Synchronized Tree Languages Revisited and New Applications

Valérie Gouranton[1], Pierre Réty[1], and Helmut Seidl[2]

[1] LIFO, Université d'Orléans, France.
{gouranto,rety}@lifo.univ-orleans.fr
[2] Dept. of Computer Science, University of Trier, Germany.
seidl@psi.uni-trier.de

Abstract. We present a new formulation for tree-tuple synchronized languages, much simpler than the existing one. This new formulation allows us to prove stronger structural results. As a consequence, synchronized languages give rise to new applications :

– to rewriting : given tree languages L_1 (synchronized), L_2 (regular), $Rel(L_1) \subseteq L_2$ is decidable for several rewrite-like relations *Rel*.
– to concurrency : we prove decidability of the logic EF for a process calculus allowing some bounded form of communication. Consequently, the absence of deadlocks is decidable.

Keywords: tree-tuple language, rewriting, concurrency.

1 Introduction

In the field of tree[1] languages, let us consider the so-called *tree-tuple synchronized languages*, i.e. the languages generated by *Tree-Tuple Synchronized Grammars* (TTSG for short). TTSG's have been introduced to solve some equational unification [4,5] and disunification [6] problems. They have next been applied to logic program validation [8], and one-step rewriting theory [7]. Before going on, it is necessary to recall what a TTSG is, using an example. The following TTSG contains four packs of synchronized productions :

$$\begin{cases} X \Rightarrow f(X,X') \\ Y \Rightarrow f(Y,Y') \end{cases} \quad \begin{cases} X \Rightarrow b \\ Y \Rightarrow b \end{cases} \quad \begin{cases} X' \Rightarrow f(X,X') \\ Y' \Rightarrow f(Y,Y') \end{cases} \quad \begin{cases} X' \Rightarrow b \\ Y' \Rightarrow b \end{cases}$$

The first pack means that if X is derived into $f(X,X')$, then Y must be derived into $f(Y,Y')$ at the same time. Therefore if X appears in a given tree and Y does not, the tree cannot be derived by the first pack. If the axiom is the pair of non-terminals (X,Y), a possible derivation is :

$$(X,Y) \Rightarrow (f(X,X'),\, f(Y,Y')) \Rightarrow (f(X,f(X,X')),\, f(Y,f(Y,Y')))$$

Now there is an ambiguity : which X should be synchronized (derived at the same time) with which Y ? To remove this ambiguity, a control (an integer) is attached to each non-terminal, along the derivation, and is increased to a new

[1] Trees are first-order terms.

value whenever a pack of productions is applied. And the rule is: only non-terminals having the same control can be derived at the same time. Thus the above derivation is actually (For readability, control is written as non-terminal index):

$$(X_0, Y_0) \Rightarrow (f(X_1, X'_1),\ f(Y_1, Y'_1)) \Rightarrow (f(X_1, f(X_2, X'_2)),\ f(Y_1, f(Y_2, Y'_2)))$$

Now, an X and a Y can be derived at the same time only if they appear at identical positions in the two trees. Thus, this grammar generates the pairs of identical terms.

To get something interesting and more general for applications, control is indispensable. Unfortunately, this simple control is not enough when computing the natural join of two tree-tuple synchronized languages: we get a TTSG that needs a more complicated control (pairs of integers). Worse, when computing several joins incrementally, each step needs a control more complicated than the previous one (tuples of integers, see [5] for details). In this paper :

– We present a simple formalism which essentially is equivalent to TTSG's of level 1 (i.e., those with an integer as control) but elegantly circumvents the use of controls. The idea is that in the new formalism, a non-terminal will represent tuples of synchronizable[2] trees (instead of single trees) from which we will extract components, as needed. This new grammar formalism is given in the form of constraint systems for which we consider their least fix-point solutions. In other words, we adopt the bottom-up point of view instead of the top-down one[3].
– Instead of formally proving the equivalence of the two formalisms, we prefer to present the vastly simplified proofs of existing results using the new formalism, and also take advantage of its simplicity by proving new results. In particular, we show closure by union and cartesian product (existing results), as well as closure by projection (new) and, as our key technical result, also closure under join (new[4]). Moreover, we present a linear time emptiness for constraints (an exponential-time pumping technique, very complicated because of control, was used for TTSG's), and a rather efficient membership test (comparable to [12] but much simpler).
– Thanks to the new results, we get new applications :
  - to rewriting: given tree languages L_1 (synchronized, then non-necessarily regular), L_2 (regular), we prove that $Rel(L_1) \subseteq L_2$ is decidable for one-step, parallel, one-pass, one-pass root-started rewritings.
  - to concurrency : we extend the work of [10] on PA-processes by introducing bounded communications. We get a new concurrency formalism for which we show that the whole logic EF is decidable. In particular, we are able to verify that a given process is deadlock free.

[2] I.e. that had identical control values.
[3] Due to the essential equivalence, the reader may still also interpret constraint systems as grammars.
[4] This amounts to show that TTSG's of level $n > 1$ are actually useless.

2 Constraint Systems for Tuple Synchronized Languages

Example 2.1. This is the example given in the introduction, but now expressed with a constraint system. In the signature $\Sigma = \{f^{\backslash 2}, b^{\backslash 0}\}$ let $L_{id} = \{(t,t) \mid t \in T_\Sigma\}$ be the set of pairs of identical terms. L_{id} can be defined by the following grammar, given in the form of a constraint system:

$$\begin{array}{l} X_{id} \supseteq (b,b) \\ X_{id} \supseteq (f(1_1,2_1),\ f(1_2,2_2))\ (X_{id}, X_{id}) \end{array}$$

where $1_1, 2_1, 1_2, 2_2$ abbreviate pairs (for readability). For example 2_1 means $(2,1)$, which denotes the first component of the second argument (the second X_{id}). Note that since 1_1 and 1_2 come from the same X_{id}, they represent two identical terms, in other words they are linked (synchronized), whereas for example 1_1 and 2_1 are independent.

