined with the above reformulation of Minsky's theorem in terms of a 3-counter machine.

Theorem 4. *The $ACUIH^C$-unification problem is undecidable.*

Remark 2. It is immediate from the above lines of proof that the unit element '0' for the AC-symbol '+' has played no special role in the reasoning. In other words, we may also conclude that: *the $ACIH^C$-unification problem is undecidable.*

Remark 3. It might be of interest to observe here that the way in which we have modeled the functioning of a 2-counter Minsky machine in terms of a marked rewrite system (on commutative strings) "revisits" known proofs for the following well-known result: Reachability in a Petri net with two (or more) inhibitor arcs is undecidable (cf. [13]).

3 The *ACID*-Unification Problem

We are interested here in the unification problem w.r.t. the following theory that we shall denote $ACID$:

$$\begin{aligned} x + x &\approx x \\ x * (y + z) &\approx (x * y) + (x * z) \\ (u + v) * w &\approx (u * w) + (v * w) \end{aligned}$$

This set of equations can be converted naturally to a convergent rewrite system; every ground term in normal form (over any given set of free constants) w.r.t. this system can be viewed as a *set* of terms over '$*$' and the constants: indeed '+' can be viewed as set union. An ACID-unification problem with free constants is that of solving modulo the above equational theory, a family of equations of the form: $\{s_1 =^? t_1, \ldots, s_k =^? t_k\}$, where the terms in the equations and/or the solutions can involve the given constants.

3.1 Undecidability if '*' is in addition AC or Associative

In this section we prove that the $ACID$-unification problem is undecidable if we assume that the symbol '$*$' is in addition assumed either AC, or associative. The AC-case follows from our previous results on $ACIH^C$-unification:

Theorem 5. *If the symbol '$*$' above is also assumed AC (i.e., if the equations of associativity and commutativity on '$*$' are added on to ACID), then the ACID-unification problem is undecidable.*

Proof. Given finitely many free constants $\{a, b, c, ...\}$, let $h_a, h_b, h_c, ...$ denote the homomorphisms they define in a natural manner over the term algebra: $h_a(x) = a * x$ etc.. Since we are assuming the symbol '$*$' to be AC, these homomorphisms *commute*. (*Note*: it does **not** suffice to assume just the commutativity of '$*$' here.) Consider then any arbitrarily given family of equations of the form:

$$l_1\, X_1 + \ldots + l_p\, X_p + V = r_1\, X_1 + \ldots + r_p\, X_p + U$$

where the l_i, r_i are '*'-commutative strings over these homomorphisms, and V, U are sets of ground terms over the given constants. Solving this family of equations amounts to solving the $ACIH^C$-unification problem expressed in terms of such linear equations. And we saw in the previous section that $ACIH^C$-unification is undecidable. □

Theorem 6. *If the symbol '$*$' is also assumed associative (i.e., if the equations of associativity on '$*$' are added on to ACID), then the ACID-unification problem is undecidable.*

Proof. The proof is via reduction from the so-called modified Post correspondence problem (MPCP). Recall that the usual formulation of MPCP goes as follows: Given a list of pairs $\{(w_i, w_i')\}_{i=0..n}$ of non-empty (finite) strings over some alphabet Σ, it is undecidable in general to determine whether there exist indices $i_1, \ldots, i_k$ with $0 \leq i_j \leq n$ for all $j = 1..k$, such that: $w_0 w_{i_1}\, w_{i_2}\, \ldots\, w_{i_k} = w_0' w_{i_1}'\, w_{i_2}\, \ldots\, w_{i_k}'$. We first point out that this problem will still remain undecidable if we add the condition that each word w_i (resp. w_i') must appear at least once in a solution. It is such a formulation of MPCP that we propose to code as an ACID-unification problem.

Let $(w_0, w_0'), \ldots .(w_n, w_n')$ be such an instance of MPCP. We then consider the $ACID$-unification problem defined by the following two equations:

$$(1) \qquad w_1 * X_1 * \overline{1} + \ldots + w_n * X_n * \overline{n} + w_0 * \# \;=\; X_1 + X_2 + ... + X_n + V$$
$$(2) \qquad w_1' * Z_1 * \overline{1} + \ldots + w_n' * Z_n * \overline{n} + w_0' * \# \;=\; Z_1 + Z_2 + ... + Z_n + V$$

where the unknowns are the X_i, Z_i and V, the $\#, \overline{1}, \overline{2}, \ldots, \overline{n}$ are some fixed new symbols. Assume this $ACID$-unification problem to be solvable.

We then look at the solution for equation (1), and observe that: words of maximal length over the rhs of (the solution for) this equation must come from the set V, and can be in no instance of any X_j over the rhs: indeed, if v is a word from an X_j, then $w_j * v * \overline{j}$ will be an element of some set over the lhs, so must also be an element over the rhs, and necessarily of bigger length than the word v.

Let then $v \in V$ be a word of maximal length over the rhs of (1). If v is not equal to $w_0 * \#$, then it has to be of the form $w_i * w * \overline{i}$ for some unique i. We may suppose $i = 1$, write $v = w_1 * w * \overline{1}$ and set $X_1 = X_1' + w$, for a $w \in X_1$, $w \notin X_1'$. Then v has a unique occurrence in both the lhs and rhs of the equation, so we can cancel it from each side. After such a cancellation, we get an equality of the form:

$$w_1 * X_1' * \overline{1} + \ldots + w_n * X_n * \overline{n} + w_0 * \# \;=\; X_1' + X_2 + ... + X_n + V'$$

where $V' = (V \setminus \{w_1 * w * \overline{1}\}) + w$.

Now (the solution for) V' is smaller than (that of) V for the multiset ordering over the lengths of words. Thus we can apply an inductive argument to deduce that each element of V has to be a word of the form: $w_{i_1} * \ldots * w_{i_n} * w_0 * \# * s$, where s is some string over the symbols $\overline{1}, \ldots, \overline{n}$. A similar reasoning with equation (2) leads us then to the assertion that

each such element of V must also be, at the same time, of the form: $w'_{i_1} * \ldots * w'_{i_n} * w'_0 * \# * s$, with the same sequence of indices for the w'_i as for the w_i, because of the new symbols introduced $\overline{i}$.

In other words, if our above $ACID$-unification problem were solvable (with non-empty set values for the unknowns), then the solution for V contains a solution for the instance of MPCP we started with. Conversely from a solution of MPCP – with the condition that each w_i, w'_i appears in the solution – we can build a solution to the $ACID$-unification problem defined by the above three equations (1), (2), (3), in terms of non-empty sets of terms (over '$*$' and the constants). □

3.2 The General Case

We shall be assuming henceforth that our theory $ACID$ contains no further equations involving the symbol '$*$'; and that our $ACID$-unification problem (after some normalization and decomposition steps) is in a *standard form*, i.e., to say, every equation in our problem has one of the following forms (respectively referred to as of type *'product'*, *'sum'*, or *'constant'*):

$$x =^? y * z, \quad u =^? v + w, \quad u =^? c$$

where u, v, w, x, y, z are variables and c is any constant. A given $ACID$-unification problem can be reduced to a standard form in more than one manner. Since '$+$' is idempotent and '$*$' distributes left and right over '$+$', we may view this $ACID$-unification problem as a set constraint problem; e.g. in the first case, if y and z are interpreted as sets of terms over $*$ and the constants, then $y * z = \{s * t \mid s \in y,\ t \in z\}$. Actually the set constraints in this context are *with union* only, following the terminology introduced in [9]; with the additional restriction that *all sets must be finite, and non-empty* (since '$+$' has no unit element in the theory).

The problem of satisfiability of set constraints in general, i.e., allowing arbitrary sets in the solutions, has been studied intensively over the past decade, in particular in [1, 2, 7, 10, 12, 8, 9], although not all known positive results give a complexity estimate. However very few results seem to be known for solvability in terms of finite sets, or finite non-empty sets. The only result we actually know of is very general, and is based on the Σ-graph automata of Gilleron, Tison and Tommasi (cf. Proposition 14, [12]); by appealing to this result, we may formulate our next result (without being able to give a complexity bound):

Proposition 1. *The $ACID$-unification problem is decidable if the symbol '$*$' is involved in* no other equations *except those of distributivity w.r.t. '$+$'.*

Actually we propose to show in the following section that this problem is NEXPTIME-decidable, via an approach based on the formalism of dag automata ([8]), enriched with labels at its nodes in order to serve our purposes.

4 *ACID*-unification via Labeled Dag Automata

We first "pre-process" our unification problem, eliminating some easy cases thereby.

4.1 Pre-Processing an *ACID*-Unification Problem

The pre-processing of our $ACID$-unification problem (in a standard form) is based on a graph called its associated *dependency graph*; the arcs of the graph will be called *dependency arcs*. Here is how this graph is constructed.

