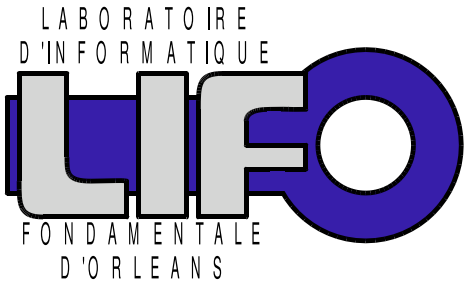

4 rue Léonard de Vinci
BP 6759
F-45067 Orléans Cedex 2
FRANCE
http://www.univ-orleans.fr/lifo

Rapport de Recherche

Synchronized Tree Languages for Reachability in Non-right-linear Term Rewrite Systems (full version)

Yohan Boichut, Vivien Pelletier and Pierre Réty
LIFO, Université d'Orléans

Rapport n^o **RR-2015-03**

Synchronized Tree Languages for Reachability in Non-right-linear Term Rewrite Systems

Yohan Boichut, Vivien Pelletier and Pierre Réty

LIFO - Université d'Orléans, B.P. 6759, 45067 Orléans cedex 2, France
{yohan.boichut, vivien.pelletier, pierre.rety}@univ-orleans.fr

Abstract. Over-approximating the descendants (successors) of an initial set of terms under a rewrite system is used in reachability analysis. The success of such methods depends on the quality of the approximation. Regular approximations (i.e. those using finite tree automata) have been successfully applied to protocol verification and Java program analysis. In [9, 2], non-regular approximations have been shown more precise than regular ones. In [3] (fixed version of [2]), we have shown that sound over-approximations using synchronized tree languages can be computed for left-and-right-linear term rewriting systems (TRS). In this paper, we present two new contributions extending [3]. Firstly, we show how to compute at least all innermost descendants for any left-linear TRS. Secondly, a procedure is introduced for computing over-approximations independently of the applied rewrite strategy for any left-linear TRS.

Keywords: tree languages, term rewriting, reachability analysis.

The reachability problem $R^*(I) \cap Bad \stackrel{?}{=} \emptyset$ is a well-known undecidable problem, where I is an initial set of terms, Bad is a set of *forbidden* terms and $R^*(I)$ denotes the terms issued from I using the rewrite system R. Some techniques compute regular over-approximations of $R^*(I)$ in order to show that no term of Bad is reachable from I [7, 6, 1, 4].

In [5], we have defined a reachability problem for which none of those techniques works. In [3] (corrected version of [2]), we have described a technique for computing non-regular approximations using synchronized tree languages. This technique can handle the reachability problem of [5]. These synchronized tree languages [10, 8] are recognized using CS-programs [11], i.e. a particular class of Horn clauses. From an initial CS-program $Prog$ and a linear term rewrite system (TRS) R, another CS-program $Prog'$ is computed in such a way that its *language* represents an over-approximation of the set of terms (called descendants) reachable by rewriting using R, from the terms of the language of $Prog$. This algorithm is called completion.

In this paper, we present two new results that hold even if the TRS is not right-linear:

1. We show that a slight modification of completion gives an over-approximation of the descendants obtained with an innermost strategy (see Section 2).

2. We introduce a technique for over-approximating[1] copying[2] clauses by non-copying ones, so that all descendants (not only the innermost ones) are obtained (see Section 3).

1 Preliminaries

Consider two disjoint sets, Σ a *finite ranked alphabet* and Var a set of variables. Each symbol $f \in \Sigma$ has a unique arity, denoted by $ar(f)$. The notions of *first-order term*, *position* and *substitution* are defined as usual. Given two substitutions σ and σ', $\sigma \circ \sigma'$ denotes the substitution such that for any variable x, $\sigma \circ \sigma'(x) = \sigma(\sigma'(x))$. T_Σ denotes the set of ground terms (without variables) over Σ. For a term t, $Var(t)$ is the set of variables of t, $Pos(t)$ is the set of positions of t. For $p \in Pos(t)$, $t(p)$ is the symbol of $\Sigma \cup Var$ occurring at position p in t, and $t|_p$ is the subterm of t at position p. The term t is *linear* if each variable of t occurs only once in t. The term $t[t']_p$ is obtained from t by replacing the subterm at position p by t'. $PosVar(t) = \{p \in Pos(t) \mid t(p) \in Var\}$, $PosNonVar(t) = \{p \in Pos(t) \mid t(p) \notin Var\}$.

A *rewrite rule* is an oriented pair of terms, written $l \to r$. We always assume that l is not a variable, and $Var(r) \subseteq Var(l)$. A *rewrite system* R is a finite set of rewrite rules. *lhs* stands for left-hand-side, *rhs* for right-hand-side. The rewrite relation $\to_R$ is defined as follows: $t \to_R t'$ if there exist a position $p \in PosNonVar(t)$, a rule $l \to r \in R$, and a substitution θ s.t. $t|_p = \theta(l)$ and $t' = t[\theta(r)]_p$. $\to_R^*$ denotes the reflexive-transitive closure of $\to_R$. t' is a *descendant* of t if $t \to_R^* t'$. If E is a set of ground terms, $R^*(E)$ denotes the set of descendants of elements of E. The rewrite rule $l \to r$ is *left (resp. right) linear* if l (resp. r) is linear. R is *left (resp. right) linear* if all its rewrite rules are left (resp. right) linear. R is *linear* if R is both left and right linear.

1.1 CS-Program

In the following, we consider the framework of *pure logic programming*, and the class of synchronized tree-tuple languages defined by CS-clauses [11, 12]. Given a set $Pred$ of *predicate* symbols; *atoms*, *goals*, *bodies* and *Horn-clauses* are defined as usual. Note that both *goals* and *bodies* are sequences of atoms. We will use letters G or B for sequences of atoms, and A for atoms. Given a goal $G = A_1, \ldots, A_k$ and positive integers i, j, we define $G|_i = A_i$ and $G|_{i.j} = (A_i)|_j = t_j$ where $A_i = P(t_1, \ldots, t_n)$.

Definition 1. *Let B be a sequence of atoms.*
B is flat *if for each atom $P(t_1, \ldots, t_n)$ of B, all terms $t_1, \ldots, t_n$ are variables.*

[1] This approximation is often exact, but not always. This is due to the fact that a tree language expressed by a copying CS-program cannot always be expressed by a non-copying one.