Example 2.2. Now if we consider the slightly different constraint system :

$$\begin{array}{l} X_{sym} \supseteq (b,b) \\ X_{sym} \supseteq (f(1_1,2_1),\ f(2_2,1_2))\ (X_{sym}, X_{sym}) \end{array}$$

we get the set $L_{sym} = \{(t, t_{sym}) \mid t_{sym}$ is the symmetric of $t\}$.

Example 2.3. In the signature $\Sigma = \{s^{\backslash 1}, b^{\backslash 0}\}$ let $L_{dble} = \{(s^n(b), s^{2n}(b))\}$. It can be defined by the constraint system :

$$\begin{array}{l} X_{dble} \supseteq (b,b) \\ X_{dble} \supseteq (s(1_1),\ s(s(1_2)))\ X_{dble} \end{array}$$

General Formalization

Assume we are given a (universal) index set N for tuple components. For $I \subseteq N$ and any set M, the set of I-tuples $a : I \to M$ is denoted by M^I. Often, we also write $a = (a_i)_{i \in I}$ provided $a(i) = a_i$ which for $I = \{1, \ldots, k\} \subseteq \mathbb{N}$, is also written as $a = (a_1, \ldots, a_k)$.

Different tree-tuple languages may refer to tuples of different length, or, to different index sets. Our constraint variables represent tree-tuple languages. Consequently, they have to be equipped with the intended index set. Such an assignment is called *classification.* Accordingly, a *classified* set of tuple variables (over N) is a pair $(\mathcal{X}, \rho)$ where $\rho : \mathcal{X} \to 2^N$ assigns to each variable X a subset of indices. This subset is called the *class* of X. For convenience and whenever ρ is understood, we omit ρ and denote the classified set $(\mathcal{X}, \rho)$ by $\mathcal{X}$. The maximal cardinality of the classes in $\mathcal{X}$ is also called the *width* of $\mathcal{X}$. In particular, in example 2.1, $N = \{1,2\}$, $\mathcal{X} = \{X_{id}\}$, and $\rho(X_{id}) = \{1,2\}$. Thus, the width of $\mathcal{X}$ is 2.

A constraint system for tree-tuple languages consists of a classified set $(\mathcal{X}, \rho)$ of constraint variables, together with a set $\mathcal{E}$ of inequations of the form

$$X \supseteq \Box(X_1, \ldots, X_k) \qquad (1)$$

where $X, X_1, \ldots, X_k \in \mathcal{X}$ and $\Box$ is an operator mapping the concatenation of tuples for the variables X_i to tuples for X. More precisely, let

$$J = \{(i, x) \mid 1 \le i \le k, x \in \rho(X_i)\} \tag{2}$$

denote the *disjoint union* of the index sets corresponding to the variables X_i (in example 2.1, $J = \{(1,1), (1,2), (2,1), (2,2)\}$ abbreviated into $\{1_1, 1_2, 2_1, 2_2\}$). Then $\Box$ denotes a mapping $T_\Sigma^J \to T_\Sigma^{\rho(X)}$. Each component of this mapping is specified through a tree expression t which may access the components of the argument tuple and apply constructors from signature Σ. Thus, t can be represented as an element of $T_\Sigma(J)$ where $T_\Sigma(J)$ denotes all trees over Σ which additionally may contain nullary symbols from the index set J.

Consider, e.g., the second constraint in example 2.1. There, the first component of the operator is given by $t = f(1_1, 2_1)$.

The mapping induced by such a tree t then is defined by

$$t(s_j)_{j \in J} = t\{j \mapsto s_j\}_{j \in J}$$

for every $(s_j)_{j \in J} \in T_\Sigma^J$. Accordingly, $\Box$ is given by a tuple

$$\Box \in T_\Sigma(J)^{\rho(X)}$$

Let us collect a set of useful special forms of constraint systems. The constraint (1) is called

- *non-copying* iff no index $j \in J$ occurs twice in $\Box$;
- *irredundant* iff $\Box$ refers to each argument at least once, i.e., $\Box$ contains at least one occurrence of some (j, x_j) for each j;
- *empty* iff $\Box$ contains no constructor application, i.e., $\Box_x \in J$ for all $x \in \rho(X)$;
- *Greibach* iff each component $\Box_x$ either is contained in J or is a single constructor application $a(j_1, \ldots, j_n)$, $a \in \Sigma, j_1, \ldots, j_n \in J$;
- *regular*[5] iff each component of $\Box$ is a single constructor application of the form $\Box_x = a((1,x), \ldots, (n,x))$ for each $x \in \rho(X)$.

The whole constraint system is called non-copying, (irredundant, non-empty, Greibach, regular) iff each constraint in $\mathcal{E}$ is so.

For example, the constraint systems that define L_{id} and L_{sym} are non-copying, irredundant, and Greibach. Moreover L_{id} is regular. On the other hand, L_{dble} is not Greibach. It is always possible to write any constraint system in Greibach form. Thus the Greibach form of L_{dble} is :

$$X_{dble} \supseteq (b, b) \qquad\qquad Y \supseteq (1_1,\, s(1_2))\ X_{dble}$$
$$X_{dble} \supseteq (s(1_1),\, s(1_2))\ Y$$

A variable assignment is a mapping σ assigning to each variable $X \in \mathcal{X}$ a subset of $T_\Sigma^{\rho(X)}$ (i.e. a set of tuples of ground terms). Variable assignment σ satisfies the constraint (1) iff

$$\sigma(X) \supseteq \{\Box(t_1, \ldots, t_k) \mid t_i \in \sigma(X_i)\} \tag{3}$$

[5] This corresponds to the notion of regular relation, i.e. the tree language over the product-alphabet obtained by overlapping the tuple components is regular.

Note here that the $\square$-operator is applied, in fact, to the cartesian product of the argument sets $\sigma(X_i)$. In particular, the tuples inside the argument sets are kept "synchronized" while tuples from different argument sets may be arbitrarily combined.

The variable assignment σ is a *solution* of the constraint system iff it satisfies all constraints in the system. Since, the operator application in (3) is monotonic, even continuous (w.r.t. set inclusion of tuple languages) we conclude that each constraint system has a unique least solution.

