Over-approximating Descendants by Synchronized Tree Languages

Yohan Boichut[1], Jacques Chabin[1], and Pierre Réty[1]

1 **LIFO - Université d'Orléans, B.P. 6759, 45067 Orléans cedex 2, France**
`{yohan.boichut, jacques.chabin, pierre.rety}@univ-orleans.fr`

Abstract

Over-approximating the descendants (successors) of a initial set of terms by a rewrite system is used in verification. The success of such verification methods depends on the quality of the approximation. To get better approximations, we are going to use non-regular languages. We present a procedure that always terminates and that computes an over-approximation of descendants, using synchronized tree-(tuple) languages expressed by logic programs.

1998 ACM Subject Classification F.4.2

Keywords and phrases Rewriting systems, non-regular approximations, logic programming, tree languages, descendants

Digital Object Identifier 10.4230/LIPIcs.xxx.yyy.p

1 Introduction

Given an initial set of terms I, computing the descendants (successors) of I by a rewrite system R is used in the verification domain, for example to check cryptographic protocols or Java programs [2, 8, 10, 9]. Let $R^*(I)$ denote the set of descendants of I, and consider a set Bad of *undesirable* terms. Thus, if a term of Bad is reached from I, i.e. $R^*(I) \cap Bad \neq \emptyset$, it means that the protocol or the program is flawed. In general, it is not possible to compute $R^*(I)$ exactly. Instead, we compute an over-approximation App of $R^*(I)$ (i.e. $App \supseteq R^*(I)$), and check that $App \cap Bad = \emptyset$, which ensures that the protocol or the program is correct.

Most often, I, App and Bad have been considered as regular tree languages, recognized by finite tree automata. In the general case, $R^*(I)$ is not regular, even if I is. Moreover, the expressiveness of regular languages is poor, and the over-approximation App may not be precise enough, and we may have $App \cap Bad \neq \emptyset$ whereas $R^*(I) \cap Bad = \emptyset$. In other words, the protocol is correct, but we cannot prove it. Some work has proposed CEGAR-techniques (Counter-Example Guided Approximation Refinement) in order to conclude as often as possible [2, 4, 6]. However, in some cases, no regular over-approximation works, whatever the quality of the approximation is [5].

To overcome this theoretical limit, we want to use more expressive languages to express the over-approximation, i.e. non-regular ones. However, to be able to check that $App \cap Bad = \emptyset$, we need a class of languages closed under intersection and whose emptiness is decidable. Actually, since we still assume that Bad is regular, closure under intersection with a regular language is enough. The class of context-free tree languages has these properties, and an over-approximation of descendants using context-free tree languages has been proposed in [14]. This class of languages is quite interesting, however it cannot express relations (or countings) in terms between independent branches, except if there are only unary symbols and constants. For example, let $R = \{f(x) \to c(x, x)\}$ and the infinite set $I = \{f(t)\}$ where

Conference title on which this volume is based on.
Editors: Billy Editor, Bill Editors; pp. 1–15
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

t denotes any term composed with the binary symbol g and constant b. Then $R^*(I) = I \cup \{c(t,t)\}$, which is not a context-free language [1, 13].

We want to use another class of languages that has the needed properties, and that can express relations between independent branches: the synchronized tree-(tuple) languages [15, 12], which were finally expressed thanks to logic programs (Horn clauses) [16, 17]. This class has the same properties as context-free tree languages: closure under union, closure under intersection with a regular language (in quadratic time), decidability of membership and emptiness (in linear time). Both include regular languages, however they are different. The example given above is not context-free, but synchronized. The language $\{s^n(p^n(a))\}$ (where s^n means that s occurs n times vertically) is context-free, but it is not synchronized. $\{c(s^n(a), p^n(a))\}$ belongs to both classes (note that s and p are unary).