The nodes of our graph will be the variables $\{x, y, z, \ldots\}$ appearing in our equations, arranged in an ordered list; the edges of the graph are directed and tentatively labeled by one among the two symbols '$\{*, \supset\}$' as follows. For any two variables u, v in the list, we draw a directed :

- edge from node u to node v and labeled with '$*$' iff there is an equation of the form $u =^? v * w$ in our $ACID$-problem,

- edge from node u to node v and labeled with '$\supset$' iff there is an equation of the form $u =^? v + w$ in our $ACID$-problem.

Pruning: We first get rid of the variables which are redundant in the problem: those defining the same set of ground terms as a variable appearing earlier in the list, as follows: If there is a directed path in our graph from u to v, and also a directed path from v to u, such that all the edges concerned all along are labeled *only* with '$\supset$', and u precedes v in the list, then replace v by u in all the equations of our problem; suppress multiple occurrences among equations, if any, after such a modification; repeat the pruning operation if appropriate.

Occur-Check: Return '*OC-Fail*' iff:

- there is a (directed) loop from a node u to itself on the graph, with at least one edge along the loop labeled with '$*$';
- or, there is a directed path from u (to some node v) at least one edge of which is labeled with '$*$', and also an equation of the form $u =^? a$ in the $ACID$-problem,
- or, if the $ACID$-problem contains two different equations of the form $u =^? a$, $u =^? b$.

Lemma 1. *Pruning and testing for* Occur-Check *on any ACID-unification problem (given in standard form) can be performed in time NP w.r.t. to the number of equations in the problem. If the problem is pruned, and if Occur-Check does not return '*OC-Fail*', then the directed graph associated to the problem contains no loops.*

The proof of this lemma being obvious is omitted. Pruning and Occur-Check will play a useful role in our algorithm.

Example 1. Consider the following two distinct problems:

(1) $\quad x =^? y * z, \quad y =^? a + x$

(2) $\quad x =^? a + y, \quad y =^? z + t, \quad t =^? b + x, \quad z =^? c$

Problem $n^o 1$ is pruned, but Occur-Check returns 'OC-Fail'. Problem $n^o 2$ is not pruned; successive pruning leads to the pruned problem below:

(2′) $\quad x =^? a + x, \quad x =^? z + x, \quad z =^? c$ □

We shall be assuming henceforth that our $ACID$-unification problem in standard form has been pruned, and has successfully passed the Occur-Check test (*we shall just say "pruned"*, in the sequel). The associated dependency graph is then loop-free. We shall construct in the following section a labeled dag automaton the language of which will correspond to the set of solutions of the given unification problem.

4.2 Labeled Term Dags and Labeled Dag Automata

Let us recall briefly the notions of term dags and of dag automata as developed in [8]. A *term dag* (or *t-dag*) over a ranked alphabet Σ is a rooted dag where each node has a symbol from Σ such that:

(i) the outdegree of the node is the same as the rank of the symbol,

(ii) edges going out of a node are ordered, and

(iii) no two distinct subgraphs are isomorphic.

Every node represents a unique term in a t-dag, so we often treat "node" and "term" as synonymous on a t-dag.

Definition 2 *A t-dag automaton (or dag automaton, "DA" for short) over a ranked alphabet Σ is a tuple (Σ, Q, F, Δ), where Q is a finite set of states, $F \subset Q$ is the set of final (or accepting) states, and Δ is a set of transition rules of the form : $f(q_1, q_2, ..., q_n) \rightarrow q$, where $f \in \Sigma$ is of arity of n, and the $q_i, 1 \leq i \leq n, q$ are in Q.*

A *run* r of a DA $\mathcal{A} = (\Sigma, Q, F, \delta)$ on a t-dag t is a mapping from the set of nodes of t to the set of states Q of DA, that respects the transition relation δ, i.e., for every node u, if the symbol at u is f of arity k, then $f(r(u_1), \ldots, r(u_k)) \rightarrow r(u)$ must be a transition in δ, where u_1, ..., u_k are the successor-nodes of u given in order. A run r is *successful* on t if and only if $r(t) \in F$, i.e, it maps the root node to an accepting state. A t-dag t is accepted by a DA if and only if there is a successful run on t. (Note that runs are defined bottom-up.) The 'language' of a DA is the set of all t-dags that it accepts. It is easy to show that the class of t-dag automata is stable under union and intersection (cf. the Appendix), but emptiness is a much harder problem; that emptiness of a DA *is* NP-decidable is proved in [8].

A *labeled term dag*, or lt-dag, is a term dag equipped additionally with a mapping from the nodes of the dag to a given set of labels E. A motivation for adding labels is e.g. the fact that when the labels are boolean, i.e., $E = \{0, 1\}$, a labeled term dag can be used to specify finite sets of terms. For instance, the labeled term dag in Figure 1 can represent the set of terms: $\{a, g(g(a, a), b)\}$. (But for what concerns us in this paper, we shall be making use of labels on term dags with a different objective; cf. the following section.)

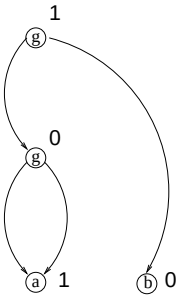


Fig. 1. A labeled term dag

A labeled dag automaton ("LDA") is a quintuple $(\Sigma, Q, F, \Delta, E)$, where the first four components form a DA, E is a finite set of *labels*, and the transition relation Δ consists of *labeled* rewrite rules, of the form: $f(q_1, \ldots, q_k) \xrightarrow{l} q$, where k is the arity of f, and l is a label from E, and the q_i, q are elements of Q. A *run* r of an LDA $(\Sigma, Q, F, \Delta, E)$ on an lt-dag t with label set E is a mapping from the nodes of t to Q that respects the labels and the transition relation Δ in the following sense: for every node u on t, if the symbol at u is f of arity k, and the label of t at u is l, then transitions are possible via rules in Δ of the form $f(r(u_1), \ldots, r(u_k)) \xrightarrow{l} r(u)$, where u_1, ..., u_k are the successor-nodes of u on t given in order. we may paraphrase this condition as: "the label on an LDA-transition must tally with the one at the node reached". A run r is said to be *pseudo-successful* on t if and only if $r(t) \in F$, i.e, it maps the root node to an accepting state.

An lt-dag t is said to be pseudo-accepted by an LDA if and only if there is a pseudo-successful run on t. The 'language' of an LDA is the set of all lt-dags that it pseudo-accepts. That its emptiness is NP-decidable for any given LDA is a direct consequence of the NP-decidability of emptiness of the language for any given DA. This fact will be used in the coming section where we will be defining *acceptance* of a run in a more precise sense, by taking a closer look at the labels on both the lt-dag and the LDA.

4.3 The LDA associated to an *ACID*-Unification Problem

Suppose given an $ACID$-unification problem (pruned), and denote by $\{X_i\}_{i=1..n}$ the set of its (set-)variables. The LDA that we associate to this problem has some similarities to what was called tree automata with free variables in [11]; the differences stem from the fact that we look for solutions for the X_i in terms of *finite and non-empty* sets. To achieve such a goal, (in a manner somewhat similar to that of [17]) we shall work with two sets of valuations (i.e., boolean valued functions) on the list of set-expressions $\{X_i\}_{i=1..n}$; the first set for testing the satisfiability of the constraints on the variable X_i, and the second for testing X_i for non-emptiness. In other words, the states of our LDA will be essentially the set of all vectors of length $2n$ over (the boolean values) $\{0, 1\}$. Typically then, a state on our LDA can be denoted by a vector of the form $(l_1, l_2, \ldots, l_n; h_1, h_2, \ldots, h_n)$, with $l_i, h_i \in \{0, 1\}$; and for reasons which will become clear shortly, we only retain such vectors

verifying the further assumption that $l_i \leq h_i$ for all $i = 1..n$. Before formally defining the LDA along these lines, here are a few words by way of explanation.

The lt-dags on which the runs of this LDA will be executed are arbitrary terms over the symbol '$*$' and the given constants in the signature, with n-bit labels at their various nodes; labels on these lt-dags will be denoted by $(l_1, l_2, \ldots, l_n)$, with $l_i \in \{0, 1\}$. The labels of the various nodes on the lt-dags will be so chosen that $l_i = 1$ is meant to signify: *the sub-term of the lt-dag at the current node is in the set* X_i.

The transition rules on the LDA will also be labeled therefore by the n-bits $(l_1, l_2, \ldots, l_n)$. The 'lower i-th bit' l_i of any state on the LDA is meant to signify (when 1) that the term *at* the current node is accepted by the given run as element of the set X_i; while its 'upper i-th bit' h_i is meant to signify (when 1) that the some sub-term *below* the current node has been accepted in the set X_i by the run. (This explains the assumption $l_i \leq h_i$ for $1 \leq i \leq n$, made above.) The successful runs will naturally be defined in such a way that the given set constraints must be satisfied at every node; the accepting states on the LDA will thus have the property that $h_i = 1$ for all i; we shall be more restrictive below however, since the notion of acceptance of an lt-dag by the LDA under a run is more delicate to define.