3 Properties

3.1 Deciding Emptiness

The fix-point characterization of languages defined by tuple language constraint systems, allows us immediately to derive a polynomial emptiness test. In order to do that, we first observe that we can always remove from a constraint $X \supseteq \square\ (X_1, \ldots, X_k)$ all arguments X_j which do not contribute to the components of X. Consequently, we have:

Proposition 3.1. *For every constraint system $\mathcal{C}$ we can construct in linear time a irredundant constraint system $\mathcal{C}'$ such that $\mathcal{C}$ and $\mathcal{C}'$ have the same least solution.*

For irredundant systems, however, emptiness can be decided along the same lines as for tree automata. For the constraint system $\mathcal{C} = (\mathcal{X}, \mathcal{E})$ we (conceptually) introduce a new ranked alphabet Δ which contains a new letter a_r for each constraint r in $\mathcal{E}$ where a_r is of rank k provided the right-hand of r refers to k variables. Then the *uninterpreted* regular constraint system $\mathcal{C}_0 = (\mathcal{X}_0, \mathcal{E}_0)$ of $\mathcal{C}$ is obtained as follows:

- $\mathcal{X}_0$ is obtained from $\mathcal{X}$ by forgetting the classification.
- $\mathcal{E}_0$ is obtained from $\mathcal{E}$ by replacing each constraint r of the form $X \supseteq \square\ (X_1, \ldots, X_k)$ with the constraint $X \supseteq a_r\ (X_1, \ldots, X_k)$.

Theorem 3.2. *Let $\mathcal{C}$ denote a constraint system and $\mathcal{C}_0$ the uninterpreted regular constraint system of $\mathcal{C}$. Furthermore, let σ and σ_0 denote the least solutions of $\mathcal{C}$ and $\mathcal{C}_0$, respectively. Then the following holds:*

1. *If $\mathcal{C}$ is irredundant, then for every $X \in \mathcal{X}$, $\sigma\ X \neq \emptyset$ iff $\sigma_0\ X \neq \emptyset$.*
2. *The set $\{X \in \mathcal{X} \mid \sigma\ X \neq \emptyset\}$ can be computed in time linear in the size of $\mathcal{C}$.*

Indeed, theorem 3.2 does not come as a surprise. Taking a closer look at our definition of tuple languages, reveals that these can be viewed as polynomial systems over a specific algebra of graphs (cf. [2]).

3.2 Basic Closure Properties

We are actually interested in non-copying constraint systems, which are enough to formalize tree-tuple synchronized languages, because there is no copying in TTSG's. Let $\mathcal{L}$ denote the class of languages defined by non-copying constraint systems. We now collect some basic closure properties of languages in $\mathcal{L}$. Obviously, $\mathcal{L}$ is closed under union and cartesian product.

Proposition 3.3. *Assume that $L_1, L_2 \in \mathcal{L}$ are tree-tuple languages of ranks I_1 and I_2, respectively. Then we have:*

– *If $I_1 = I_2$, then $L_1 \cup L_2$ is in $\mathcal{L}$, and*
– *If $I_1 \cap I_2 = \emptyset$, then $L_1 \times L_2 \in \mathcal{L}$ where for $I = I_1 \cup I_2$, the* cartesian product *$L_1 \times L_2$ is defined as the set of all I-tuples*

$$L_1 \times L_2 = \{t \in T_\Sigma^I \mid t|_{I_i} \in L_i\}$$

The class of tree-tuple languages defined through constraint systems is *not* closed under intersection – even with a regular relation. The reason intuitively is that through intersection, we can construct the set of all complete binary trees which cannot be defined by any constraint system as considered here.

Finally, let us remark that $\mathcal{L}$ is closed under projections.

Proposition 3.4. *Assume that $L \subseteq T_\Sigma^I$ and $A \subseteq I$. then also the projection $\pi_A(L)$ of L onto A is in $\mathcal{L}$ where the* projection *$\pi_A(L)$ is defined by*

$$\pi_A(L) = \{t|_A \mid t \in L\}$$

Although not stated explicitly in the propositions 3.3 and 3.4, the constructions for union and cartesian product can be implemented in linear lime. The same clearly also holds for projection.

3.3 Join and Membership

Another operation on languages of tree-tuples is the *join* operation. Assume that I_1, I_2 are two index sets with $I_1 \cap I_2 = \{x\}$. The *join* of two languages $L_i \subseteq T_\Sigma^{I_i}$ is defined by

$$L_1 \bowtie L_2 = \{t \in T_\Sigma^I \mid t|_{I_i} \in L_i\}$$

where $I = I_1 \cup I_2$. In case that the index sets are disjoint, we may select elements $x_1 \in I_1$ and $x_2 \in I_2$ which are to be identified. In order to do so, we construct the cartesian product of the two languages, select all tuples t where $t_{x_1} = t_{x_2}$ and then project onto the set $I = I_1 \cup I_2 \backslash \{x_2\}$. This operation is denoted by $L_1 \bowtie_{x_1,x_2} L_2$. Note that this operation can also be implemented by renaming the component x_2 of the language L_2 to x_1 and then apply the ordinary join. In general, the family $\mathcal{L}$ will not be closed under joins. An exception is the case where $I_1 = \{x\}$ and L_1 is a regular set of trees. Then $L_1 \bowtie L_2 = \{t \in L_2 \mid t_x \in L_1\}$, and we find:

Proposition 3.5. *If $L_1 \subseteq T_\Sigma^{\{x\}}$ is regular, and $L \subseteq T_\Sigma^I$ is in $\mathcal{L}$ with $x \in I$, then $L' = L_1 \bowtie L$ is in $\mathcal{L}$ as well. Given a finite tree automaton of size m for L_1 and a constraint system of size n for L using a classified set of variables of width r, the construction of a constraint system for L' can be performed in time $\mathcal{O}(n \cdot (m+1)^r)$.*

Prop. 3.5 can be used to compute the *intersection* of cartesian products of regular sets with languages from $\mathcal{L}$.