[2] I.e. clause heads are not linear.

B is linear *if each variable occurring in B (possibly at sub-term position) occurs only once in B. So the empty sequence of atoms (denoted by $\emptyset$) is flat and linear.*

A CS-clause[3] *is a Horn-clause $H \leftarrow B$ s.t. B is flat and linear. A* CS-program *$Prog$ is a logic program composed of CS-clauses. Variables contained in a CS-Clause have to occur only in this clause. $Pred(Prog)$ denotes the set of predicate symbols of $Prog$. Given a predicate symbol P of arity n, the tree-(tuple) language generated by P is $L_{Prog}(P) = \{\boldsymbol{t} \in (T_\Sigma)^n \mid P(\boldsymbol{t}) \in Mod(Prog)\}$, where T_Σ is the set of ground terms over the signature Σ and $Mod(Prog)$ is the least Herbrand model of $Prog$. $L_{Prog}(P)$ is called* synchronized language.

The following definition describes syntactic properties over CS-clauses.

Definition 2. *A CS-clause $P(t_1, \ldots, t_n) \leftarrow B$ is :*

- empty *if $\forall i \in \{1, \ldots, n\}$, t_i is a variable.*
- normalized *if $\forall i \in \{1, \ldots, n\}$, t_i is a variable or contains only one occurrence of function-symbol.*
- non-copying *if $P(t_1, \ldots, t_n)$ is linear.*

A CS-program is normalized *and* non-copying *if all its clauses are.*

Example 1. Let x, y, z be variables. $P(x) \leftarrow Q(f(x))$ is not a CS-clause.
$P(x, y, z) \leftarrow Q(x, y, z)$ is a CS-clause, and is empty, normalized and non-copying.
The CS-clause $P(f(x), y, g(x, z)) \leftarrow Q_1(x), Q_2(y, z)$ is normalized and copying.
$P(f(g(x)), y) \leftarrow Q(x)$ is not normalized.

Given a CS-program, we focus on two kinds of derivations.

Definition 3. *Given a logic program $Prog$ and a sequence of atoms G,*

- *G derives into G' by a* resolution *step if there exist a clause $H \leftarrow B$ in $Prog$ and an atom $A \in G$ such that A and H are unifiable by the most general unifier σ (then $\sigma(A) = \sigma(H)$) and $G' = \sigma(G)[\sigma(A) \leftarrow \sigma(B)]$. It is written $G \rightsquigarrow_\sigma G'$.*
- *G* rewrites *into G' if there exist a clause $H \leftarrow B$ in $Prog$, an atom $A \in G$, and a substitution σ, such that $A = \sigma(H)$ (A is not instantiated by σ) and $G' = G[A \leftarrow \sigma(B)]$. It is written $G \rightarrow_\sigma G'$.*

Sometimes, we will write $G \rightsquigarrow_{[H \leftarrow B, \sigma]} G'$ or $G \rightarrow_{[H \leftarrow B, \sigma]} G'$ to indicate the clause used by the step.

Example 2. Let $Prog = \{P(x_1, g(x_2)) \leftarrow P'(x_1, x_2).\ P(f(x_1), x_2) \leftarrow P''(x_1, x_2).\}$, and consider $G = P(f(x), y)$. Thus, $P(f(x), y) \rightsquigarrow_{\sigma_1} P'(f(x), x_2)$ with $\sigma_1 = [x_1/f(x), y/g(x_2)]$ and $P(f(x), y) \rightarrow_{\sigma_2} P''(x, y)$ with $\sigma_2 = [x_1/x, x_2/y]$.

[3] In former papers, synchronized tree-tuple languages were defined thanks to sorts of grammars, called constraint systems.Thus "CS" stands for Constraint System.

Note that for any atom A, if $A \to B$ then $A \rightsquigarrow B$. On the other hand, $A \rightsquigarrow_\sigma B$ implies $\sigma(A) \to B$. Consequently, if A is ground, $A \rightsquigarrow B$ implies $A \to B$.

We consider the transitive closure $\rightsquigarrow^+$ and the reflexive-transitive closure $\rightsquigarrow^*$ of $\rightsquigarrow$.

For both derivations, given a logic program $Prog$ and three sequences of atoms G_1, G_2 and G_3 :

- if $G_1 \rightsquigarrow_{\sigma_1} G_2$ and $G_2 \rightsquigarrow_{\sigma_2} G_3$ then one has $G_1 \rightsquigarrow^*_{\sigma_2 \circ \sigma_1} G_3$;
- if $G_1 \to_{\sigma_1} G_2$ and $G_2 \to_{\sigma_2} G_3$ then one has $G_1 \to^*_{\sigma_2 \circ \sigma_1} G_3$.

In the remainder of the paper, given a set of CS-clauses $Prog$ and two sequences of atoms G_1 and G_2, $G_1 \rightsquigarrow^*_{Prog} G_2$ (resp. $G_1 \to^*_{Prog} G_2$) also denotes that G_2 can be derived (resp. rewritten) from G_1 using clauses of $Prog$.

It is well known that resolution is complete.

Theorem 1. *Let A be a ground atom. $A \in Mod(Prog)$ iff $A \rightsquigarrow^*_{Prog} \emptyset$.*

1.2 Computing descendants

Due to the lack of space, we just give the main ideas using an example. See [2] for a formal description.

Example 3. Let $R = \{f(x) \to g(h(x))\}$ and let $I = \{f(a)\}$ generated by Predicate P in the CS-program $Prog = \{P(f(x)) \leftarrow Q(x).\ Q(a) \leftarrow\}$.
Note that $R^*(I) = \{f(a), g(h(a))\}$.

To simulate the rewrite step $f(a) \to g(h(a))$, we consider the rewrite-rule left-hand-side $f(x)$. We can see that $P(f(x)) \to_{Prog} Q(x)$ and $P(f(x)) \to_R P(g(h(x)))$. Then the clause $P(g(h(x))) \leftarrow Q(x)$ is called *critical pair*[4]. This critical pair is not *convergent* (in $Prog$) because $\neg(P(g(h(x))) \to^*_{Prog} Q(x))$. To get the descendants, the critical pairs should be convergent. Let $Prog' = Prog \cup \{P(g(h(x))) \leftarrow Q(x)\}$. Now the critical pair is convergent in $Prog'$, and note that the predicate P of $Prog'$ generates $R^*(I)$.