In this paper, we propose a procedure that always terminates and that computes an over-approximation of the descendants obtained by a left-linear rewrite system, using synchronized tree-(tuple) languages expressed by logic programs. Note that the left-linearity of rewrite systems (or transducers) is a usual restriction, see [2, 6, 8, 10, 9]. Nevertheless, such rewrite systems are still Turing complete [7].

The paper is organized as follows: classical notations and notions manipulated throughout the paper are introduced in Section 2. Our main contribution, i.e. computing approximations using synchronized languages, is explained in Section 3. Finally, in Section 4 our technique is applied on two pertinent examples: an example illustrating a non-regular approximation of a non-regular set of terms, and another one that cannot be handled by any regular approximation.

2 Preliminaries

Consider a *finite ranked alphabet* Σ and a set of variables *Var*. Each symbol $f \in \Sigma$ has a unique arity, denoted by $ar(f)$. The notions of *first-order term, position, substitution,* are defined as usual. Given σ and σ' two substitutions, $\sigma \circ \sigma'$ 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 $t[t']_p$ is obtained from t by replacing the subterm at position p by t'. $PosVar(t) = \{p \in Pos(t) \mid t(p) \in Var\}$, $PosNonVar(t) = \{p \in Pos(t) \mid t(p) \notin Var\}$. Note that if $p \in PosNonVar(t)$, $t|_p = f(t_1, \dots, t_n)$, and $i \in \{1, \dots, n\}$, then $p.i$ is the position of t_i in t. For $p, p' \in Pos(t)$, $p < p'$ means that p occurs in t strictly above p'. Let t, t' be terms, t is *more general than* t' (denoted $t \leq t'$) if there exists a substitution ρ s.t. $\rho(t) = t'$. Let σ, σ' be substitutions, σ is *more general than* σ' (denoted $\sigma \leq \sigma'$) if there exists a substitution ρ s.t. $\rho \circ \sigma = \sigma'$.

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.

In the following, we consider the framework of *pure logic programming*, and the class of synchronized tree-tuple languages defined by CS-clauses [16, 17]. 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. B is *linear* if each variable occurring in B (possibly at sub-term position) occurs only once in B. Note that the empty sequence of atoms (denoted by $\emptyset$) is flat and linear.
A *CS-clause*[1] 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.

Given a predicate symbol P of arity n, the tree-(tuple) language generated by P is $L(P) = \{\vec{t} \in (T_\Sigma)^n \mid P(\vec{t}) \in Mod(Prog)\}$, where T_Σ is the set of ground terms over the signature Σ and $Mod(Prog)$ is the least Herbrand model of $Prog$. $L(P)$ is called *Synchronized language.*

The following definition describes the different kinds of CS-clauses that can occur.

▶ **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. A CS-program is *normalized* if all its clauses are normalized.
- *preserving* if $Var(P(t_1, \ldots, t_n)) \subseteq Var(B)$. A CS-program is *preserving* if all its clauses are preserving.
- *synchronizing* if B is composed of only one atom.

▶ **Example 3.** The CS-clause $P(x, y, z) \leftarrow G(x, y, z)$ is empty, normalized, and preserving (x, y, z are variables). The CS-clause $P(f(x), y, g(x, z)) \leftarrow G(x, y)$ is normalized and non-preserving. Both clauses are synchronizing.

Given a CS-program, we focus on two kinds of derivations: a classical one based on unification and a rewriting one based on matching and a rewriting process.

▶ **Definition 4.** Given a logic program $Prog$ and a sequence of atoms G,

- G derives into G' by a *resolution* step if there exist a clause[2] $H \leftarrow B$ 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 $G \rightarrow_\sigma G'$.

▶ **Example 5.** 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 $\sigma_2 = [x_1/x, x_2/y]$.

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 $G_1 \rightsquigarrow^*_{\sigma_2 \circ \sigma_1} G_3$;

[1] In former papers, synchronized tree-tuple languages were defined thanks to sorts of grammars, called constraint systems. Thus "CS" stands for Constraint System.
[2] We assume that the clause and G have distinct variables.