Definition 3 *i) Let* $l = (l_1, l_2, \ldots, l_n)$ *any n-bit vector. Then l is said to be a* permitted label for constants *w.r.t.* $\mathcal{P}$ *(in short: "*permitted*")* **iff***:*

$l_k = 0$ *for any k such that there is a 'product'-equation* $X_k = X_i * X_j$ *in the problem* $\mathcal{P}$.

ii) For any $2n$*-bit vector* $A = (l_1, l_2, \ldots, l_n; h_1, h_2, \ldots, h_n)$ *we shall denote by A.l and A.h the n-bit vectors which constitute respectively its left- and right- half.*

Definition 4 *Let* $\{X_i\}_{i=1..n}$ *be the list of all variables in the ACID-unification problem* $\mathcal{P}$ *(in a standard form, and pruned). The LDA associated to this problem is the labeled dag automaton* $(\Sigma, Q, F, \Delta, E)$ *such that:*

- Σ *is the signature of the problem* $\mathcal{P}$,
- *The set of states* Q *is the set of all* $2n$*-bit vectors* $(l_1, l_2, \ldots, l_n; h_1, h_2, \ldots, h_n)$ *satisfying the following conditions:*
  - $l_i \leq h_i$ *for all i, and*
  - $l_k = l_i \vee l_j$ *for* every *sum-equation* $X_k = X_i + X_j$ *in the problem* $\mathcal{P}$;
- *The set* F *of* `accepting states` *consists of states* C *in* Q *such that:*
  - $C.h_i = 1$ *for all* $i = 1..n$;
  - *if k is an integer such that* $C.l_k = 1$ *and there exists an equation* $X_k = X_i * X_j$ *in the problem* $\mathcal{P}$, *then* $C.l_i = 0 = C.l_j$.
- *The set of labels* E *is the set of all n-bit vectors* $(l_1, l_2, \ldots, l_n)$;
- (`progression`) *For any* $A, B, C \in Q$ *and any label* $m = (m_1, m_2, \ldots, m_n)$, *there is a transition rule in* Δ *with label* m *of the form* $A * B \xrightarrow{m} C$, **if and only if***:*
  - $m = C.l$ *('label on the LDA-transition must tally with the one at the node reached'),*
  - *for every equation in* $\mathcal{P}$ *of the form* $X_k = X_i + X_j$, *we have* $m_k = m_i \vee m_j$;

- *for* every *equation in* $\mathcal{P}$ *of the form* $X_k = X_i * X_j$, *we have:*
  $C.l_k = A.l_i \wedge B.l_j$, *and*
- $C.h_i = C.l_i \vee (A.h_i \vee B.h_i)$ *for* all i.

- `(initialization)` *For any constant* $a \in \Sigma$ *and any* permitted *label* m, *there are transitions in* Δ *of the form* $a \xrightarrow{m} A$, *for every* $A \in Q$ *such that* $A.l \leq m$. *If* $\mathcal{P}$ *contains an equation of the form* $X_k = a$, *then the label* m *must satisfy:* $m_k = 1$.

Comments: i) The last condition in the above definition is coherent since the set of equations has been pruned: if there is an equation of the form $X_k = a$, then X_k cannot be the lhs of a 'product'-equation.

ii) The condition imposed on the left-half n-bit vector of an accepting state has the following justification: suppose the state C reached by the run at a node w is such that $C.l_k = 1$; this means that the subterm t_w at node w of the dag is accepted by the run, in state C, as an element in the set X_k; if we have an $ACID$-equation $X_k = X_i * X_j$ in the problem, then w is of the form $w = u * v$ and $t_w = t_u * t_v$, where $t_u \in X_i$ and $t_v \in X_j$; if $C.l$ is such that we *also* have, for instance, $C.l_i = 1$, then the term t_w is also accepted by the run at the same node as element of the set X_i; so (for the $ACID$-equation to be valid) the set X_k must also contain the 'overterm' $t_w * t_v$; this cannot be accepted by the run at the node w, and so we must wait for this overterm to be covered by the run, before 'accepting the run'.

Note that *for a given label, the transitions on our LDA are uniquely defined.* Note also that although these transitions impose conditions on the bit vectors of the states, that may **not** give sufficient information on the the labels of the nodes which get mapped to these states under a run. This is taken care of in the following definition.

Definition 5 *i) For any lt-dag* t, *the subterm of the dag at a node* w *will be denoted by* t_w; *the underlying term at the root node, will also be denoted* t. *The label of* t *at node* w *will be denoted by* l^w, *and for any* i, i*-th entry of this label by* l^w_i.

ii) An lt-dag t *is said to be* coherent *(w.r.t. '*$*$*') if and only if: for any node* w *on* t *of the form* $u * v$, *the labels at the nodes satisfy the following condition: if* $k = 1..n$, *and there is an equation of the form* $X_k = X_i * X_j$ *in the ACID-problem, then:* $l^w_k \geq l^u_i \wedge l^v_j$.

iii) A run r *on the lt-dag is said to be* coherent *w.r.t. the ACID-problem iff for any node* w *on* t *with* $r(w) = C$, *we have:* $C.l = l^w$.

We shall be *assuming that all our lt-dags and the runs considered henceforth are coherent* (w.r.t. the given $ACID$-problem). We want to define the notion of acceptance of an lt-dag by the LDA, so that from an accepted lt-dag we can deduce a solution to the $ACID$-problem by looking at the labels at its various nodes. For this to be true, it is obviously *necessary* that the following condition is satisfied : if the subterm t_u of t at node u (resp. t_v at node v) gets accepted by a run as element in a set X_i (resp. in a set X_j) and if there is a product equation $X_k = X_i * X_j$ in the problem, then the run r *must* reach a state where the product term $t_u * t_v$ gets accepted in the set X_k. Such a condition can be formulated in terms of the run alone, as in the following definition.

Definition 6 *i) A run r on an lt-dag t is said to be* **closed** *w.r.t. a given ACID-problem in (standard, pruned) form, iff the following property holds:*

- *If $A = r(u), B = r(v)$ are* any *two states in the image of r on the LDA, such that $A.l_i \neq 0$ and $B.l_j \neq 0$ for some i, j, and if there is an equation $X_k = X_i * X_j$ in the given ACID-problem, then a transition of the form $A * B \longrightarrow C$* must *be present in r* and *lead from nodes $u, v \in t$ to a node of t.*

ii) An lt-dag t is accepted *by the LDA associated to the ACID-problem if and only if: there is a run r which is coherent and* closed*, and maps the root node of t to an accepting state. We then say that r is an* accepting run *on t, or that it is* successful *on t.*

The following examples illustrate the notions developed above.

Example 2a. The $ACID$-problem: $Z =^{?} X * Y$ admits as a solution $Y = \{b\}$, $X = \{b * b,\ b\}, Z = \{b * b,\ (b * b) * b\}$. Arrange the variables into an ordered list $\{X, Y, Z\}$, and consider the (coherent) lt-dag t with term $(b * b) * b$, labeled with (110) at the leaf b, (101) at the subterm $(b * b)$, and (001) at the root. The following run is coherent only when the bit λ is 1:

$$b \stackrel{110}{\longrightarrow} B = (110; 110), \quad B * B \stackrel{\lambda 01}{\longrightarrow} C = (\lambda 01; 111)$$

(but the progression from $B * B$ to C is correct on the LDA, for any λ). The term at the node reached is $b * b$; if $\lambda = 1$ the run is coherent, and the values deduced from the labels for the sets are: $Y = \{b\},\ X = \{b * b,\ b\}, Z = \{b * b\}$. This does not give a solution to the $ACID$-problem, although all the $C.h_i$ are 1. But we can complete this run into a closed, accepting one: $C * B \stackrel{001}{\longrightarrow} D = (001; 111)$, mapping the root of t to the accepting state D; the solution can be deduced from the labels of the lt-dag t. *Note: The 'product equations are not necessarily satisfied* all along *a successful run'.* □

This shows that it is necessary for the run to be coherent, for the LDA-approach to be correct. The next example shows that, the coherence of the lt-dag and of the run alone do not suffice to guarantee that the LDA-approach is correct.