Since critical pairs are computed only at root positions, we consider only normalized CS-programs, and $Prog'$ is not normalized. The critical pair can be normalized using a new predicate symbol, and replaced by normalized clauses $P(g(y)) \leftarrow Q_1(y).\ Q_1(h(x)) \leftarrow Q(x)$. This is the role of Function norm in the completion algorithm below.

In general, adding a critical pair (after normalizing it) into the CS-program may create new critical pairs, and the completion process may not terminate. To force termination, two bounds *predicate-limit* and *arity-limit* are fixed. If *predicate-limit* is reached, Function norm should re-use existing predicates instead of creating new ones. If a new predicate symbol is created whose arity[5] is greater than *arity-limit*, then this predicate has to be cut by Function norm into several predicates whose arities do not exceed *arity-limit*.

[4] In former work, a critical pair was a pair. Here it is a clause since we use logic programs.

[5] The number of arguments.

Before normalizing a critical pair $H \leftarrow B$ (more precisely at the beginning of Function norm), for efficiency we first try to reduce H (into some H') using the CS-clauses of $Prog$. This mechanism is called *simplification*.

On the other hand, for a given CS-program, the number of critical pairs may be infinite. Function $\mathsf{removeCycles}$ modifies some clauses so that the number of critical pairs is finite.

Definition 4 ([3]). *Let arity-limit and predicate-limit be positive integers. Let R be a linear rewrite system, and $Prog$ be a finite, normalized and non-copying CS-program. The completion process is defined by:*
Function $\mathsf{comp}_R(Prog)$
  $Prog = \mathsf{removeCycles}(Prog)$
  while *there exists a non-convergent critical pair $H \leftarrow B$ in $Prog$* ***do***
    $Prog = \mathsf{removeCycles}(Prog \cup \mathsf{norm}_{Prog}(H \leftarrow B))$
  end while
  return *Prog*

The following results show that an over-approximation of the descendants is computed.

Theorem 2 ([3]). *Let R be a left-linear*[6] *rewrite system and $Prog$ be a normalized non-copying CS-program.*
If all critical pairs are convergent, then $Mod(Prog)$ is closed under rewriting by R, i.e. $(A \in Mod(Prog) \wedge A \rightarrow_R^* A') \implies A' \in Mod(Prog)$.

Theorem 3 ([3]). *Let R be a linear rewrite system and $Prog$ be a normalized non-copying CS-program. Function* comp *always terminates, and all critical pairs are convergent in $\mathsf{comp}_R(Prog)$.*
Moreover, $R^*(Mod(Prog)) \subseteq Mod(\mathsf{comp}_R(Prog))$.

2 Computing innermost descendants

Starting from a non-copying program $Prog$ and given a left-linear TRS R, using the completion algorithm presented in the previous section we may obtain a copying final program $Prog'$. Consequently, the language accepted by $Prog'$ may not be closed under rewriting i.e. $Prog'$ may not recognize an over-approximation of the descendants. Example 4 illustrates this problem.

Example 4. Let $Prog = \{P(g(x)) \leftarrow Q(x).\ Q(a) \leftarrow\}$ and $R = \{a \rightarrow b,\ g(x) \rightarrow f(x,x)\}$. Performing the completion algorithm detailed in Definition 4 returns $\mathsf{comp}_R(Prog) = \{P(g(x)) \leftarrow Q(x).\ P(f(x,x)) \leftarrow Q(x).\ Q(a) \leftarrow .\ Q(b) \leftarrow\}$. Note that $P(f(a,b)) \notin Mod(\mathsf{comp}_R(Prog))$ although $P(g(a)) \in Mod(Prog)$ and $P(g(a)) \rightarrow_R^* P(f(a,b))$.

[6] From a theoretical point of view, left-linearity is sufficient when every critical pair is convergent. However, to make every critical pair convergent by completion, full linearity is necessary (see Theorem 3).

Thus, some descendants of $Mod(Prog)$ are missing in $Mod(\mathsf{comp}_R(Prog))$. However, all descendants obtained by innermost rewriting (subterms are rewritten at first) are in $Mod(\mathsf{comp}_R(Prog))$, since the only innermost rewrite derivation issued from $g(a)$ is $g(a) \to_R^{in} g(b) \to_R^{in} f(b,b)$.

In this section, we show that with a slight modification of [3], if the initial CS-program $Prog$ is non-copying and R is left-linear (and not necessarily right-linear), we can perform reachability analysis for innermost rewriting. Theorem 5 shows that, in that case, we compute at least all the descendants obtained by innermost rewriting. To get this result, it has been necessary to prove a result about closure under innermost rewriting (Theorem 4).

To prove these results, additional definitions are needed. Indeed, to perform innermost rewriting, the rewrite steps are done on terms whose subterms are irreducible (cannot be rewritten). However, for a given TRS, the property of irreducibility is not preserved by instantiation, i.e. if a term t and a substitution θ are irreducible, then θt is not necessarily irreducible. This is why we need to consider a stronger property.

Definition 5. *Let R be a TRS. A term t is* strongly irreducible *(by R) if for all $p \in PosNonVar(t)$, for all $l \to r \in R$, $t|_p$ and l are not unifiable.*
A substitution θ is strongly irreducible *if for all $x \in Var$, θx is strongly irreducible.*

Lemma 1. *If t is strongly irreducible, then t is irreducible.*

Proof. By contrapositive. If $t \to_{[p,l\to r,\sigma]} t'$, then $t|_p = \sigma l$. Since it is assumed that $Var(t) \cap Var(l) = \emptyset$, then $t|_p$ and l are unifiable by σ.

Lemma 2. *If t is strongly irreducible, then for all $p \in Pos(t)$, $t|_p$ is strongly irreducible.*
For a substitution θ, if θt is strongly irreducible, then for all $x \in Var(t)$, θx is strongly irreducible (but t is not necessarily strongly irreducible).

Proof. Obvious.

Example 5. Let $t = f(x)$, $\theta = (x/a)$, $R = \{f(b) \to b\}$. Thus $\theta t = f(a)$ is strongly irreducible whereas t is not.