Proposition 3.6. *Assume $I = \{x_1, \ldots, x_k\}$ for pairwise different indices x_i, $L_i \subseteq T_\Sigma^{\{x_i\}}, i = 1, \ldots, k$, are regular sets, and $L \subseteq T_\Sigma^I$ is in $\mathcal{L}$. Then the intersection $L' = (L_1 \times \ldots \times L_k) \cap L$ is in $\mathcal{L}$ as well. Given finite tree automata of joint size m for the L_i and a constraint system of size n for L using a classified set of variables of width r, the construction of a constraint system for L' can be performed in time $\mathcal{O}(n \cdot (m+1)^r)$.*

Indeed, prop. 3.6 follows directly from prop. 3.5, since

$$L' = L_1 \bowtie (L_2 \bowtie \ldots (L_k \bowtie L) \ldots)$$

In order to derive the given complexity bound we have to recall that each component of a class on the right-hand side can be used at most once on the left hand side – implying that, when keeping track of the states of components, we only have to track the state of one of the r automata for each component. Nonetheless, as the width of the set of variables of the constraint system for L occurs in the exponent of the complexity estimation, we conclude that the corresponding construction will only be feasible for "small" index sets I. Sadly enough, the complexity estimation cannot be easily improved. In order to see this, recall that there is a simple constraint system for the language

$$Id^k = \{(t, \ldots, t) \mid t \in T_\Sigma\} \subseteq T_\Sigma^k$$

The intersection of Id^k with the cartesian product $L_1 \times \ldots \times L_k$ of regular tree languages L_i is nonempty iff $L_1 \cap \ldots \cap L_k \neq \emptyset$. Deciding the emptiness of the intersection of a sequence of regular tree languages, however, is well-known to be complete for DEXPTIME [13]. Now since emptiness for our constraint systems can be decided in linear time, we have succeeded in reducing a DEXPTIME complete problem to our intersection construction.

Prop. 3.6, however, offers a conceptually simple method for solving the membership problem. Since singleton sets of trees are trivially regular, we obtain:

Theorem 3.7. *Given a language $L \subseteq T_\Sigma^I$ in $\mathcal{L}$ and a tree-tuple $t \in T_\Sigma^I$, it can be decided whether or not $t \in L$. Assuming that L is given through a constraint system of size n using a classified set of variables of width r, membership of t in L can be decided in time $\mathcal{O}(n \cdot (|t|+1)^r)$.*

Let us now return to prop. 3.5. The basic idea of the construction used in the proof there consists in the observation that, while building up tuples according to a constraint system, we very well may keep track of the states of a finite tree automaton on all components of classes that contribute to a given component.

Indeed, this idea can be generalized. A component x of a class X will be considered as *free* if its values trees are generated by the constraint system in a "regular way". By this we mean that no use is ever made of several components from the same variable, i.e., "internal synchronization".

Example 3.8. Consider the constraint system :

$$X \supseteq (a,\ f(1_1, 1_2))\ X_{id}$$

where X_{id} is defined as previously. In X, the second component is not free, since both arguments of f come from the same variable implying that their values are potentially inter-related and indeed, the projection of X onto its second argument is not regular. On the other hand, the first component is trivially free. Now let us add the constraint :

$$Y \supseteq s(1_2)\ X$$

Then the component of Y is not free either because the second component of X is used to define Y which is not free.

In order to make this notion formal, assume that we are given a constraint system $(\mathcal{X}, \mathcal{E})$ which is Greibach and non-copying. The relation $\mathsf{free} \subset N \times \mathcal{X}$ is given as the maximal relation $\mathcal{F}$ such that for each $(x, X) \in \mathcal{F}$, $x \in \rho(X)$ and for all constraints $X \supseteq \Box(X_1, \ldots, X_k)$ in $\mathcal{E}$ and $t = \Box_x$ the following holds:

1. If $(i, y) \in I$ occurs in t then $(y, X_i) \in \mathcal{F}$;
2. If $(i_1, y_1) \neq (i_2, y_2)$ occur in t then $i_1 \neq i_2$.

Since properties (1) and (2) are preserved by unions, free is well-defined. If $(x, X) \in \mathsf{free}$, we also say that component x is *free* in X. The second property above then essentially implies that there are no "internal synchronizations" in free components.

We call a component x free w.r.t. a language L (from $\mathcal{L}$) if there is a constraint system $(\mathcal{X}, \mathcal{E})$ with least solution σ such that $L = \sigma\, X$ for some $X \in \mathcal{X}$ where x is free in X.

Proposition 3.9. *If x is free in the language L (from $\mathcal{L}$), then the projection $\pi_{\{x\}}(L)$ of L onto the component x is a regular tree language.*

Here is the main result of this section.

Theorem 3.10. *Assume $L_i \subseteq T_\Sigma^{I_i}$, $i = 1, 2$, are in $\mathcal{L}$ where I_1 and I_2 are disjoint. Then the join $L_1 \bowtie_{x_1,x_2} L_2$ is in $\mathcal{L}$ whenever x_1 is free in L_1. Moreover if x_3 is free in L_2, then x_3 is still free in $L_1 \bowtie_{x_1,x_2} L_2$.*

A proof of this theorem can be found in the full version [3].

4 Application to Rewriting

For a binary relation on terms Rel and two regular tree languages L_1, L_2, the decidability question $Rel(L_1) \subseteq L_2$ has already been studied in [14] assuming Rel is a rewrite-like relation. Now, if $Rel \in \mathcal{L}$ (i.e. is defined by a non-copying constraint system), we also get a decidability result (the proof is in [3]).

Proposition 4.1. *If $L_1, Rel \in \mathcal{L}$ and L_2 is regular, then*

$$\left.\begin{array}{r} L_1 \textit{ is regular} \\ \textit{or the first component of } Rel \textit{ is free}^6 \end{array}\right\} \implies Rel(L_1) \subseteq L_2 \textit{ is decidable}$$

Note that this result can be used incrementally to compose relations. If the first component of each relation $Rel_1, \ldots, Rel_n$ is free, then $Rel_1(\ldots\ldots(Rel_n(L_1)\ldots) \subseteq L_2$ is still decidable.