Example 2b. The following $ACID$-problem is unsolvable in terms of *finite non-empty* sets:

$$X_1 = X_2 + X_3, \qquad X_1 = X_4 + X_5, \qquad X_1 = X_6 + X_7$$

$$X_4 = X_2 * X_8, \qquad X_6 = X_9 * X_3$$

Indeed set d_i to be the maximum depth of the terms in the set X_i. We then have: $d_1 = max(d_2, d_3) = max(d_4, d_5) = max(d_6, d_7)$; but also $d_4 = max(d_2, d_8) + 1 > d_2$, and $d_6 = max(d_3, d_9) + 1 > d_3$, which is impossible. But here is a coherent run on a coherent lt-dag - with a as the unique leaf, and $((a*a)*(a*a))$ at the root - leading to an accepting state (variables in the natural order of indices):

$$a \stackrel{111010111}{\longrightarrow} A = (111010111; 111010111)$$

$$A * A \stackrel{110101000}{\longrightarrow} B = (110101000; 111111111)$$

And finally: $B * B \xrightarrow{000000000} C = (000000000; 111111111)$. Note that the run is *not* closed: for example, no transition of the form $B * A \longrightarrow E$ is present in the run. □

We show now that the LDA-approach is sound.

Proposition 3 ('The LDA-approach is sound') *Suppose given an ACID-problem in standard (pruned) form, and let t be an lt-dag accepted by the LDA associated to the problem, in the sense of the previous definition. Then the values assigned to the various set variables by the different nodes on t at the end of a successful run give a solution to the ACID-problem.*

Proof. There is an accepting, closed, run r on t by assumption. We shall reason inductively on the run, w.r.t. the height of the *lt*-dag t. For every node reached w and every variable X_i, define E_i^w as the set of all subterms of the underlying term of the dag accepted as elements of the set X_i by the run r *at or below* the node w. (The run being coherent, this is also the set of subterms of t at nodes at or below w with labels l such that $l_i = 1$.) At the root node of t these sets will be just denoted E_i. We show then that the following invariants are satisfied all along the run r:

Let node w be such that $r(w) = (.., l_i, ...; .., h_i, ...)$
  then for any $k = 1..n$
  if $h_k = 1$ then E_k^w is non-empty
    AND
      (i) for EVERY equation $X_k = X_i + X_j$,
      we have $E_k^w = E_i^w + E_j^w$,
    AND
      (ii) for EVERY equation $X_k = X_i * X_j$,
      we have $E_k^w \subset E_i^w * E_j^w$

The validity of the invariant at initialization is straightforward. Along the run, that $h_k = 1$ implies E_k^w is non-empty, follows from (the coherence of t and r) and the fact that the E_k^w are monotonic increasing with the height of the node w. Suppose r is using a transition $A * B \xrightarrow{m} C$ to reach node w from nodes u, v; and assume that the properties are satisfied at the nodes strictly below w. Now, at the node w: property (i) remains true by the first of the conditions on the transition. As for property (ii), note that - due to the inductive hypothesis, and the fact that the sets E_k^w are monotonic with the height of the node - it suffices to consider the subterm of the dag accepted by the run at w i.e. to say the subterm t_w. Now t_w is accepted in X_k by the run at w (the progression rule) if and only if $t_w = t_u * t_v$ with $t_u \in E_i^u,\ t_v \in E_j^v$. Since $t_u \in E_i^u \subset E_i^w,\ \ t_v \in E_j^v \subset E_j^w$, we deduce that property (ii) is satisfied at w.

If t is accepted by the run, then we have to prove that at the root node of t, all the inclusions of property (ii) become equalities (even if they may be strict inequalities at some nodes along the run; cf. Example 2b). Now choose any k for which we have an equation of the form $X_k = X_i * X_j$, let t_u, t_v be two subterms of t accepted at their

nodes u, v as elements in the sets X_i and X_j respectively. Let $r(u) = A$ and $r(v) = B$; then $A.l_i \wedge B.l_j \neq 0$; now by assumption r is a closed run, so there must be a transition $A * B \xrightarrow{m} C$ used by the run r to reach a node w in t from u, v; the term $t_u * t_v$ gets accepted in X_k at this node w of t; this proves the reverse inclusion $E_k \supset E_i * E_j$.

Finally, it is immediate that a solution to the problem can be deduced by reading off the labels at the various nodes of t, at the end of the successful run. □

Example 3. The $ACID$-unification problem: $X + Z = X * Y + U$ may be transformed into the following standard (pruned) form:

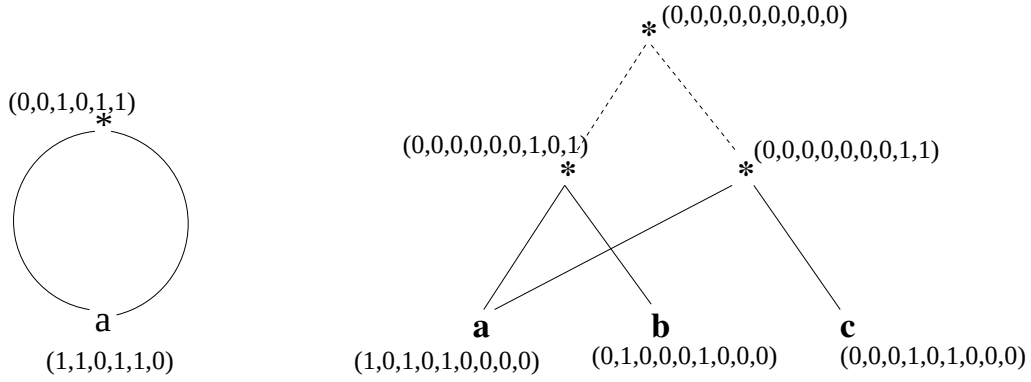
$$V = X + Z\,; \quad V = W + U\,; \quad W = X * Y$$

Let us show that the lt-dag to the left in the following Figure 2 is accepted by the associated LDA. We first arrange the set-variables into an ordered list, say as $\{X, Y, Z, U, V, W\}$.

The states on the LDA are therefore 12-bit vectors. From the node 'a' on the lt-dag with the given label $m = (1, 1, 0, 1, 1, 0)$ we first have a transition (with this label m) to a state q_1 on the LDA:

$$a \xrightarrow{m} q_1 = (1, 1, 0, 1, 1, 0; 1, 1, 0, 1, 1, 0)$$

representing the fact that at the current node the values $X = Y = a = V = U$ have been accepted. Now we have a transition on the LDA from $q_1 * q_1$ to the state $q_2 = (0, 0, 1, 0, 1, 1; 1, 1, 1, 1, 1, 1)$ with label $(0, 0, 1, 0, 1, 1)$; at this node, the assignments $W = a * a = Z = V$ have been accepted. The run is closed, and the state reached at the root node is an accepting state. So the lt-dag is accepted; and a solution to the unification problem is obtained by reading off the assignments to the various set-variables, at the nodes of this lt-dag.



Solution - Example 3 Solution - Example 4

Fig. 2. lt-dags as solutions to $ACID$-unification problems

Proposition 4 *Suppose given an ACID-unification problem, and let $\mathcal{A}$ be the associated LDA. If the ACID-problem is solvable, then for any given solution we can construct an*

lt-dag t accepted by $\mathcal{A}$, which corresponds to this solution in the sense of the previous proposition.

Proof. Since the given $ACID$-problem is solvable, its dependency graph (cf. Section 4.1) is loop-free. So we can arrange the set of all the equations of the problem as a stack such that: for any variable X in the rhs of any given equation, any equation with X on the left is above the given equation in the stack. Let us call an *end-variable* of the problem any X from which there is no outgoing edge on the dependency graph, or which is the lhs of a (uniquely determined) equation of the form $X = a$; such variables appear over the rhs of equations on top of the stack. Also, if n is the number of variables in the problem we arrange these variables into an ordered list of length n.

Now, the solution given for our $ACID$-problem assigns finite non-empty sets of terms over '$*$' and the constants, to each of these end-variables X. To each such assignment we can therefore associate (uniquely) a set of lt-dags, whose common nodes are all shared, the various nodes of each dag being labeled with appropriate n-bits w.r.t. the ordered list of the set variables of the problem: set the i-th bit to 1 iff the subterm of the dag at the node is in the solution for the i-th variable of the list. We then apply the following algorithm:

0). Start with the set $\mathcal{D}$ of all the lt-dags associated to all the end-variables, and the stack $\mathcal{S}$ of all the equations of the problem;
1). If $\mathcal{S}$ is empty then STOP, else pop an equation from $\mathcal{S}$;
if it is a 'sum' discard, and goto step 1;
2). if the equation popped is a 'product' $Z = X * Y$, then
`if` the labels of the lt-dags currently in $\mathcal{D}$ give all the elements
in (the solution for) Z, `then` discard; goto step 1;
`else`, for *every* lt-dag $s \in X$ and $t \in Y$: form the dag $s * t$ (sharing its
common nodes with those of $\mathcal{D}$), label its root with the n-bit derived
from the solution given, add it to $\mathcal{D}$ if not already there, otherwise discard;
3). Goto step 1.

Such a procedure necessarily terminates; at that point we have a set $\mathcal{D}$ of lt-dags (sharing their common nodes), the labels of which tally with the given solution w.r.t. all the set variables of our problem. We can then build a single (not necessarily unique) lt-dag containing all these as sub-dags; and label the nodes of this 'global' lt-dag which are strictly above the nodes of the dags coming from $\mathcal{D}$, with the *null* n-vector (see example below).