Corollary 1. *For substitutions α, θ, if $\alpha.\theta$ is strongly irreducible, then α is strongly irreducible.*

Note that the previous definitions and lemmas trivially extend to atoms and atom sequences.

Lemma 3. *(closure by instantiation) If t is strongly irreducible and θ is irreducible, then θt is irreducible.*

Proof. By contrapositive. If $\theta t \to_{[p,l\to r,\sigma]} t'$, then $(\theta t)|_p = \sigma l$.

- If $p \notin PosNonVar(t)$, then there exist a variable x and a position p' s.t. $(\theta x)|_{p'} = \sigma l$. Then θ is reducible.

- Otherwise, $\theta(t|_p) = \sigma l$. Then $t|_p$ and l are unifiable, hence t is not strongly irreducible.

Example 6. Let $t = f(x)$, $\theta = (x/g(y))$, and $R = \{g(a) \to b\}$. Thus t is strongly irreducible, θ is irreducible, and $\theta t = f(g(y))$ is irreducible. Note that θt is not strongly irreducible.

Before introducing two families of derivations, we show in Example 7 that performing the completion, as presented in Section 1.2, with a non-right-linear TRS may introduce copying clauses, and some innermost descendants may be missing.

Example 7. Let $R = \{f(x) \to g(h(x), h(x)),\ i(x) \to g(x, x),\ h(a) \to b\}$, and $Prog$ be the initial non-copying program:
$Prog = \{P(i(x)) \leftarrow Q_1(x).\ Q_1(a) \leftarrow .\ P(f(x)) \leftarrow Q_2(x).\ Q_2(a) \leftarrow\}$. We start with $Prog' = \emptyset$. The completion procedure computes the critical pairs:

1. $P(g(x, x)) \leftarrow Q_1(x)$ and add it into $Prog'$,
2. $P(g(h(x), h(x))) \leftarrow Q_2(x)$, which could be simplified into:
   $Q_1(h(x)) \leftarrow Q_2(x)$, which is added into $Prog'$,
3. $Q_1(b) \leftarrow$, which is added into $Prog'$.

No more critical pairs are detected, thus all critical pairs are convergent in $Prog'' = Prog \cup Prog'$. However $P(f(a)) \to_R P(g(h(a), h(a))) \to_R P(g(b, h(a)))$ by an innermost derivation, whereas $P(f(a)) \in Mod(Prog)$ and $P(g(b, h(a))) \notin Mod(Prog'')$.
Actually, the clause $P(g(x, x)) \leftarrow Q_1(x)$ prevents the reduction of $P(g(b, h(a)))$ and consequently, it is impossible to get the set of all innermost-descendants up to now. Now, we introduce two families of derivations, i.e. NC and SNC, which allow us to compute every innermost descendant. For an atom H, $Var^{mult}(H)$ denotes the set of the variables that occur several times in H. For instance, $Var^{mult}(P(f(x, y), x, z)) = \{x\}$.

Definition 6. *Let A be an atom (A may contain variables).*
The step $A \leadsto_{[H \leftarrow B, \sigma]} G$ is NC (resp. SNC[7]) if for all $x \in Var^{mult}(H)$, σx is irreducible (resp. strongly irreducible) by R.
A derivation is NC (resp. SNC) if all its steps are.

Remark 1.

- SNC implies NC.
- If the clause $H \leftarrow B$ is non-copying, then the step $A \leadsto_{[H \leftarrow B, \sigma]} G$ is SNC (and NC).

Example 8. Consider the clause $P(g(x, x)) \leftarrow Q(x)$ and $R = \{h(a) \to b\}$. The step $P(g(h(y), h(y))) \leadsto_{[(x/h(y)]} Q(h(y))$ is NC ($h(y)$ is irreducible), but it is not SNC ($h(y)$ is not strongly irreducible).

[7] NC stands for Non-Copying. SNC stands for Strongly Non-Copying.

Lemma 4. *If $A \rightarrow_{[H \leftarrow B,\sigma]} G$ is SNC and $\forall x \in Var^{mult}(H)$, $\forall y \in Var(\sigma(x))$, θy is irreducible, then $\theta A \rightarrow_{[H \leftarrow B,\theta.\sigma]} \theta G$ is NC.*

Proof. Let $x \in Var^{mult}(H)$. Then σx is strongly irreducible. From Lemma 3, $\theta.\sigma(x)$ is irreducible. Therefore $\theta A \rightarrow_{[H \leftarrow B,\theta.\sigma]} \theta G$ is NC.

Lemma 5. *If $\sigma' A \rightsquigarrow_{[H \leftarrow B,\gamma]} G$ is NC, then $A \rightsquigarrow_{[H \leftarrow B,\theta]} G'$ is NC and there exists a substitution α s.t. $\alpha G' = G$ and $\alpha.\theta = \gamma.\sigma'$.*

Proof. From the well-known resolution properties, we get $A \rightsquigarrow_{[H \leftarrow B,\theta]} G'$ and there exists a substitution α s.t. $\alpha G' = G$ and $\alpha.\theta = \gamma.\sigma'$.
Now, if $A \rightsquigarrow_{[H \leftarrow B,\theta]} G'$ is not NC, then there exists $x \in Var^{mult}(H)$ s.t. θx is reducible. Then $\gamma x = \gamma.\sigma'(x) = \alpha.\theta(x)$ is reducible. Therefore $\sigma' A \rightsquigarrow_{[H \leftarrow B,\gamma]} G$ is not NC, which is impossible.

Let us now define a subset of $Mod(Prog)$.

Definition 7. *Let $Prog$ be a CS-program and R be a rewrite system. $Mod^R_{NC}(Prog)$ is composed of the ground atoms A such that there exists a NC derivation $A \rightsquigarrow^* \emptyset$.*

Remark 2.

- $Mod^R_{NC}(Prog) \subseteq Mod(Prog)$.
- If $Prog$ is non-copying, then $Mod^R_{NC}(Prog) = Mod(Prog)$.

Example 9. Let $Prog = \{P(f(x), f(x)) \leftarrow Q(x).\ \ Q(a) \leftarrow.\ \ Q(b) \leftarrow.\}$ and $R = \{a \rightarrow b\}$. $P(f(a), f(a)) \notin Mod^R_{NC}(Prog)$, hence $Mod^R_{NC}(Prog) \neq Mod(Prog)$.