- 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, $G_1 \leadsto^*_{Prog} G_2$ (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 6.** *Let A be a ground atom. $A \in Mod(Prog)$ iff $A \leadsto^*_{Prog} \emptyset$.*

▶ **Example 7.** Let $A = P(f(g(a)), g(a), c)$ and $A' = P'(f(g(a)), h(c))$ be two ground atoms. Let $Prog$ be the CS-program defined by:
$Prog = \{P(f(g(x)), y, c) \leftarrow P_1(x), P_2(y).\ P_1(a) \leftarrow .\ \ P_2(g(x)) \leftarrow P_1(x).\ P'(f(x), u(z)) \leftarrow .\}$
Thus, $A \in Mod(Prog)$ and $A' \notin Mod(Prog)$.

Note that for any atom A, if $A \to B$ then $A \leadsto B$. If in addition $Prog$ is preserving, then $Var(A) \subseteq Var(B)$. On the other hand, $A \leadsto_\sigma B$ implies $\sigma(A) \to B$. Consequently, if A is ground, $A \leadsto B$ implies $A \to B$.

The following lemma focuses on a preserving property of the relation $\leadsto$.

▶ **Lemma 8.** *Let $Prog$ be a CS-program, and G be a sequence of atoms. Let $|G|_\Sigma$ denote the number of occurrences of function-symbols in G. If G is linear and $G \leadsto^* G'$, then G' is also linear and $|G'|_\Sigma \leq |G|_\Sigma$.*
Consequently, if G is flat and linear, then G' is also flat and linear.

Proof. Let $G = A^1 \dots A^k$ be a linear sequence of atoms and suppose that $G \leadsto_\sigma G'$.
Then there exist an atom $A^i(s_1, \dots, s_n)$ of G and a CS-clause $A^i(t_1, \dots, t_n) \leftarrow B$ in $Prog$ such that $G' = \sigma(G)[\sigma(A^i) \leftarrow \sigma(B)]$. As G is linear and σ is the most general unifier between $A^i(s_1, \dots, s_n)$ and $A^i(t_1, \dots, t_n)$, σ does not instantiate variables from $A^1, \dots, A^{i-1}, A^{i+1}, \dots A^k$. So $G' = A^1, \dots, A^{i-1}, \sigma(B), A^{i+1}, \dots A^k$.

G' is not linear only if $\sigma(B)$ is not linear. As B is linear, $\sigma(B)$ is not linear would require that two distinct variables x_{j_1}, x_{j_2} from B are instantiated by two terms containing a same variable $y \in Var(\sigma(x_{j_1}) \cap Var(\sigma(x_{j_1}))$. Since σ is the most general unifier, x_{j_1}, x_{j_2} are also in $Var(A^i(t_1, \dots, t_n))$ (σ does not instantiate extra variables). Then y occurs at least twice in $A^i(s_1, \dots, s_n)$ (the atom of goal G), which is impossible since G is linear. Consequently G' is linear.

By contradiction: to obtain $|G'|_\Sigma > |G|_\Sigma$, we must have in $\sigma(B)$ a duplication of a non-variable subterm of $\sigma((A^i(s_1, \dots, s_n))$ (because B is flat), which is not possible because B and $A^i(s_1, \dots, s_n)$ are linear and σ is the most general unifier.

The result trivially extends to the case of several steps $G \leadsto^* G'$. ◀

▶ **Example 9.** Let $Prog = \{P(g(x), f(x)) \leftarrow P_1(x)\}$ and $G = P(g(f(y)), z)$. Then $G \leadsto G'$ with $G' = P_1(f(y))$, and G' is linear. Moreover, $|G'|_\Sigma \leq |G'|_\Sigma$ with $\Sigma = \{f^{\backslash 1}, a^{\backslash 0}\}$.