To construct a successful run on such a 'global' lt-dag, we proceed again incrementally, as above. Observe first that for every root node of any lt-dag associated to an end-variable, with label m, it is an easy matter to construct an 'initial run' from this node to a state A in $\mathcal{A}$ which is the $2n$-bit vector such that $A.l = A.h = m$. These initial runs can be completed, at every step 2 of the above algorithm which adds a new lt-dag to $\mathcal{D}$, into a run from the new node added to a state B in $\mathcal{A}$ such that: $B.l$ is the label of the node added, $B.h \geq B.l$ **and** for any i, $B.h_i = 1$ if some node below the node added is in the solution for the i-th variable (cf. the definition of the transition rules on the LDA). If we need to

build a global lt-dag from the final set of lt-dags in $\mathcal{D}$, we proceed in the same manner for defining an accepting run from this global dag an accepting state in the LDA. We are constructing the run *from a solution*, and the set of terms in a solution is 'stable w.r.t. the product-equations in the problem'; so it follows that the above construction gives a closed run. □

Example 4. For the following $ACID$-unification problem (in standard, pruned form):

$$W = X * Y\,;\quad W = U + V\,;\quad U = U1 * U2\,;\quad V = V1 * V2\,;$$

$$U1 = a\,;\quad U2 = b\,;\quad V1 = a\,;\quad V2 = c.$$

the solution is given by the substitution $X = \{a\}, Y = \{b, c\}$ on the 'end-variables' X, Y. The lt-dag to the right in Figure 2 gives a successful run on the LDA associated to the problem (the set variables being arranged into the ordered list $\{U1, U2, V1, V2, X, Y, U, V, W\}$).

Theorem 7. *$ACID$-unification is NEXPTIME-decidable, and DEXPTIME-hard.*

Proof. We assume that the $ACID$-unification problem is in standard form, and has been pruned. Then its set of equations admits a solution iff: seen as a set-constraints problem the associated LDA has a non-empty language. This LDA being of size exponential w.r.t. the number of variables in the problem, it suffices to show that finding an lt-dag *accepted* by this LDA is NP-decidable. Now we already know (cf. the end section 4.2) that finding an lt-dag t *pseudo-accepted* by the LDA, i.e. to say with a run r leading to an accepting state, is NP-decidable.

It suffices to show that checking that this run r is coherent and closed, can be done in polynomial time w.r.t. the number of states on the LDA. For ensuring this we proceed as follows. We first observe that any lt-dag t (w.r.t. the given $ACID$-problem) can be naturally seen as (or more precisely: extended to) a GTS (*Generalized Tree Set*), defined in [12] as a map from the set of ground terms to the set $\{0, 1\}^n$. The LDA $\mathcal{A}$ associated to the $ACID$-problem, as well as a run r of this LDA on t, can then be seen naturally, respectively as a GTSA (*Generalized Tree Set Automaton*, as defined in *loc. cit.*), and a run of this GTSA on the GTS. Then, the lt-dag t is pseudo-accepted by the LDA under the run r, *if and only if* the run maps the GTS into the set of accepting states on the GTSA. Now, by the 'Main Lemma' in the appendix of [12], there exists such a run on the GTSA if and only if there also exists a successful GTSA-run on a tree set t' bounded in height by a polynomial of degree 4 w.r.t. the number of states of the GTSA, such that t' is 'stably completable' w.r.t. the finite set of terms represented by t. We may thus assume that the (tree underlying the lt-dag) t is of height bounded polynomially w.r.t. the number of states on the LDA. Checking that the LDA-run r is coherent and closed on the lt-dag t, can then be done in polynomial time w.r.t. the number of states on the LDA.

This proves the assertion of NEXPTIME-decidability. As for the DEXPTIME-hardness it is an adaptation of a proof that $ACUID$-unification is DEXPTIME-hard, that we give in the coming section. □

5 The *ACUID*-Unification Problem

We are interested here in the unification problem w.r.t. the following theory denoted $ACUID$, obtained from $ACID$ by adding a unit element '0' for '+' which is null for '$*$'; i.e. by adding the following two additional equations: $x + 0 \approx x$, $x * 0 \approx 0$.

The equations to solve for this unification problem are of the same form as for $ACID$-unification; the difference here is that '0' represents the empty set, and in any solution, we allow the empty set assignment for the variables. (Note: standard forms can be derived here as in the $ACID$-case, but *Occur-Check* has to be defined somewhat differently. But this is of no importance to what follows.)

Theorem 8. *If the equations of associativity-commutativity, or just associativity, of '$*$' are added on to ACUID, then ACUID-unification is undecidable.*

Proof. If the distributing symbol '$*$' of $ACUID$ is assumed AC in addition, undecidability follows from the undecidability of $ACUIH^C$-unification, . cf. proof of Theorem 5. If the additional assumption on the symbol '$*$' is just associativity, we can still derive undecidability by reduction from MPCP (with no restriction this time), as in the proof of Theorem 6. But the lines of reasoning have to be modified slightly here, as indicated below, since a priori the variables X_i can all have value 0 in a solution to equation (1).

Starting with $(w_0, w'_0), \ldots (w_n, w'_n)$ the given instance of MPCP over the alphabet Σ, we consider here the $ACUID$-unification problem defined by the following *three* equations:

$$(1) \qquad w_1 * X_1 * \overline{1} + \ldots + w_n * X_n * \overline{n} + w_0 * \# \;=\; X_1 + X_2 + ... + X_n + V$$
$$(2) \qquad w'_1 * Z_1 * \overline{1} + \ldots + w'_n * Z_n * \overline{n} + w'_0 * \# \;=\; Z_1 + Z_2 + ... + Z_n + V$$
$$(3) \qquad a_1 * Y_1 * \overline{1} + \ldots + a_p * Y_p * \overline{p} + \# \;=\; Y_1 + Y_2 + ... + Y_p + V$$

where the unknowns are the X_i, Z_i, Y_i and V, the $\#, \overline{1}, \overline{2}, \ldots, \overline{n}$ are some fixed new symbols, and the $\{a_i\}_{i=1..p}$ are constant symbols (representing the alphabet Σ). Assume this $ACUID$-unification problem to be solvable.

We claim then that not all the X_i can be 0 in the solution of (1). Otherwise V will have to be $w_0 * \#$; but then, not all the Y_j can be 0 in the solution for (3). Now, arguing as previously, we deduce that a word of maximal length in the rhs of (the solution for) (3) cannot be in any Y_j; consequently, V must have words ending with symbols from $\{\overline{1}, \overline{2}, \ldots, \overline{p}\}$, which gives a contradiction.

The rest of the proof can now be carried through, along the same lines of reasoning as for Theorem 6. □

Theorem 9. *ACUID-unification is NEXPTIME-decidable and DEXPTIME-hard.*

Proof. The NEXPTIME-decidability follows from that of $ACID$-unification: choose nondeterministically the variables going to 0, and solve for the others via $ACID$.

For the DEXPTIME lower bound, we proceed as follows: Consider any $ACUIh$-unification problem – where $ACUIh$ is the theory with an AC-symbol '+', assumed idempotent with a unit $U = 0$, and h is any given finite family of non-commuting homomorphisms w.r.t. '+'. First, the homomorphisms in the family h can be seen as defined by finitely many free constants $\{a, b, c, \ldots\}$ in an $ACUID$-signature; next, the $ACIDh$-problem itself can be seen as solving the following $ACUID$-unification problem:

$$l_1(X_1) + l_2(X_2) + \ldots + l_n(X_n) + T \stackrel{?}{=} r_1(X_1) + r_2(X_2) + \ldots + r_n(X_n) + S$$

where the unknowns are the X_i, the $\{l_i, r_i\}_{i=1..n}$ are *right-parenthesized terms* over '$*$' and the constants, and S, T are two finite sets of such terms. Indeed, it is easily checked that in any $ACUID$-solution of this problem, the X_i have *got to be* right-parenthesized terms over '$*$' and the constants; i.e. to say, terms representing (right-)compositions of the homomorphisms defined by the constants. This shows that $ACUIh$-unification can be seen as an instance of $ACUID$-unification; and proves the DEXPTIME-hardness of the latter, since the former has been proved DEXPTIME-complete in [4].

To prove Theorem 7, we show now how the proof of the DEXPTIME-hardness of $ACUIh$-unification, as given in [4], can be adapted so as to get rid of the (possible) role played by the unit element U of the theory.