Theorem 4. *Let $Prog$ be a normalized CS-program and R be a left-linear rewrite system. If all critical pairs are convergent by SNC derivations, $Mod^R_{NC}(Prog)$ is closed under innermost rewriting by R, i.e.*

$$(A \in Mod^R_{NC}(Prog) \wedge A \rightarrow^{in,*}_R A') \implies A' \in Mod^R_{NC}(Prog)$$

Proof. Let $A \in Mod^R_{NC}(Prog)$ s.t. $A \rightarrow^{in}_{l \rightarrow r} A'$. Then $A|_i = C[\sigma(l)]$ for some $i \in \mathbb{N}$, σ is irreducible, and $A' = A[i \leftarrow C[\sigma(r)]$.

Since $A \in Mod^R_{NC}(Prog)$, $A \rightsquigarrow^* \emptyset$ by a NC derivation. Since $Prog$ is normalized, resolution consumes symbols in C one by one, thus $G''_0{=}A \rightsquigarrow^* G''_k \rightsquigarrow^* \emptyset$ by a NC derivation, and there exists an atom $A'' = P(t_1, \ldots, t_n)$ in G''_k and j s.t. $t_j = \sigma(l)$ and the top symbol of t_j is consumed (or t_j disappears) during the step $G''_k \rightsquigarrow G''_{k+1}$.

Since t_j is reducible by R and $A \in Mod^R_{NC}(Prog)$, $t_j = \sigma(l)$ admits only one antecedent in A. Then $A' \rightsquigarrow^* G''_k[A'' \leftarrow P(t_1, \ldots, \sigma(r), \ldots, t_n)]$ by a NC derivation (I).

Consider new variables $x_1, \ldots, x_n$ s.t. $\{x_1, \ldots, x_n\} \cap Var(l) = \emptyset$, and let us define the substitution σ' by $\forall i$, $\sigma'(x_i) = t_i$ and $\forall x \in Var(l)$, $\sigma'(x) = \sigma(x)$.
Then $\sigma'(P(x_1, \ldots, x_{j-1}, l, x_{j+1}, \ldots, x_n)) = A''$.

From $G''_k \rightsquigarrow^* \emptyset$ we can extract the sub-derivation $G_k = A'' \rightsquigarrow_{\gamma_k} G_{k+1} \rightsquigarrow_{\gamma_{k+1}} G_{k+2} \rightsquigarrow^* \emptyset$, which is NC. From Lemma 5, there exist a positive integer $u > k$,

a NC derivation $G'_k = P(x_1, \ldots, l, \ldots, x_n) \leadsto^*_\theta G'_u$, and a substitution α s.t. $\alpha G'_u = G_u,\ \alpha.\theta = \gamma_{u-1}.\ldots.\gamma_k.\sigma',\ G'_u$ is flat, and for all $i,\ k < i < u$ implies G'_i is not flat. In other words, there is a critical pair, which is assumed to be convergent by a SNC derivation. Therefore $\theta(G'_k[l \leftarrow r]) \rightarrow^* G'_u$ by a SNC derivation.

Let us write $\gamma = \gamma_{u-1}.\ldots.\gamma_k$. If there exist a clause $H \leftarrow B$ used in this derivation, and $x \in Var^{mult}(H)$ s.t. $\alpha.\theta(x)$ is reducible, then there exist i and p s.t. $\alpha.\theta(x) = \gamma.\sigma'(x) = \gamma(t_i|_p)$ (because σ is irreducible). Note that γ is a unifier, then $\gamma x = \gamma(t_i|_p)$. Therefore $\gamma x = \gamma(t_i|_p) = \gamma.\sigma'(x) = \alpha.\theta(x)$, which is reducible. This is impossible because $x \in Var^{mult}(H)$ and $G_k \leadsto^*_\gamma G_u$ is a NC derivation.

Consequently, from Lemma 4, $\alpha.\theta(G'_k[l \leftarrow r]) \rightarrow^* \alpha(G'_u) = G_u \leadsto^* \emptyset$ by a NC derivation. Note that $\alpha.\theta(G'_k[l \leftarrow r]) = \gamma.\sigma'(P(x_1, \ldots, r, \ldots, x_n)) = \gamma(P(t_1, \ldots, \sigma(r), \ldots, t_n))$. Then $\gamma(P(t_1, \ldots, \sigma(r), \ldots, t_n)) \leadsto^* \emptyset$ by a NC derivation. From Lemma 5 we get:
$P(t_1, \ldots, \sigma(r), \ldots, t_n) \leadsto^* \emptyset$ by a NC derivation. Considering Derivation (I) again, we get $A' \leadsto^* G''_k[A'' \leftarrow P(t_1, \ldots, \sigma(r), \ldots, t_n)] \leadsto^* \emptyset$ by a NC derivation. In other words, $A' \in Mod^R_{NC}(Prog)$.

By trivial induction, the proof can be extended to the case of several rewrite steps.

In the following result, we consider an initial non-copying CS-program $Prog$, and a possibly copying program $Prog'$ composed of the CS-clauses added by the completion process. The normalization function norm makes critical pairs convergent by SNC derivations, provided the simplification step is achieved only if it is SNC.

Theorem 5. *Let R be a left-linear rewrite system and $Prog'' = Prog \cup Prog'$ be a normalized CS-program s.t. $Prog$ is non-copying and all critical pairs of $Prog''$ are convergent by SNC derivations. If $A \in Mod(Prog)$ and $A \rightarrow^*_R A'$ with an innermost strategy, then $A' \in Mod(Prog'')$.*

Proof. Since $Prog$ is non-copying, $Mod(Prog) = Mod^R_{NC}(Prog)$. Then $A \in Mod^R_{NC}(Prog)$, and since $Prog \subseteq Prog''$ we have $A \in Mod^R_{NC}(Prog'')$. From Theorem 4, $A' \in Mod^R_{NC}(\ Prog'')$, and since $Mod^R_{NC}(Prog'') \subseteq Mod(Prog'')$, we get $A' \in Mod(Prog'')$.