Concerning rewrite-like relations, we can easily encode one-step rewriting, parallel rewriting by non-copying constraint systems (see the full version [3]). If the rewrite system is entirely linear we can also encode one-pass rewriting and one-pass root-started rewriting. Thus, if L_1 is regular, we get the same decidability results as in [14] (or weaker because of linearity). On the other hand, if L_1 is not regular, for example if L_1 is composed of the instances of a non-linear term, we get new results, assuming however left-linearity for one-step rewriting and parallel rewriting, to ensure the freeness of the first component of Rel.

Moreover, assuming left-linearity, we can compose n times the relation one-or-zero-step rewriting with itself, and we get the decidability of $\xrightarrow{\leq n} (L_1) \subseteq L_2$, where $\xrightarrow{\leq n}$ is the rewrite relation in no more than n steps.

5 Application to Concurrency

We introduce a new concurrency formalism, the *Bounded-Communication Process Calculus* (BCPC). The idea is : when running two processes in parallel, they cannot communicate to each other if either (or both) has performed too many statements (actions), because it lasts too long. In other words, communication channels are not kept indefinitely, and nothing new is tried after the time limit. We think that this formalism could apply to check some properties in communication protocols.

Since everything is assumed to be time-limited, and to simplify the formalization, we consider that process derivation rules are folded into bigger ones, that simulate the time limit. For example, given the derivation rules $\{x \xrightarrow{a} y||z,\ y \xrightarrow{b} t\}$ and considering that communication is not allowed beyond two actions, the derivation rules are replaced by $x \xrightarrow{c} t||z$ where c is a new action representing the sequence $a.b$. Thus, the process $x||t'$ derives in one step, with

6 I.e. the projection of Rel on the first component is a regular language.

action c, into $(t\|z)\|t'$, and $t\|z$ will not be allowed to communicate with t' because we consider that applying one rule means that the time limit is reached, i.e. the new subterm created by the rule cannot communicate with the rest of the world anymore. Obviously, the user may fold the rules as he likes, and may for instance consider that all actions do not have the same durations.

Starting from a process name x, we are interested in testing properties on processes derived from x. Properties are expressed by means of the temporal logic CTL on an infinite structure[7], restricted to operators EF and EX, and we show that the model-checking problem is decidable. As a consequence, the absence of deadlocks is decidable.

To prove this, we express the transitive closure $\rightarrow^*$ of the process-derivation relation by a constraint system[8], and apply constraint system properties. For readability, we first express $\rightarrow^*$ for the PA formalism (i.e. without communication), which we then extend to BCPC by introducing communication (also called synchronization).

5.1 PA

PA introduced in [1] is a process algebra which permits non-determinism, sequential and parallel compositions, and recursion.

Syntax and semantics. *Act* is a set of *action names*, $Act = \{a, b, c, \ldots\}$. $Const$ is a finite set of *process constants* (or process names), $Const = \{x, y, z, \ldots\}$. T is the set $\{t_1, t_2, \ldots\}$ of PA-terms defined by the following equation :

$$t_1, t_2 ::= 0 \;\mid\; x \;\mid\; t_1 \parallel t_2 \;\mid\; t_1 \,.\, t_2$$

where $x \in Const$. The interpretation of the syntax expressions is the following : 0 represents the process which performs no events, $t_1 \parallel t_2$ the parallel composition, and $t_1 \,.\, t_2$ represents the sequential composition.

A PA declaration is a finite family of recursive process rewrite rules :

$$\Delta = \{x_i \xrightarrow{a_i^j} t_i^j \mid x_i \in Const,\ t_i^j \in T,\ i = 1, \ldots, n,\ j = 1, \ldots, k_i\}$$

We define $\Delta(x) = \{t \mid (x \xrightarrow{a} t) \in \Delta\}$. $\Delta(x) = \emptyset$ denotes that there is no rule $(x \xrightarrow{a} t)$ belonging to Δ.

A family Δ of process rewrite rules determines a labeled transition relation $\rightarrow_\Delta \subseteq T \times Act \times T$. We omit the Δ and we note $t \xrightarrow{a} t'$ for $(t, a, t') \in \rightarrow$ with $t, t' \in T$ and $a \in Act$. The semantics is defined in Figure 1 by a Structural Operational Semantics [11]. In the parallel composition $\|$, the two processes t_1 and t_2 are evaluated independently. In the sequential composition $t_1 \,.\, t_2$, t_1 is first computed, and then t_2 is evaluated when the process t_1 is terminated. $Const(t)$ denotes the set of process constants occurring in term t. The predicate $Finished$ represents the information of process termination. A term t is finished when it contains no process constants: $Finished(t) = (Const(t) = \emptyset)$.

[7] States are process terms.

[8] For concision, in this section constraint system stands for non-copying constraint system.

$$x \xrightarrow{a} t \quad \text{if } (x \xrightarrow{a} t) \in \Delta$$

$$\frac{t_1 \xrightarrow{a} t_1'}{t_1 \parallel t_2 \xrightarrow{a} t_1' \parallel t_2} \qquad \frac{t_2 \xrightarrow{a} t_2'}{t_1 \parallel t_2 \xrightarrow{a} t_1 \parallel t_2'}$$

$$\frac{t_1 \xrightarrow{a} t_1'}{t_1 \,.\, t_2 \xrightarrow{a} t_1' \,.\, t_2} \qquad \frac{t_2 \xrightarrow{a} t_2'}{t_1 \,.\, t_2 \xrightarrow{a} t_1 \,.\, t_2'} \quad \text{if } \mathit{Finished}(t_1)$$

Fig. 1. The semantics

Constraint system. We use the standard notation $\to^*$ for transitive closure of the rewrite relation (labels are omitted). The set $Pre^*(t)$ denotes the iterated predecessors of the term t : $Pre^*(t) = \{t' \in T \mid t' \xrightarrow{*} t\}$ and the set $Post^*(t)$ denotes the iterated successors of the term t : $Post^*(t) = \{t' \in T \mid t \xrightarrow{*} t'\}$.