Recall that this proof relies on Seidl's result [16] that if $A_i, i = 1..k$ are given finitely many DTTA's (deterministic topdown tree automata), then deciding the emptiness of their intersection is DEXPTIME-hard. In [4], an $ACUIh$-equation E_i is constructed for each of the A_i; these equations share exactly one common variable X. Solutions to any equation E_i are assignments of finite unions of 'run tree sets' constructed from finite trees accepted by the corresponding automaton A_i; the unique common variable X serves to glue these different assignments together into a single one, whenever there is a common solution for *all* the equations $E_i, i = 1..k$. A single global equation combining all these equations is defined, such that it is solvable iff there is a *common tree* t accepted by *all* the $A_i, i = 1..k$, in which case the run tree set derived from t is assigned to this unique common variable X; if the acceptance of this common tree t by any of the A_i does not involve all the transitions in the DTTA A_i, then the value assigned to this variable X is $U = 0$. We propose to restructure this reasoning in such a manner that the solvability of the global equation amounts to the existence of **one run tree set** common to the assignments for the different variables (which is weaker than asking for a common accepted tree t).

First we show that Seidl's result holds even if we restrict to signatures with *one* binary symbol g, *one* unary symbol s, and *one* constant '#'. The key idea is this: suppose have a set of binary function symbols $\{g_1, ..., g_n\}$. Then we can replace each g_i with $\lambda(x, y).s(s(...s(g(x, y))))$, with i occurrences of the symbol s. (In other words, the term is $s^i(g(x, y))$.) A DTTA on the original alphabet can thus be converted into a DTTA on the new alphabet in polynomial time.

Next, we form the equations E_i for each DTTA A_i like in [4] and following the notation therein, except that the variables are all *disjoint*:

$$\{q_f\} \cdot X_i \cup \bigcup_{(q,g) \in Q \times \Sigma} \{q\} \cdot X^i{}_{(q,g)} = \{q_0\} \cup \bigcup_{\delta(q,g) = (q_1, \ldots, q_k)} \{q_1 1 g, \ldots, q_k k g\} \cdot X^i{}_{(q,g)}$$

(Recall that the $\{q_f\}\cdot X_i$ in each of these equations E_i will be 'storing' finite unions of run tree sets.) We then form an additional DTTA A with q_f as its **only** state, and with the transitions: $\delta(q_f, g) := (q_f, q_f)$, $\delta(q_f, s) := q_f$. We form an equation for this DTTA too:

$$\{q_f\}\cdot Z \cup q_f \cdot X_{(q_f,g)} \cup q_f \cdot X_{(q_f,s)} = \{q_f\} \cup \{q_f 1g, q_f 2g\} \cdot X_{(q_f,g)} \cup \{q_f 1s\} \cdot X_{(q_f,s)}$$

Then we add the following equations which express the conditions that $\{q_f\} \cdot Z$ is a subset of all the other $\{q_f\}\cdot X_i$, for the variables X_i coming from the equations $E_i, i = 1..k$:

$$\{q_f\} \cdot X_i = \{q_f\} \cdot Z \cup \{q_f\} \cdot X_i$$

The rest of the reasoning in [4] can now be carried through: We can solve all these equations simultaneously with non-empty run tree sets for the variables, iff there is a common run tree set that can be assigned to the variable Z, i.e. iff the DTTA $A_i, i = 1..k$ and A have a non-empty intersection. □

6 Conclusion

Our concern in this paper has been to address the unification problem w.r.t. theories "above AC" which might be useful in practical situations. $ACID$ is a natural candidate and of special interest among them, being well-suited for set constraints which are so often employed in program analysis. Now, as mentioned in the introduction, the notion of a family of homomorphisms lies hidden under the distributive operator in $ACID$; the theory $ACIH^C$ can thus be viewed as situated "somewhere between" AC and $ACID$. It turns out that unification modulo $ACIH^C$ is undecidable since it can be coded in terms of the counter-machine configurations reachability problem. From the proof of this undecidability we deduce that unification modulo $ACID$ is undecidable if the distributing symbol '$*$' is also AC. If this distributing symbol is only associative, again unification modulo $ACID$ is undecidable, since it can be reduced from the MPCP problem. However, if no further theory on the distributing symbol '$*$' is present in $ACID$, then unification modulo $ACID$ is decidable; a NEXPTIME upper bound and a DEXPTIME lower bound can be given for this decidability result, if we formulate our problem in terms of set constraints, and resort to a labeled dag automata approach for their solvability in terms of finite non-empty sets. It is shown that the same results hold also for the $ACUID$-unification problem, where $ACUID$ is the theory obtained from $ACID$ by adding a unit element $U = 0$ for the AC symbol, which is 'null' for the distributing symbol.

It might be of interest to consider the problem of the 'maximum' number of commuting homomorphisms that one can allow so that the $AC(U)IH^C$-unification problem is decidable; this clearly depends on the smallest number of instructions needed for the reachability problem on the configurations of 2-counter machines to remain undecidable.

References

1. A. Aiken, E. Wimmers. *Solving Systems of Set Constraints.* In Proc. of the 7th IEEE Symposium on Logic in Computer Science, 1992 (LICS'92), pp. 329 – 340.

2. A. Aiken, D. Kozen,, M. Vardi, E. Wimmers. *The Complexity of Set Constraints.* In Proc. Conf. CSL'93, EACSL, September 1993, pp. 1–18.
3. F. Baader. *Unification in Commutative Theories.* Journal of Symbolic Computation 8:479–497, 1989.
4. F. Baader, P. Narendran. *Unification of Concept Terms in Description Logics.* Journal of Symbolic Computation 31 (3): 277–305, 2001.
5. F. Baader, T. Nipkow. *Term Rewriting and all that.* Cambridge University Press, 1998.
6. F. Baader, W. Snyder. *Unification Theory.* In: J.A. Robinson and A. Voronkov, editors, Handbook of Automated Reasoning. Elsevier Science Publishers, 2001.
7. L. Bachmair, H. Ganzinger, U. Waldemann. *Set Constraints are the Monadic Class.* In Proc. of the 8th IEEE Symposium on Logic in Computer Science, 1993 (LICS'93), pp. 75 – 83.
8. W. Charatonik. *Automata on DAG representations of finite trees.* Technical Report MPI-I-99-2-001, Max-Planck-Institut für Informatik, Saarbrücken, Germany.
9. W. Charatonik, A. Podelski. *Set Constraints with Intersection.* In Proc. of the 12th IEEE Symposium on Logic in Computer Science, Warsaw 1997 (LICS'97), pp 362 – 372. (To appear in: *Information and Computation*).
10. H. Comon, M. Dauchet, R. Gilleron, F. Jacquemard, D. Lugiez, S. Tison, M. Tommasi. *Tree Automata Techniques and Applications.* http://www.grappa.univ-lille3.fr/tata/
11. R. Gilleron, S. Tison, M. Tommasi. *Solving Systems of Set Constraints using Tree Automata.* In Proc. STACS'93, Springer-Verlag, LNCS n^o 665, pp. 505–514.
12. R. Gilleron, S. Tison, M. Tommasi. *Set Constraints and Tree Automata* Information and Computation 149, pp. 1–41, 1999. (cf. also Technical Report IT 292, Laboratoire-LIFL, Lille, 1996.)
13. M. Hack. *Petri Net Languages.* Computer Struct. Gr. (CSG), Memo 124. Project HAC, MIT, 1975.
14. M. Minsky. *Computation: Finite and Infinite Machines. Prentice-Hall International*, London, (1972)
15. W. Nutt. *Unification in Monoidal Theories.* In Stickel (Ed.), Proc. CADE-10, Springer-Verlag, LNAI n^o449, pp. 618–632.
16. H. Seidl. *Haskell Overloading is DEXPTIME-complete.* In Information Processing Letters, 52(2):57–60.
17. J.-M. Talbot, Ph. Devienne, S. Tison. *Generalized Definite Set Constraints* In CONSTRAINTS: an International Journal 5(1-2):161–202, 2000.

7 APPENDIX: Closure Properties of DA's and LDA's

7.1 Universality of Dag Automata is Undecidable

The question of stability under complementation of the class of DA's is raised in [8], but left open. We give a negative answer here to this question, by first establishing the following result.

Theorem 10. *The problem of universality is undecidable on the class of DA's. In other words the following problem is* undecidable*:*

input*: an arbitrarily given dag automaton* $\mathcal{A}$

question*: Does* $\mathcal{A}$ *accept all inputs ?*

Proof. We reduce the problem to the halting problem for a deterministic 2-counter machine (q_0, F, Q, Δ) where Q is a finite set of states, $F \subseteq Q$ is the set of final states, q_0 is the initial state and Δ is a *transition function*. Each transition of the machine must correspond to a correct instruction as described in Section 2; the only difference here is that a computation

of the machine is accepted if it leads from q_0, with initial counter values both 0, to an accepting state in F.