Example 10.
Let us focus on the critical pair given in Example 7 Item 2 i.e. $P(g(h(x), h(x))) \leftarrow Q_2(x)$. Adding the clause $Q_1(h(x)) \leftarrow Q_2(x)$ makes the clause convergent in $Prog''$ (in Example 7), but not convergent by a SNC derivation. Indeed (just here, we add primes to avoid conflict of variables) $P(g(h(x'), h(x'))) \leadsto_{[x/h(x')]} Q_1(h(x')) \leadsto_{[x/x']} Q_2(x')$. But the step $P(g(h(x'), h(x'))) \leadsto_{[x/h(x')]} Q_1(h(x'))$ is not SNC. Consequently, one has to normalize $P(g(h(x), h(x))) \leftarrow Q_2(x)$ in an SNC way. For instance, $P(g(h(x), h(x))) \leftarrow Q_2(x)$ can be normalized into the following clauses: $P(g(x, y)) \leftarrow Q_3(x, y)$. $Q_3(h(x), h(x)) \leftarrow Q_2(x)$.
After adding these clauses, new critical pairs are detected, and the clauses $Q_3(b, h(x)) \leftarrow Q_2(x)$. $Q_3(h(x), b) \leftarrow Q_2(x)$. $Q_3(b, b) \leftarrow$. will be added.

So, the final CS-program is $Prog'' = Prog \cup$
$\{P(g(x,x)) \leftarrow Q_1(x).\ Q_3(b,b) \leftarrow.\ P(g(x,y)) \leftarrow Q_3(x,y).\ Q_3(h(x),h(x)) \leftarrow Q_2(x).\ Q_3(b,h(x)) \leftarrow Q_2(x).\ Q_3(h(x),b) \leftarrow Q_2(x).\ \}$.
Thus $P(g(b,h(a))) \in Mod(Prog'')$.

One can apply this approach to a well-known problem: the Post Correspondence Problem.

Example 11. Consider the instance of the Post Correspondence Problem (PCP) composed of the pairs (ab, aa) and (bba, bb). To encode it by tree languages, we see a and b as unary symbols, and introduce a constant 0.
Let $R = \{Test(x) \to g(x,x),\ g(0,0) \to True,\ g(a(b(x)),a(a(y))) \to g(x,y),\ g(b(b(a(x))),b(b(y))) \to g(x,y)\}$, and let $I = \{Test(t) \mid t \in T_{\{a,b,0\}}, t \neq 0\}$ be the initial language generated by P_0 in $Prog = \{P_0(Test(z)) \leftarrow P_1(z).\ P_1(a(z)) \leftarrow P_2(z).\ P_1(b(z)) \leftarrow P_2(z).\ \ P_2(a(z)) \leftarrow P_2(z).\ P_2(b(z)) \leftarrow P_2(z).\ P_2(0) \leftarrow\}$.

Thus, this instance of PCP has at least one solution iff $True$ is reachable by R from I. Note that R is not right-linear. However, each descendant is innermost, and from Theorem 5 it is recognized by the CS-program obtained by completion: $\mathsf{comp}_R(Prog) = Prog \cup$

$$\left\{\begin{array}{l} P_0(g(x,x)) \leftarrow P_1(x). \quad P_0(g(x,y)) \leftarrow P_4(x,y).\ P_4(x,a(x)) \leftarrow P_2(x). \\ P_0(g(x,y)) \leftarrow P_5(x,y).\ P_5(x,b(x)) \leftarrow P_2(x). \quad P_0(g(x,y)) \leftarrow P_6(x,y). \\ P_6(x,b(y)) \leftarrow P_7(x,y).\ P_7(x,a(x)) \leftarrow P_2(x). \end{array}\right\}$$

Note that $P_0(True) \notin Mod(\mathsf{comp}_R(Prog))$, which proves that this instance of PCP has no solution.

3 Getting rid of copying clauses

In this section, we propose a process (see Definition 8) that transforms a copying CS-clause into a set of non-copying ones. Forcing termination of this process may lead to an over-approximation. In that way, even if the TRS is not right-linear and consequently copying clauses may be generated during the completion process, we can get rid of them as soon as they appear. Thus, the final CS-program is non-copying, and Theorem 2 applies. Therefore, an over-approximation of the set of all descendants can be computed.

For instance, let $Prog = \{P(f(x,x)) \leftarrow Q(x).\ Q(s(x)) \leftarrow Q(x).\ Q(a) \leftarrow\}$. Note that the language generated by P is $\{f(s^n(a),s^n(a)) \mid n \in \mathbb{N}\}$. We introduce a new binary predicate symbol Q^2 that generates the language $\{(t,t) \mid Q(t) \in Mod(Prog)\}$, and we transform the copying clause $P(f(x,x)) \leftarrow Q(x)$ into a non-copying one as follows: $P(f(x,y)) \leftarrow Q^2(x,y)$. Now Q^2 can be defined by the clauses $Q^2(s(x),s(x)) \leftarrow Q(x)$ and $Q^2(a,a) \leftarrow$. Unfortunately $Q^2(s(x),s(x)) \leftarrow Q(x)$ is copying. Then using the same idea again, we transform it into the non-copying clause $Q^2(s(x),s(y)) \leftarrow Q^2(x,y)$. The body of this clause uses Q^2, which is already defined. Thus the process terminates with $Prog' = \{P(f(x,y)) \leftarrow Q^2(x,y).\ Q^2(s(x),s(y)) \leftarrow Q^2(x,y).\ Q^2(a,a) \leftarrow.\ Q(s(x)) \leftarrow Q(x).\ Q(a) \leftarrow\}$.

Note that $Prog'$ is non-copying and generates the same language as $Prog$. The clauses that define Q are useless in $Prog'$, but in general it is necessary to keep them.

Now, let us formally define the *uncopying* process.