3 Computing Descendants

Given a CS-program $Prog$ and a left-linear rewrite system R, we propose a technique allowing us to compute a CS-program $Prog'$ such that $R^*(Mod(Prog)) \subseteq Mod(Prog')$. First of all, a notion of critical pairs is introduced in Section 3.1. Roughly speaking, this notion makes the detection of uncovered rewriting steps possible. Critical pair detection is at the heart of the technique. Thus, in Section 3.2 some restrictions are underlined on CS-programs in order to make the number of critical pairs finite. Moreover, when a CS-program does not fit these restrictions, we have proposed a technique in order to transform such a CS-program into another one of the expected form (REMOVE CYCLES in Fig.1). The detected critical pairs lead to a set of CS-clauses to be added in the current CS-program. However, they may not be in the expected form i.e. normalized CS-clauses. Indeed, one of the restritions set in Section 3.2 is that the CS-program has to be normalized. So, we propose in Section 3.3 an algorithm providing normalized CS-clauses from non-normalized ones. Finally, in Section 3.4, our main contribution, i.e. the computation of an over-approximating CS-program, is fully described.

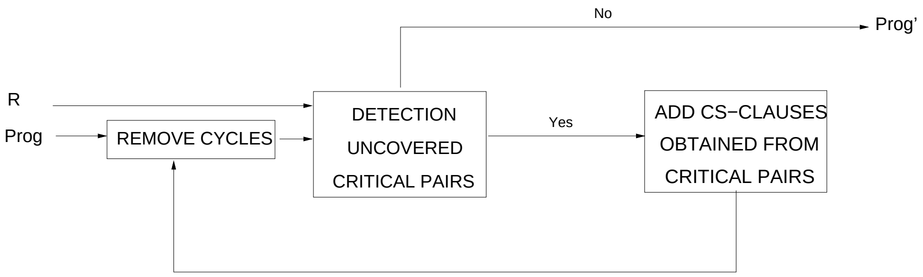

Figure 1 An overview of our contribution

3.1 Critical pairs

The notion of critical pair is at the heart of our technique. Indeed, it allows us to add CS-clauses into the current CS-program in order to cover rewriting steps. This notion is described in Definition 10.

▶ **Definition 10.** Let *Prog* be a CS-program and $l \to r$ be a left-linear rewrite rule. Let $x_1, \dots, x_n$ be distinct variables s.t. $\{x_1, \dots, x_n\} \cap Var(l) = \emptyset$. If there are P and k s.t. $P(x_1, \dots, x_{k-1}, l, x_{k+1}, \dots, x_n) \rightsquigarrow_\theta^+ G$ where resolution is applied only on non-flat atoms, G is flat, and the clause $P(t_1, \dots, t_n) \leftarrow B$ used during the first step of this derivation satisfies t_k is not a variable[3], then the clause $\theta(P(x_1, \dots, x_{k-1}, r, x_{k+1}, \dots, x_n)) \leftarrow G$ is called *critical pair*.

▶ Remark. Since l is linear, $P(x_1, \dots, x_{k-1}, l, x_{k+1}, \dots, x_n)$ is linear, and thanks to Lemma 8 G is linear, then a critical pair is a CS-clause. Moreover, if *Prog* is preserving then a critical pair is a preserving CS-clause[4].

▶ **Example 11.** Let *Prog* be the normalized and preserving CS-program defined by:

$$Prog = \{P(c(x), c(x), y) \leftarrow Q(x, y). \quad Q(a, b) \leftarrow . \quad Q(c(x), y) \leftarrow Q(x, y)\}.$$

and consider the left-linear rewrite rule: $c(c(x')) \to h(h(x'))$. Recall that for all goals G, G', the step $G \to G'$ means that $G \rightsquigarrow_\sigma G'$ where σ does not instantiate the variables of G. Thus $P(c(c(x')), y', z') \rightsquigarrow_\theta Q(c(x'), y) \to Q(x', y)$ where $\theta = [x/c(x'),\, y'/c(c(x')),\, z'/y]$. It generates the critical pair $P(h(h(x')), c(c(x')), y) \leftarrow Q(x', y)$. There are also two other critical pairs: $P(c(c(x')), h(h(x')), y) \leftarrow Q(x', y)$ and $Q(h(h(x')), y) \leftarrow Q(x', y)$.