We shall design a non-deterministic dag automaton $\mathcal{A}$ accepting precisely all the non-accepting or incorrect computations of the 2-counter machine. The non-universality of $\mathcal{A}$ is then equivalent to the existence of an accepting correct computation of the 2-counter machine. The design of this automaton $\mathcal{A}$ is realized by encoding the non-accepting or incorrect computations of the machine in terms of an appropriate set of ground rewrite rules, which will define the transition rules of our dag automaton $\mathcal{A}$.

For this encoding, we first consider the alphabet $Q \cup \{s, 0\}$ where every symbol of the set of states of the machine, $Q = \{q_0, \ldots, q_n\}$, will be seen as a ternary symbol, s is a new unary symbol and 0 is a constant. The integer value n for a counter will be represented as $s^n(0)$. A machine configuration is in general a triple $< q, c_1, c_2 >$ where q is the current state and the c_i are the current values of the two counters. We shall also be encoding these configurations as t-dags. The initial configuration of the machine will be coded as the dag representation of the term $t_0 = q_0(0, 0, \perp)$, where $\perp$ is the empty dag; a correct computation of length n corresponding to a sequence of machine instructions will be coded as the dag representation of

$$q_{i_n}(s^{l_n}(0), s^{r_n}(0), q_{i_{n-1}}(s^{l_{n-1}}(0), s^{r_{n-1}}(0), t_{n-1}))$$

where $q_{i_{n-1}}(s^{l_{n-1}}(0), s^{r_{n-1}}(0), t_{n-1})$ encodes a correct computation of length $n-1$ and the transition from configuration $< q_{i_{n-1}}, s^{l_{n-1}}(0), s^{r_{n-1}}(0) >$ to $< q_{i_n}, s^{l_n}(0), s^{r_n}(0) >$ is possible with the given counter machine.

We specify a dag automaton $\mathcal{A}$ for recognizing the incorrect or non-accepting sequence of configurations.

1. the set of states of $\mathcal{A}$ is $\Theta = \{\sigma_q \mid q \in Q\} \cup \{\mathcal{Q}_s, \mathcal{Q}_0, \mathcal{Q}_1, \mathcal{Q}_2, \texttt{error}, \omega, q_\perp\}$
   $\cup \{\tau_{(q,i,j)}, t_{(q,i,j)} \mid i, j \in \{0, 1, 2, \texttt{zero}, \neq \texttt{zero}\}, q \in Q\}$
2. $\Sigma = Q \cup \{s, 0\}$ is the ranked alphabet.
3. the set of *accepting* states is $\{\texttt{error}\} \cup \{\sigma_q \mid q \notin F\}$
   (recall that F is the set of accepting states of the counter machine)
4. and δ is the transition relation which consists of the rewrite rules specified below.

A few words of annotation will certainly help in understanding the extra state symbols added to $\mathcal{A}$. Symbol $\mathcal{Q}_s$ is a state to check that counter values are built up with 0 and s; $\mathcal{Q}_i, i = 0, 1, 2$, respectively are states where at least one of the counters has value i; the symbol $q_\perp$ is the starting state, ω is for "*any*" state, "`error`" a state for an incorrect machine configuration, $\tau_{(q,i,j)}$ for states where the current counter-machine state is q and machine-counter i has value j, while $t_{(q,i,j)}$ for states where current counter-machine state is q and machine-counter i is *tested* to check for value j, instantiating j to one of the symbols $\texttt{zero}, \neq \texttt{zero}$.

The following set of rewrite rules of the dag automaton $\mathcal{A}$ check that a dag represents a term of the type

$$q_{i_n}(s^{l_n}(0), s^{r_n}(0), q_{i_{n-1}}(s^{l_{n-1}}(0), s^{r_{n-1}}(0), \ldots, q_0(0, 0, \perp)))$$

or corresponds to an incorrect machine configuration.

I) Rules for *climbing up* a dag to some position where we want to apply a checking, and to extract a state from a computation at some step (nondeterministically):

$0 \longrightarrow \omega$
$q_\perp \longrightarrow \omega$
$q(\omega, \omega, \omega) \longrightarrow \omega$ for all $q \in Q$
$s(\omega) \longrightarrow \omega$
$q(\omega, \omega, \omega) \longrightarrow \sigma_q$ for all $q \in Q$

Note that if the extracted state at the root of the dag is not accepting (σ_q with $q \notin F$) then the dag will be accepted by our automaton $\mathcal{A}$ (since it will not encode a successful computation).

II) Rules for well-formed dags:

- initial state is q_0 and counters are initially 0:
  $\perp \longrightarrow q_\perp$
  $q(\omega, \omega, q_\perp) \longrightarrow \texttt{error}$ if $q \neq q_0$
  $q_0(\mathcal{Q}_1, \omega, q_\perp) \longrightarrow \texttt{error}$
  $q_0(\omega, \mathcal{Q}_1, q_\perp) \longrightarrow \texttt{error}$
- state $\mathcal{Q}_s$ checks that counters are built with $0, s$:
  $0 \longrightarrow \mathcal{Q}_s$ (0 is an acceptable counter content)
  $s(\mathcal{Q}_s) \longrightarrow \mathcal{Q}_s$
  $q(\omega, \omega, \mathcal{Q}_s) \longrightarrow \texttt{error}$ for all $q \neq s$ (no counter symbol in third position)
  $q(\sigma_{q'}, \omega, \omega) \longrightarrow \texttt{error}$ for $q' \in Q$ (no state symbol in first position)
  $q(\omega, \sigma_{q'}, \omega) \longrightarrow \texttt{error}$ for $q' \in Q$ (no state symbol in second position)
- rules for propagating errors up to the root:
  $q(\texttt{error}, \omega, \omega) \longrightarrow \texttt{error}$
  $s(\texttt{error}) \longrightarrow \texttt{error}$

III) Rules for transitions of type (i): $< q', x, y > \vdash < q, x+1, y >$, incrementing the first counter. These rules attempt to build a run such that the first counter after the instruction does not get incremented or such that the second counter is modified.

- rules to count s:
  $0 \longrightarrow \mathcal{Q}_0$
  $s(\mathcal{Q}_0) \longrightarrow \mathcal{Q}_0$
  $s(\mathcal{Q}_0) \longrightarrow \mathcal{Q}_1$ for counting one s
  $s(\mathcal{Q}_1) \longrightarrow \mathcal{Q}_2$ for counting two s
  $s(\mathcal{Q}_2) \longrightarrow \mathcal{Q}_2$ for counting more than two s
- rule to ensure that next state is the right one:
  $q''(\omega, \omega, \sigma_{q'}) \longrightarrow \texttt{error}$ if $q'' \neq q$

- rules to ensure that the counter is not incremented by more than 1:
  $q'(\mathcal{Q}_0, \omega, \omega) \longrightarrow \tau_{(q',1,0)}$
  $q(\mathcal{Q}_2, \omega, \tau_{(q',1,0)}) \longrightarrow \texttt{error}$
- rules to ensure that the counter value is not the same as previously:
  $q'(\mathcal{Q}_1, \omega, \omega) \longrightarrow \tau_{(q',1,\neq zero)}$
  $q(\mathcal{Q}_1, \omega, \tau_{(q',1,\neq zero)}) \longrightarrow \texttt{error}$
  $0 \longrightarrow \texttt{zero}$
  $q'(\texttt{zero}, \omega, \omega) \longrightarrow \tau_{(q',1,zero)}$
  $q(\texttt{zero}, \omega, \tau_{(q',1,zero)}) \longrightarrow \texttt{error}$
- rules to ensure that the counter value is not less than previous value:
  $q'(\mathcal{Q}_1, \omega, \omega) \longrightarrow \tau_{(q',1,1)}$
  $q(\mathcal{Q}_0, \omega, \tau_{(q',1,1)}) \longrightarrow \texttt{error}$
- rules to ensure that the second counter is not modified:
  $q'(\omega, \mathcal{Q}_0, \omega) \longrightarrow \tau_{(q',2,0)}$
  $q(\omega, \mathcal{Q}_1, \tau_{(q',2,0)}) \longrightarrow \texttt{error}$
  $q'(\omega, \mathcal{Q}_j, \omega) \longrightarrow \tau_{(q',2,1)}$ if $j \neq 0$
  $q(\omega, \mathcal{Q}_0, \tau_{(q',2,1)}) \longrightarrow \texttt{error}$

We omit the similar rules for transitions of type (i) incrementing the second counter, of the form: $< q', x, y > \vdash < q, x, y+1 >$.

IV) Rules for transitions of type (ii) with zero-test on first counter: $< q', 0, y > \vdash < q, 0, y >$ and $< q', s(x), y > \vdash < q'', x, y >$. (In these rules qq means any state of the counter machine.)