The constraint system G (axiom A) for $\to^*$ is presented[9] in Figure 2. We note $L(N)$ the language generated from the non-terminal N. $L(A)$ represents the language of pairs (t, t') where $t \in T$ and $t' \in Post^*(t)$. In this definition, we use $Sub(\Delta) = \{s \mid s \text{ is a subterm of } t, t \in \Delta(x), x \in Const\}$. We use the non-terminal A_t to generate $Post^*(t)$. For each $x \in Const$, the non-terminal X is introduced to define $Post^+(x)$. The non-terminals F, F_X and F_t play the same part as A, X and A_t except that they express only terminated processes (without constants).

$$\begin{array}{ll}
A \supseteq (0,0) & X \supseteq A_t \quad \forall t \;:\; (x \xrightarrow{a} t) \in \Delta \\
\left.\begin{array}{l} A \supseteq (x,x) \\ A \supseteq (x,X) \end{array}\right\} \forall x \in Const & A_0 \supseteq 0 \\
A \supseteq (1_1 \parallel 2_1, 1_2 \parallel 2_2)(A,A) & \left.\begin{array}{l} A_x \supseteq x \\ A_x \supseteq X \end{array}\right\} \forall x \in Const \\
A \supseteq (1_1 \,.\, 2_1, 1_2 \,.\, 2_2)(A,Id) & A_{t_1 \parallel t_2} \supseteq A_{t_1} \parallel A_{t_2} \quad \forall\, t_1 \parallel t_2 \in Sub(\Delta) \\
A \supseteq (1_1 \,.\, 2_1, 1_2 \,.\, 2_2)(F,A) & \left.\begin{array}{l} A_{t_1 \,.\, t_2} \supseteq A_{t_1} \,.\, t_2 \\ A_{t_1 \,.\, t_2} \supseteq F_{t_1} \,.\, A_{t_2} \end{array}\right\} \forall\, t_1 \,.\, t_2 \in Sub(\Delta)
\end{array}$$

$$\begin{array}{l}
F \supseteq (0,0) \\
F \supseteq (x, F_X) \quad \forall x \in Const \\
F \supseteq (1_1 \parallel 2_1, 1_2 \parallel 2_2)(F,F) \quad \forall\, t_1 \parallel t_2 \in Sub(\Delta) \\
F \supseteq (1_1 \,.\, 2_1, 1_2 \,.\, 2_2)(F,F) \quad \forall\, t_1 \,.\, t_2 \in Sub(\Delta)
\end{array}$$

$$\begin{array}{l}
F_X \supseteq F_t \quad \forall t \;:\; (x \xrightarrow{a} t) \in \Delta \\
F_0 \supseteq 0 \\
F_x \supseteq F_X \\
F_{t_1 \parallel t_2} \supseteq F_{t_1} \parallel F_{t_2} \quad \forall\, t_1 \parallel t_2 \in Sub(\Delta) \\
F_{t_1 \,.\, t_2} \supseteq F_{t_1} \,.\, F_{t_2} \quad \forall\, t_1 \,.\, t_2 \in Sub(\Delta)
\end{array}$$

Fig. 2. The constraint system G for $\to^*$

[9] Shortened notations are used, like $A \supseteq (x, X)$ which means $A \supseteq (x, 1_1)X$.

Theorem 5.1. *G generates exactly $\rightarrow^*$ (i.e. $L(A) = \{(t,t') \mid t \rightarrow^* t'\}$). Moreover G is regular.*

As a consequence, we get the regularity of $\rightarrow^*$, already proved in [9]. The proof of this theorem can be achieved using the following lemma :
Lemma 5.2.

$$\begin{array}{ll} & L(F) = \{(t,t') \mid t \rightarrow^* t' \wedge Finished(t')\} \\ L(X) = \{t' \mid x \rightarrow^+ t'\} & L(F_X) = \{t' \mid x \rightarrow^* t' \wedge Finished(t')\} \\ L(A_t) = \{t' \mid t \rightarrow^* t'\} = Post^*(t) & L(F_t) = \{t' \mid t \rightarrow^* t' \wedge Finished(t')\} \end{array}$$

The proof can be made by classical induction or using the results of [10,9]. In [10], tree automata techniques are used to compute $Post^*(t)$ and $Pre^*(t)$. Two families of tree languages are defined, as the least solution of a recursive equation set. They define $L'_t = Post^*(t)$ and L''_t is the restriction of L'_t to terminated processes. It is clear that we have $L(A_t) = L'_t$ and $L(F_t) = L''_t$. The non-terminal A is represented by $[I, R]$ and F by $[I', RT]$. In [9], automata for tree languages are replaced by automata for tree relations. A notion of cost is introduced. A regular tree constraint system for $Post^*(X)$ is defined. F is represented by I and A by F.

5.2 BCPC

Now $Act = \{a, b, c, \ldots, \overline{a}, \overline{b}, \overline{c}, \ldots\}$, $\Delta = \Delta_1 \cup \Delta_2$ where Δ_2 is a set of synchronized rules:

$$\Delta_1 = \{x_i \overset{a_i^j}{\rightarrow} t_i^j \mid x_i \in Const,\ t_i^j \in T,\ i = 1, \ldots, n,\ j = 1, \ldots, k_i\}$$

$$\Delta_2 = \{\left\{\begin{array}{l} y_i \overset{a_i^j}{\rightarrow} t_{i,1}^j \\ z_i \overset{\overline{a_i^j}}{\rightarrow} t_{i,2}^j \end{array}\right. \mid \begin{array}{l} y_i, z_i \in Const,\ t_{i,1}^j, t_{i,2}^j \in T, \\ i = 1, \ldots, n,\ j = 1, \ldots, p_i \end{array}\}$$

and

$$\Delta_1(x) = \{t \mid (x \overset{a}{\rightarrow} t) \in \Delta_1\} \qquad \Delta_2(y,z) = \{(t_1, t_2) \mid \left\{\begin{array}{l} y \overset{a}{\rightarrow} t_1 \\ z \overset{\overline{a}}{\rightarrow} t_2 \end{array}\right. \in \Delta_2\}$$

A synchronization is only allowed within a rule right-hand-side. In this paper, to avoid a conflict with the semantics of sequential composition, we assume that a synchronization is only allowed within a right-hand-side subterm that contains no sequential composition. We hope that in further work, by refining the semantics of synchronized process-derivation as well as the constraint system for $\rightarrow^*$, we will manage to remove this restriction.