Definition 8. *Let Prog be a CS-program containing copying clauses. Thus,* uncopying($Prog$) *is a non-copying CS-program obtained from Prog as described below:*

Input: *Prog that may contain copying clauses*
Output: uncopying($Prog$) *that does not contain copying clauses anymore*

1: ***while*** *There exists a copying clause* $P(\overrightarrow{t}) \leftarrow Q_1(\overrightarrow{u_1}), \ldots, Q_n(\overrightarrow{u_n})$ *in Prog* ***do***
2: *Let* $Var(\overrightarrow{t}) = \{x_1, \ldots, x_k\}$ *be the set of variables occurring in* $\overrightarrow{t}$
3: *Let* $m_1, \ldots, m_k \in \mathbb{N}$ *be integers such that* x_i *occurs exactly* m_i *times in* $\overrightarrow{t}$
// Create $\overrightarrow{v_j}$ *which will replace* $\overrightarrow{u_j}$ *in the uncopied clause*
4: ***for all*** $j \in \{1, ..., n\}$ ***do***
5: *let* I_j *the set of variables indices of* $\overrightarrow{u_j}$ *occurring in* $\overrightarrow{t}$
6: $max_j = Max\left(\bigcup_{i \in I_j} \{m_i\}\right)$
7: *Given an integer* l, $\overrightarrow{u_j}^l$ *denotes a renaming of variables occurring in* $\overrightarrow{u_j}$ *such that* $\overrightarrow{u_j}^l = \langle x_1^l, \ldots, x_m^l \rangle$ *with* $\overrightarrow{u_j} = \langle x_1, \ldots, x_m \rangle$
8: $\overrightarrow{v_j} = \overrightarrow{u_j}^1 \ldots \overrightarrow{u_j^{max_j}}$, *i.e.* $\overrightarrow{v_j}$ *is the concatenation of* $\overrightarrow{u_j}^1, \ldots, \overrightarrow{u_j}^{max_j}$
9: ***end for***
// Create the terms $\overrightarrow{t'}$ *that will replace* $\overrightarrow{t}$ *in the uncopied clause*
10: *Let* $\overrightarrow{t'}$ *obtained from* $\overrightarrow{t}$ *by replacing for each* $j \in \{1, \ldots, k\}$, *the different occurrences of* x_j *by* $x_j^1, \ldots, x_j^{m_j}$
11: $UncopiedClause := P(\overrightarrow{t'}) \leftarrow Q_1^{max_1}(\overrightarrow{v_1}), \ldots, Q_n^{max_n}(\overrightarrow{v_n})$
12: $Prog := (Prog \setminus \{P(\overrightarrow{t}) \leftarrow Q_1(\overrightarrow{u_1}), \ldots, Q_n(\overrightarrow{u_n})\}) \cup \{UncopiedClause\}$
// Add the definitions of new predicates
13: ***for all*** $Q_i^{max_i}, max_i \neq 1$ ***do***
14: ***if*** $Q_i^{max_i}$ *not defined* ***then***
15: ***for all*** $\{ Q_i(\overrightarrow{t_j}) \leftarrow B_j\}$ *in Prog* ***do***
16: $Prog := Prog \cup \{Q_i^{max_i}(\overrightarrow{t_j} \ldots \overrightarrow{t_j}) \leftarrow B_j\}$
17: ***end for***
18: ***end if***
19: ***end for***
// Introduction of new definitions may generate new clauses to uncopy
20: ***end while***
21: **return** *Prog*

Remark 3. If Q_i has p arguments, then $Q_i^{max_i}$ has $max_i \times p$ arguments, and $L(Q_i^{max_i}) = \left\{ \underbrace{\overrightarrow{t} \ldots \overrightarrow{t}}_{max_i \text{ times}} \mid \overrightarrow{t} \in L(Q_i) \right\}$. Then $L(Q_i^1) = L(Q_i)$ and[8] $L((Q_i^x)^y) = L(Q_i^{x \times y})$. Thus we will confuse Q_i^1 with Q_i, and $(Q_i^x)^y$ with $Q_i^{x \times y}$.

Now, we give some examples of completion (Definition 4) supplied with uncopying.

Example 12.
Let $R = \{f(x) \to g(x,x),\ a \to b\}$, $Prog_0 = \{P(f(x)) \leftarrow Q_1(x).\ Q_1(a) \leftarrow\}$. $Prog_0$ is a normalized non-copying CS-Program and R is a non-right-linear rewrite system. There are 2 critical pairs, $P(g(x_1,x_1)) \leftarrow Q_1(x_1).$ and $Q_1(b) \leftarrow$. To make the critical pairs convergent, we add them into the program and we get

$Prog_1 = Prog_0 \cup \{P(g(x_1,x_1)) \leftarrow Q_1(x_1).\ Q_1(b) \leftarrow\}$

$Prog_1$ contains the copying clause $P(g(x_1,x_1)) \leftarrow Q_1(x_1)$. Applying Definition 8, at line 2, we get $n = 1$, $m_1 = 2$. So, at line 11, $UnfoldClause = P(g(x_1^1, x_1^2)) \leftarrow Q_1^2(x_1^1, x_1^2)$. From $Q_1(a) \leftarrow$ and $Q_1(b) \leftarrow$ we get respectively $Q_1^2(a,a) \leftarrow$ and $Q_1^2(b,b) \leftarrow$. Finally $\mathsf{uncopying}(Prog_1) = Prog_0 \cup \{Q_1(b) \leftarrow .\ P(g(x_1^1, x_1^2)) \leftarrow Q_1^2(x_1^1, x_1^2).\ Q_1^2(a,a) \leftarrow .\ Q_1^2(b,b) \leftarrow\}$. So, $\mathsf{uncopying}(Prog_1)$ is a normalized non-copying CS-Program.

Let $Prog_2 = \mathsf{uncopying}(Prog_1)$. Now, there are 2 non-convergent critical pairs, $Q_1^2(a,b) \leftarrow$ and $Q_1^2(b,a) \leftarrow$. If we add them to $Prog_2$, we get a normalized non-copying CS-Program, all critical pairs are convergent. Applying Theorem 2, $Mod(Prog_2)$ is closed by rewriting.

Remark 4. If at least one $Q_i^{max_i}$ is not defined and there is a clause $Q_i(\overrightarrow{t_j}) \leftarrow B_j$ in $Prog$ such that $\overrightarrow{t_j}$ is not ground, then the algorithm will generate new copying clauses.