---

3 In other words, the overlap of l on the clause head $P(t_1, \dots, t_n)$ is done at a non-variable position.

4 We have $\theta(P(x_1, \dots, x_{k-1}, l, x_{k+1}, \dots, x_n)) \to^* G$, and since *Prog* is preserving $Var(\theta(P(x_1, \dots, x_{k-1}, l, x_{k+1}, \dots, x_n))) \subseteq Var(G)$. Since $Var(r) \subseteq Var(l)$ we have $Var(\theta(P(x_1, \dots, x_{k-1}, r, x_{k+1}, \dots, x_n))) \subseteq Var(G)$.

However, some of the detected critical pairs are not so *critical* since they are already covered by the current CS-program. These critical pairs are said to be convergent.

▶ **Definition 12.** A critical pair $H \leftarrow B$ is said *convergent* if $H \rightarrow^*_{Prog} B$.

▶ **Example 13.** The three critical pairs detected in Example 11 are not convergent in $Prog$.

So, here we come to Theorem 14, i.e. the corner stone making our approach sound. Indeed, given a rewrite system R and CS-program $Prog$, if every critical pair that can be detected is convergent, then for any set of terms I such that $I \subseteq Mod(Prog)$, $Mod(Prog)$ is an over-approximation of the set of terms reachable by R from I.

▶ **Theorem 14.** *Let $Prog$ be a normalized and preserving CS-program and R be a left-linear rewrite system.*
If all critical pairs are convergent, then $Mod(Prog)$ is closed under rewriting by R, i.e. $(A \in Mod(Prog) \wedge A \rightarrow^*_R A') \implies A' \in Mod(Prog)$.

Proof. Let $A \in Mod(Prog)$ s.t. $A \rightarrow_{l \rightarrow r} A'$. Then $A|_i = C[\sigma(l)]$ for some $i \in \mathbb{N}$ and $A' = A[i \leftarrow C[\sigma(r)]$.
Since resolution is complete, $A \leadsto^* \emptyset$. Since $Prog$ is normalized and preserving, resolution consumes symbols in C one by one, thus $G_0 = A \leadsto^* G_k \leadsto^* \emptyset$ 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 during the step $G_k \leadsto G_{k+1}$. 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 $\sigma'(P(x_1, \ldots, x_{j-1}, l, x_{j+1}, \ldots, x_n)) = A''$, and according to resolution (or narrowing) properties $P(x_1, \ldots, l, \ldots, x_n) \leadsto^*_\theta \emptyset$ and $\theta \le \sigma'$.
This derivation can be decomposed into : $P(x_1, \ldots, l, \ldots, x_n) \leadsto^*_{\theta_1} G' \leadsto_{\theta_2} G \leadsto^*_{\theta_3} \emptyset$ where $\theta = \theta_3 \circ \theta_2 \circ \theta_1$, and s.t. G' is not flat and G is flat[5]. $P(x_1, \ldots, l, \ldots, x_n) \leadsto^*_{\theta_1} G' \leadsto_{\theta_2} G$ can be commuted into $P(x_1, \ldots, l, \ldots, x_n) \leadsto^*_{\gamma_1} B' \leadsto_{\gamma_2} B \leadsto^*_{\gamma_3} G$ s.t. B is flat, B' is not flat, and within $P(x_1, \ldots, l, \ldots, x_n) \leadsto^*_{\gamma_1} B' \leadsto_{\gamma_2} B$ resolution is applied only on non-flat atoms, and we have $\gamma_3 \circ \gamma_2 \circ \gamma_1 = \theta_2 \circ \theta_1$. Then $\gamma_2 \circ \gamma_1(P(x_1, \ldots, r, \ldots, x_n)) \leftarrow B$ is a critical pair. By hypothesis, it is convergent, then $\gamma_2 \circ \gamma_1(P(x_1, \ldots, r, \ldots, x_n)) \rightarrow^* B$. Note that $\gamma_3(B) \rightarrow^* G$ and recall that $\theta_3 \circ \gamma_3 \circ \gamma_2 \circ \gamma_1 = \theta_3 \circ \theta_2 \circ \theta_1 = \theta$. Then $\theta(P(x_1, \ldots, r, \ldots, x_n)) \rightarrow^* \theta_3(G) \rightarrow^* \emptyset$, and since $\theta \le \sigma'$ we get $P(t_1, \ldots, \sigma(r), \ldots, t_n) = \sigma'(P(x_1, \ldots, r, \ldots, x_n)) \rightarrow^* \emptyset$. Therefore $A' \leadsto^* G_k[A'' \leftarrow P(t_1, \ldots, \sigma(r), \ldots, t_n)] \leadsto^* \emptyset$, hence $A' \in Mod(Prog)$.
By trivial induction, the proof can be extended to the case of several rewrite steps. ◀