Let *Rhs* be the set of the greatest[10] subterms of right-hand-sides of rules in Δ that contain no sequential composition. The semantics of $\rightarrow_\Delta$ is defined by Figure 1 (where a may also be replaced by ϵ) and:

$$s \overset{\epsilon}{\rightarrow} s[y/t_1, z/t_2] \quad \text{if} \left\{\begin{array}{l} y \overset{a}{\rightarrow} t_1 \\ z \overset{\overline{a}}{\rightarrow} t_2 \end{array}\right. \in \Delta_2 \text{ and } s \in Rhs$$

where $s[y/t_1, z/t_2]$ is obtained by replacing y by t_1 and z by t_2 in s.

[10] To avoid redundancies.

Example. For readability we first present an example. Consider

$$\Delta_1 = \{x \xrightarrow{a} y \| z,\ z \xrightarrow{a} y_2,\ y_1 \xrightarrow{a} 0\},\ \Delta_2 = \begin{cases} y \xrightarrow{a} (0\ .\ y_1) \\ z \xrightarrow{\overline{a}} 0 \end{cases}$$

Then $Rhs = \{y \| z\}$. The constraint system G' for $\rightarrow^*$ is presented in Figure 3. G' also includes the rules of Figure 2. The non-terminal YZ permits a synchronization of y and z.

Theorem 5.3. *G' generates exactly $\rightarrow^*$.*

Note that G' is not regular because constraints $A \supseteq \ldots$ and $F \supseteq \ldots$ introduce internal synchronizations. The proof comes from Lemma 5.2 and Lemma 5.4.
Lemma 5.4.

$L(YZ) = \{(t'_1, t'_2) \mid y \parallel z \rightarrow^+ t'_1 \parallel t'_2\}$
$L(H_{YZ}) = \{(t'_1, t'_2) \mid\ y \parallel z \rightarrow^* t'_1 \parallel t'_2 \ \wedge\ Finished(t'_1) \ \wedge\ Finished(t'_2)\}$
$L(B_{yz}) = \{(t'_1, t'_2) \mid y \parallel z \rightarrow^* t'_1 \parallel t'_2\},\ L(B_{t_1 t_2}) = \{(t'_1, t'_2) \mid t_1 \rightarrow^* t'_1 \wedge t_2 \rightarrow^* t'_2\}$

The proof can be made by induction on the length of $\rightarrow^*$.

$A \supseteq (y \parallel z, 1_1 \parallel 1_2)YZ$

$YZ \supseteq B_{t_1 t_2} \quad \forall t_1, t_2 : \begin{cases} y \xrightarrow{a} t_1 \\ z \xrightarrow{\overline{a}} t_2 \end{cases} \in \Delta_2$

$B_{t_1 t_2} \supseteq (A_{t_1}, A_{t_2})$

$F \supseteq (y \parallel z, 1_1 \parallel 1_2)H_{YZ}$

$H_{YZ} \supseteq H_{t_1 t_2} \quad \forall t_1, t_2 \ : \ \begin{cases} y \xrightarrow{a} t_1 \\ z \xrightarrow{\overline{a}} t_2 \end{cases} \in \Delta_2$

$H_{t_1 t_2} \supseteq (F_{t_1}, F_{t_2})$
$A_{y\|z} \supseteq (1_1 \parallel 1_2)YZ$

$F_{y\|z} \supseteq (1_1 \parallel 1_2)H_{YZ}$

Fig. 3. The additional constraints of G' for $\rightarrow^*$

The following process derivations:

$$x \rightarrow y \parallel z \rightarrow (0\ .\ y_1) \parallel 0 \rightarrow (0\ .\ 0) \parallel 0 \qquad x \rightarrow y \parallel z \rightarrow y \parallel y_2$$

are expressed by G' in the following way:

$A \supseteq (x, x) \quad A \supseteq (x, X) \qquad X \supseteq A_{y\|z}$ because $(x \xrightarrow{a} y \parallel z) \in \Delta_1$
$A_{y\|z} \supseteq A_y \parallel A_z \qquad A_y \supseteq y \quad A_z \supseteq z \quad A_z \supseteq Z$
$Z \supseteq A_{y_2}$ because $(z \xrightarrow{a} y_2) \in \Delta_1 \quad A_{y_2} \supseteq y_2$

So, we can prove : $A \supseteq (x,x),\ (x, y \parallel z),\ (x, y \parallel y_2)$

$$\begin{array}{ll} A_{y\parallel z} \supseteq (1_1 \parallel 1_2)YZ & YZ \supseteq B_{0.y_1\ 0} \text{ because } \begin{cases} y \xrightarrow{a} 0\ .\ y_1 \\ z \xrightarrow{\overline{a}} 0 \end{cases} \in \Delta_2 \\ B_{0.y_1\ 0} \supseteq (A_{0.y_1}, A_0) & A_{0.y_1} \supseteq A_0\ .\ y_1 \quad A_{0.y_1} \supseteq F_0\ .\ A_{y_1} \\ A_0 \supseteq 0 \quad F_0 \supseteq 0 & A_{y_1} \supseteq y_1 \quad A_{y_1} \supseteq Y_1 \\ Y_1 \supseteq 0 \text{ because } (y_1 \xrightarrow{a} 0) \in \Delta_1 & \end{array}$$

So, we can prove : $A \supseteq (x, 0\ .\ y_1 \parallel 0),\ (x, 0\ .\ 0 \parallel 0)$

General case. See the full version [3].

Model-checking. The model-checking problem solved in [10] for PA is still decidable for BCPC.

We consider a set $Prop = \{P_1, P_2, \ldots\}$ of *atomic propositions.* For $P \in Prop$, let $Mod(P)$ denote the set of PA-processes for which P holds. $Mod(P)$ is always supposed to be a regular tree-language.