- rules to ensure that the first counter remains 0:
  $0 \longrightarrow \texttt{zero}$
  $q'(\texttt{zero}, \omega, \omega) \longrightarrow t_{(q',1,\texttt{zero})}$ record that state is q' with first counter zero
  $qq(\omega, \omega, t_{(q',1,\texttt{zero})}) \longrightarrow \texttt{error}$ for all $qq \neq q$ (forces branch to q)
  $q'(\mathcal{Q}_1, \omega, \omega) \longrightarrow t_{(q',1,\neq\texttt{zero})}$ record that state is q' with first counter $\neq$ zero
  $qq(\omega, \omega, t_{(q',1,\neq zero)}) \longrightarrow \texttt{error}$ for all $qq \neq q''$ (forces branch to q'')
- rules to ensure that second counter remains the same:
  $q'(\omega, \mathcal{Q}_0, \omega) \longrightarrow t_{(q',2,0)}$
  $q(\omega, \mathcal{Q}_1, t_{(q',2,0)}) \longrightarrow \texttt{error}$
  $q'(\omega, \mathcal{Q}_1, \omega) \longrightarrow t_{(q',2,1)}$
  $q(\omega, \mathcal{Q}_0, t_{(q',2,1)}) \longrightarrow \texttt{error}$
- rules to ensure that first counter is decremented by 1:
  $q'(\mathcal{Q}_1, \omega, \omega) \longrightarrow t_{(q',1,1)}$
  $q''(\mathcal{Q}_j, \omega, t_{(q',1,1)}) \longrightarrow \texttt{error}$, $j \in 1..2$
  $q'(\mathcal{Q}_2, \omega, \omega) \longrightarrow t_{(q',1,2)}$
  $q''(\mathcal{Q}_0, \omega, t_{(q',1,2)}) \longrightarrow \texttt{error}$

Similar rules are added for test instructions on the second counter. It is straightforward to check that the language accepted by the dag automaton $\mathcal{A}$ under these transition rules are precisely the t-dags corresponding to the configurations which are incorrect or unaccepted our 2-counter machine. □

Since emptiness is decidable on DA's we may derive the following result as a consequence of our previous theorem.

Corollary 1. *The class of t-dag automata is* not *stable under complementation.*

7.2 LDA's are Closed under Union and Intersection

Given two LDAs $D_1 = (\Sigma_1, Q_1, F_1, \Delta_1, E)$ and $D_2 = (\Sigma_2, Q_2, F_2, \Delta_2, E)$ over the same label-set, with $Q_1 \cap Q_2 = \emptyset$; their (labeled) product $D_1 \times D_2$ can then be defined in the usual way:

$$D_1 \times D_2 = (\Sigma_1 \cap \Sigma_2, Q_1 \times Q_2, F_1 \times F_2, \Delta, E)$$

where Δ is defined as follows: for all symbols $f^{(k)} \in \Sigma_1 \cap \Sigma_2$ and labels l, if

$$f(p_1, \ldots, p_k) \xrightarrow{l} p_{k+1}, \qquad f(q_1, \ldots, q_k) \xrightarrow{l} q_{k+1}$$

are respectively transitions in Δ_1 and in Δ_2, then

$$f((p_1, q_1), \ldots, (p_k, q_k)) \xrightarrow{l} (p_{k+1}, q_{k+1})$$

is a transition in Δ. For a labeled dag t, given runs r_1 by D_1 and r_2 by D_2, then a unique run by $D_1 \times D_2$, which we denote by $r_1 \times r_2$ can be defined as

$$(r_1 \times r_2)(t') = (r_1(t'), r_2(t'))$$

for all subterms t' of t. This (standard reasoning) shows that the class of LDA's is closed under intersection. Stability under union is just as standard.

7.3 LDA's are Not Closed under Complement

Consider the (infinite) set of terms M, defined recursively over the signature $\{a^{(0)}, g^{(2)}\}$, as follows:

i) $a \in M$
ii) if $t \in M$ then $g(t, t) \in M$
iii) nothing else is in M

Ans let $Fin(M)$ be the set of all *finite* subsets of M. We show that there is no LDA that accepts $Fin(M)$.

The proof is by contradiction. Suppose there is such an automaton with $|Q| = k$. Consider a subset S of $Fin(M)$ that has $k + 2$ elements. Then in any accepting run r of the LDA there must be 2 distinct terms s_1 and s_2 in S, neither of them the biggest term in S, such that $r(s_1) = r(s_2) = p$ for some $p \in Q$. Since neither s_1 nor s_2 is the largest in S,

there must be a node u in the dag corresponding to $g(s_2, s_2)$. Now we can construct (see Figure 3) an accepting run r' for the labeled term-dag which is the same as before except for the first edge out of u which goes to the node for s_1 instead of s_2. In other words, the term at node u in the new dag is $g(s_1, s_2)$. This labeled term-dag clearly should not be accepted.

On the other hand, we show that the complement of $Fin(M)$ can be accepted by an LDA. Note that this is the set of all finite sets of terms which contain at least one term not in M. In other words, there must be at least one term in each of these sets that has a subterm of the form $g(t_1, t_2)$ where $t_1 \neq t_2$. In fact, the following LDA accepts this set:

$$
\begin{aligned}
a &\xrightarrow{\times} q_0 \\
a &\xrightarrow{\times} q_1 \\
g(q_0,\, q_0) &\xrightarrow{\times} q_0 \\
g(q_0,\, q_0) &\xrightarrow{\times} q_1 \\
g(q_1,\, q_1) &\xrightarrow{\times} q_0 \\
g(q_1,\, q_1) &\xrightarrow{\times} q_1
\end{aligned}
\qquad\qquad
\begin{aligned}
g(q_0,\, q_1) &\xrightarrow{0} q_w \\
g(q_0,\, q_1) &\xrightarrow{1} q_a \\
g(q_a,\, _) &\xrightarrow{\times} q_a \\
g(_,\, q_a) &\xrightarrow{\times} q_a \\
g(q_w,\, _) &\xrightarrow{0} q_w \\
g(_,\, q_w) &\xrightarrow{0} q_w \\
g(q_w,\, _) &\xrightarrow{1} q_a \\
g(_,\, q_w) &\xrightarrow{1} q_a
\end{aligned}
$$

The accepting state is q_a. Symbols $\times$ and $_$ are used as 'don't-care' symbols. The key idea is that one can always construct a run, using the transitions on the left side, in which the state at a node can be made to be either q_0 or q_1. The first two rules on the right side become applicable when and only when a node has two distinct children. □

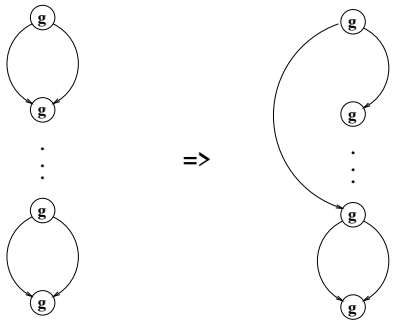


Fig. 3. Transformation used in section 7.3

7.4 Complexity of the Emptiness and Membership Problems for LDA

Lemma 2. *The emptiness problem for LDAs is NP-hard.*

Proof. The proof is by reduction from boolean satisfiability. Let B_f be a boolean formula over the binary operators $\{\wedge, \vee, |\}$ (where $|$ is Sheffer stroke or NAND) and variables $\{x_1, \ldots, x_n\}$. Let t_{bf} be a term-dag for B_f. Let m be the number of distinct nodes in t_{bf}.

We construct an automaton with $2m$ states. Corresponding to each node we have two states which stand for that subformula getting the corresponding truth-value. For ease of exposition we represent the states in the form $(s, 0)$ or $(s, 1)$ where s is a subterm of t_{bf}. The labels are 0 and 1. Now the rules are:

$$x_i \xrightarrow{0} q_{(x_i,0)}$$
$$x_i \xrightarrow{1} q_{(x_i,1)}$$
$$\vee(q_{(s_1,0)}, q_{(s_2,0)}) \xrightarrow{0} q_{(s_1 \vee s_2,0)}$$

In general, if $h \in \{\wedge, \vee, |\}$, then we have the transition

$$h(q_{(s_1,b_1)}, q_{(s_2,b_2)}) \xrightarrow{h(b_1,b_2)} q_{(s_1 \vee s_2, h(b_1,b_2))}$$

where b_1, b_2 are boolean values and $h(s_1, s_2)$ is a subterm of t_{bf}. The only accepting state is $q_{(t_{bf},1)}$. Note that this automaton is deterministic.

Now it is not hard to show that there is a labeling for t_{bf} such that the resulting labeled dag is accepted by this LDA if and only if the formula B_f is satisfiable. Note that the LDA will not accept any other dag no matter what the labels are. Thus the language of this LDA is empty if and only if B_f is satisfiable. □

Lemma 3. *The membership problem for LDAs is NP-hard.*

Proof. The same construction as above works, except that we replace the labels 0 and 1 with a *single* label c. (This makes the LDA nondeterministic.) □

Remark. A consequence of the results of this appendix is that the problem of *universality of an arbitrarily given LDA is undecidable.*