If $Prog$ is not normalized, Theorem 14 does not hold.

▶ **Example 15.** Let $Prog = \{P(c(f(a))) \leftarrow\}$ and $R = \{f(a) \rightarrow b\}$. All critical pairs are convergent since there is no critical pair. $P(c(f(a))) \in Mod(Prog)$ and $P(c(f(a))) \rightarrow_R P(c(b))$. However there is no resolution step issued from $P(c(b))$, then $P(c(b)) \notin Mod(Prog)$.

If $Prog$ is not preserving, Theorem 14 does not hold.

▶ **Example 16.** Let $Prog = \{P(c(x), c(x), y) \leftarrow Q(y).\ \ Q(a) \leftarrow\}$, and $R = \{f(b) \rightarrow b\}$. All critical pairs are convergent since there is no critical pair.
$P(c(f(b)), c(f(b)), a) \rightarrow_{Prog} Q(a) \rightarrow_{Prog} \emptyset$, then $P(c(f(b)), c(f(b)), a) \in Mod(Prog)$. On the other hand, $P(c(f(b)), c(f(b)), a) \rightarrow_R P(c(b), c(f(b)), a)$. However there is no resolution step issued from $P(c(b), c(f(b)), a)$, then $P(c(b), c(f(b)), a) \notin Mod(Prog)$.

[5] Since $\emptyset$ is flat, a flat goal can always be reached, i.e. in some cases $G = \emptyset$.

Unfortunately, for a given finite CS-program, there may be infinitely many critical pairs. In the following section, this problem is illustrated and some syntactical conditions on CS-program are underlined in order to avoid this critical situation.

3.2 Ensuring finitely many critical pairs

The following example illustrates a situation where the number of critical pairs is unbounded.

▶ **Example 17.** Let $\Sigma = \{f^{\backslash 2}, c^{\backslash 1}, d^{\backslash 1}, s^{\backslash 1}, a^{\backslash 0}\}$ and $f(c(x), y) \rightarrow d(y)$ be a rewrite rule, and $Prog = \{P_0(f(x,y)) \leftarrow P_1(x,y).\ P_1(x, s(y)) \leftarrow P_1(x,y).\ P_1(c(x), y) \leftarrow P_2(x,y).\ P_2(a,a) \leftarrow .\}$. Then $P_0(f(c(x), y)) \rightarrow P_1(c(x), y) \rightsquigarrow_{y/s(y)} P_1(c(x), y) \rightsquigarrow_{y/s(y)} \cdots P_1(c(x), y) \rightarrow P_2(x, y)$. Resolution is applied only on non-flat atoms and the last atom obtained by this derivation is flat. The composition of substitutions along this derivation gives $y/s^n(y)$ for some $n \in \mathbb{N}$. There are infinitely many such derivations, which generates infinitely many critical pairs of the form $P_0(d(s^n(y))) \leftarrow P_2(x, y)$.