Example 13. Let $Prog = \{P(c(x,x)) \leftarrow P(x).(1)\ P(a) \leftarrow .(2)\}$. $Prog$ is a normalized, copying CS-Program. Clause (1) is copying, we apply uncopying and add $\{P(c(x,x')) \leftarrow P^2(x,x').(3)\ P^2(a,a) \leftarrow .(4)\ P^2(c(x,x'),c(x,x')) \leftarrow P^2(x,x').(5)\}$ to $Prog$. Clause (5) is copying. Thus, the same process is performed and the clauses $\{P^2(c(x_1,x_1'),c(x_2,x_2')) \leftarrow P^4(x_1,x_1',x_2,x_2').(6)\ P^4(a,a,a,a) \leftarrow .(7)$ $P^4(c(x_1,x_1'),c(x_2,x_2'),c(x_1,x_1'),c(x_2,x_2')) \leftarrow P^4(x_1,x_1',x_2,x_2').(8)\}$ are added to $Prog$. Unfortunately Clause (8) is copying. The process does not terminate, consequently we will never get a program without copying clauses.

To force termination while getting rid of all copying clauses, we fix a positive integer $UncopyingLimit$. If we need to generate a predicate Q^x where $x > UncopyingLimit$ we cut Q^x into $Q^{x_1}, \ldots, Q^{x_n}$ with $\sum\limits_{i \in [1,n]} x_i = x$.

[8] If the loop while is run several times, predicate symbols of the form $(Q_i^x)^y$ may appear.

Example 14. Consider Example 13 again, and let $UncopyingLimit = 4$. Clause (8) is copying. Applying the process would generate the clause
$P^4(c(x_1, x'_1), c(x_2, x'_2), c(x_3, x'_3), c(x_4, x'_4)) \leftarrow P^8(x_1, x'_1, x_2, x'_2, x_3, x'_3, x_4, x'_4)$
However $UncopyingLimit$ is exceeded. So, we cut P^8 and obtain
$P^4(c(x_1, x'_1), c(x_2, x'_2), c(x_3, x'_3), c(x_4, x'_4))$
$$\leftarrow P^4(x_1, x'_1, x_2, x'_2), P^2(x_3, x'_3), P^2(x_4, x'_4). \quad (9)$$
Predicates P^4 and P^2 have been defined previously in $Prog$, so we do not need to add more clauses to do it.

Finally, the CS-program $\mathsf{uncopying}(Prog)$ includes the uncopying clauses (2), (3), (4), (6), (7) and (9). Recall that $L(P^8)$ is supposed to be defined so that $L(P^8) = \{\underbrace{\overrightarrow{t} \dots \overrightarrow{t}}_{8 \text{ times}} \mid \overrightarrow{t} \in L(P)\}$. Then replacing $P^8(x_1, x'_1, x_2, x'_2, x_3, x'_3, x_4, x'_4)$ by $P^4(x_1, x'_1, x_2, x'_2), P^2(x_3, x'_3), P^2(x_4, x'_4)$ in the clause-body generates the set $\{\underbrace{\overrightarrow{t} \dots \overrightarrow{t}}_{4 \text{ times}}.\overrightarrow{t'}.\overrightarrow{t'}.\overrightarrow{t''}.\overrightarrow{t''} \mid \overrightarrow{t}, \overrightarrow{t'}, \overrightarrow{t''} \in L(P)\} \subset L(P^8)$, which leads to an over-approximation. For example $P^4(c(a,a), c(a,a), c(c(a,a), c(a,a)), c(a,a))$ is in $Mod(\mathsf{uncopying}(Prog))$ but not in $Mod(Prog)$.

4 Further Work

In this paper, we have shown that the non-regular approximation technique by means of CS-programs can also deal with left-linear non-right-linear rewrite systems. Naturally, the question that still arises is: can this technique be extended to non-left-linear rewrite systems. From a theoretical point of view, applying a non-left-linear rewrite rule amounts to compute the intersection of several languages of sub-terms, i.e. the intersection of CS-programs. Unfortunately, it is known that the class of synchronized tree languages (i.e. the languages recognized by CS-programs) is not closed under intersection. In other words, except for particular cases, such intersection cannot be computed in an exact way. However, it could be over-approximated by a CS-program. We are studying this possibility.

References

1. Y. Boichut, B. Boyer, Th. Genet, and A. Legay. Equational Abstraction Refinement for Certified Tree Regular Model Checking. In *ICFEM*, volume 7635 of *LNCS*, pages 299–315. Springer, 2012.
2. Y. Boichut, J. Chabin, and P. Réty. Over-approximating descendants by synchronized tree languages. In *RTA*, volume 21 of *LIPIcs*, pages 128–142, 2013.
3. Y. Boichut, J. Chabin, and P. Réty. Erratum of over-approximating descendants by synchronized tree languages. Technical report, LIFO, Université d'Orléans, http://www.univ-orleans.fr/lifo/Members/rety/publications.html#erratum, 2015.
4. Y. Boichut, R. Courbis, P.-C. Héam, and O. Kouchnarenko. Finer is Better: Abstraction Refinement for Rewriting Approximations. In *RTA*, volume 5117 of *LNCS*, pages 48–62. Springer, 2008.

5. Y. Boichut and P.-C. Héam. A Theoretical Limit for Safety Verification Techniques with Regular Fix-point Computations. *Information Processing Letters*, 108(1):1–2, 2008.
6. A. Bouajjani, P. Habermehl, A. Rogalewicz, and T. Vojnar. Abstract Regular (Tree) Model Checking. *Journal on Software Tools for Technology Transfer*, 14(2):167–191, 2012.
7. Th. Genet and F. Klay. Rewriting for Cryptographic Protocol Verification. In *CADE*, volume 1831 of *LNAI*, pages 271–290. Springer-Verlag, 2000.
8. V. Gouranton, P. Réty, and H. Seidl. Synchronized Tree Languages Revisited and New Applications. In *FoSSaCS*, volume 2030 of *LNCS*, pages 214–229. Springer, 2001.
9. J. Kochems and C.-H. Luke Ong. Improved Functional Flow and Reachability Analyses Using Indexed Linear Tree Grammars. In *RTA*, volume 10 of *LIPIcs*, pages 187–202, 2011.
10. S. Limet and P. Réty. E-Unification by Means of Tree Tuple Synchronized Grammars. *Discrete Mathematics and Theoritical Computer Science*, 1(1):69–98, 1997.
11. S. Limet and G. Salzer. Proving Properties of Term Rewrite Systems via Logic Programs. In *RTA*, volume 3091 of *LNCS*, pages 170–184. Springer, 2004.
12. Sébastien Limet and Gernot Salzer. Tree Tuple Languages from the Logic Programming Point of View. *Journal of Automated Reasoning*, 37(4):323–349, 2006.