The EF-logic has the following syntax :

$$\varphi ::= P \mid \neg\varphi \mid \varphi \wedge \varphi' \mid \mathsf{EX}_\varphi \mid \mathsf{EF}_\varphi$$

and semantics :

$$\begin{array}{rl} t \models P \Leftrightarrow t \in Mod(P) & \\ t \models \neg\varphi \Leftrightarrow t \not\models \varphi & t \models \mathsf{EX}_\varphi \Leftrightarrow t' \models \varphi \text{ for some } t \to t' \\ t \models \varphi \wedge \varphi' \Leftrightarrow t \models \varphi \text{ and } t \models \varphi' & t \models \mathsf{EF}_\varphi \Leftrightarrow t' \models \varphi \text{ for some } t \to^* t' \end{array}$$

Definition 5.5. *The* model-checking problem *consists in testing whether* $t \models \varphi$ *for given* t *and* φ.

If we define $Mod(\varphi) = \{s \in T \mid s \models \varphi\}$, the model-checking problem for s and φ amounts to test whether $t \in Mod(\varphi)$. Trivially, Mod satisfies :

$$\begin{array}{rl} Mod(\neg\varphi) = T - Mod(\varphi) & Mod(\mathsf{EX}_\varphi) = Pre(Mod(\varphi)) \\ Mod(\varphi \wedge \varphi') = Mod(\varphi) \cap Mod(\varphi') & Mod(\mathsf{EF}_\varphi) = Pre^*(Mod(\varphi)) \end{array}$$

Theorem 5.6. *For every EF-formula* φ*,* $Mod(\varphi)$ *is a regular language.*

Proof. See the full version [3].

This result may seem surprising, because it uses $\to^*$ as an intermediate language, which is a non-regular synchronized language. It comes from the fact that the first component of G' is free, i.e. the projection of $\to^*$ on the first component is regular.

So, and thanks to membership test, the model-checking problem is decidable. As a consequence, the absence of deadlocks is decidable. Indeed, starting from a process t, there is a deadlock iff there exists an iterated successor t' of t s.t. t' has no successors and t' is not a finished process. Thus, there is no deadlocks iff $t \models \neg\mathsf{EF}(\neg\mathsf{EX}(True) \wedge \neg Finished)$, where $Mod(True) = T$ and $Mod(Finished) = \{t \in T \mid Finished(t)\}$ are regular languages.

6 Further Work

Synchronized languages defined by constraint systems are not closed by intersection, and therefore not closed by complement[11]. In further work, we will define a subclass closed by intersection, and that preserves all properties of synchronized languages. This should give rise to further applications.

BCPC allows rendez-vous of only two processes. This can be trivially extended to finitely many processes, since we can handle tuples of any size. Removing the restriction on Rhs is more difficult, but we hope for it. On the other hand, we can test properties on processes, not on actions. However, it should be possible to take actions into account by using triple languages for $\xrightarrow{a}$ and $\xrightarrow{\alpha}^*$.

References

1. J.C.M. Baeten and W.P. Weijland. Process algebra. In *Cambridge Tracts in Theoretical Computer Science*, volume 18, 1990.
2. B. Courcelle. The Monadic Second-Order Logic of Graphs I, Recognizable Sets of Finite Graphs. In *Inf. Comp.*, volume 85, pages 12–75, 1990.
3. V. Gouranton, P. Réty, and H. Seidl. Synchronized Tree Languages Revisited and New Applications. Research Report 2000-16, LIFO, 2000. http://www.univ-orleans.fr/SCIENCES/LIFO/Members/rety/publications.html.
4. S. Limet and P. Réty. E-Unification by Means of Tree Tuple Synchronized Grammars. In *Proceedings of 6th Colloquium on Trees in Algebra and Programming*, volume 1214 of *LNCS*, pages 429–440. Springer-Verlag, 1997.
5. S. Limet and P. Réty. E-Unification by Means of Tree Tuple Synchronized Grammars. *Discrete Mathematics and Theoritical Computer Science (http://dmtcs.loria.fr/)*, 1:69–98, 1997.
6. S. Limet and P. Réty. Solving Disequations modulo some Class of Rewrite Systems. In *Proceedings of 9th Conference on Rewriting Techniques and Applications, Tsukuba (Japon)*, volume 1379 of *LNCS*, pages 121–135. Springer-Verlag, 1998.
7. S. Limet and P. Réty. A New Result about the Decidability of the Existential One-step Rewriting Theory. In *Proceedings of 10th Conference on Rewriting Techniques and Applications, Trento (Italy)*, volume 1631 of *LNCS*. Springer-Verlag, 1999.
8. S. Limet and F. Saubion. On partial validation of logic programs. In M. Johnson, editor, *proc of the 6th Conf. on Algebraic Methodology and Software Technology, Sydney (Australia)*, volume 1349 of *LNCS*, pages 365–379. Springer Verlag, 1997.
9. D. Lugiez and P. Schnoebelen. Decidable firt-order transition logics for PA-processes. In Springer, editor, *ICALP*, LNCS, Geneva, Switzerland, July 2000.
10. D. Lugiez and P. Schnoebelen. The regular viewpoint on PA-processes. *Theoretical Computer Science*, 2000.
11. G.D. Plotkin. A structural approach of operational semantics. Technical Report FN-19, DAIMI, Aarhus University, Denmark, 1981.
12. F. Saubion and I. Stéphan. On Implementation of Tree Synchronized Languages. In *Proceedings of 10th Conference on Rewriting Techniques and Applications, Trento (Italy)*, LNCS. Springer-Verlag, 1999.

[11] Closure by union and complement implies closure by intersection.

13. H. Seidl. Haskell Overloading is DEXPTIME Complete. *Information Processing Letters*, 52(2):57–60, 1994.
14. F. Seynhaeve, S. Tison, and M. Tommasi. Homomorphisms and concurrent term rewriting. In G. Ciobanu and G. Paun, editors, *Proceedings of the twelfth International Conference on Fundamentals of Computation theory*, number 1684 in Lecture Notes in Computer Science, pages 475–487, Iasi, Romania, 1999.