This is annoying since the completion process presented in the following needs to compute all critical pairs. This is why we define sufficient conditions to ensure that a given finite CS-program has finitely many critical pairs.

▶ **Definition 18.** *Prog* is *empty-recursive* if there exist a predicate P and distinct variables $x_1, \dots, x_n$ s.t. $P(x_1, \dots, x_n) \rightsquigarrow^+_\sigma A_1, \dots, P(x'_1, \dots, x'_n), \dots, A_k$ where $x'_1, \dots, x'_n$ are variables and there exist i, j s.t. $x'_i = \sigma(x_i)$ and $\sigma(x_j)$ is not a variable and $x'_j \in Var(\sigma(x_j))$.

▶ **Example 19.** Let *Prog* be the CS-program defined as follows:
$Prog = \{P(x', s(y')) \leftarrow P(x', y').\ \ P(a, b) \leftarrow .\}$
From $P(x, y)$, one can obtained the following derivation: $P(x, y) \rightsquigarrow_{[x/x',\, y/s(y')]} P(x', y')$. Consequently, *Prog* is empty-recursive since $\sigma = [x/x',\, y/s(y')]$, $x' = \sigma(x)$ and y' is a variable of $\sigma(y) = s(y')$.

The following lemma shows that the non empty-recursiveness of a CS-program is sufficient to ensure the finiteness of the number of critical pairs.

▶ **Lemma 20.** *Let Prog be a normalized CS-program.*
If Prog is not empty-recursive, then the number of critical pairs is finite.

▶ Remark. Note that the CS-program of Example 17 is normalized and has infinitely many critical pairs, however it is empty-recursive because $P_1(x, y) \rightsquigarrow_{[x/x',\, y/s(y')]} P_1(x', y')$.

Proof. By contrapositive. Let us suppose there exist infinitely many critical pairs. So there exist P_1 and infinitely many derivations of the form $(i):\ P_1(x_1, \dots, x_{k-1}, l, x_{k+1}, \dots, x_n) \rightsquigarrow^*_\alpha G' \rightsquigarrow_\theta G$ (the number of steps is not bounded). As the number of predicates is finite and every predicate has a fixed arity, there exists a predicate P_2 and a derivation of the form $(ii):\ P_2(t_1, \dots, t_p) \rightsquigarrow^k_\sigma G''_1, P_2(t'_1, \dots, t'_p), G''_2$ (with $k > 0$) included in some derivation of (i), strictly before the last step, such that :

1. G''_1 and G''_2 are flat.
2. σ is not empty and there exists a variable x in $P_2(t_1, \dots, t_p)$ such that $\sigma(x) = t$ and t is not a variable and contains a variable y that occurs in $P_2(t'_1, \dots, t'_p)$. Otherwise we could not have an infinite number of σ necessary to obtain infinitely many critical pairs.

3. At least one term t'_j ($j \in \{1,\ldots,p\}$) is not a variable (only the last step of the initial derivation produces a flat goal G). As we use a CS-clause in each derivation step, we can assume that t'_j is a term among $t_1,\ldots,t_n$ and moreover that $t'_j = t_j$. This property does not necessarily hold as soon as P_2 is reached within (ii). We may have to consider further occurrences of P_2 so that each required term occurs in the required argument, which will necessarily happen because there are only finitely many permutations. So, for each variable x occurring in the non-variable terms, we have $\sigma(x) = x$.
4. From the previous item, we deduce that the variable x found in item 2 is one of the terms $t_1,\ldots,t_p$, say t_k. We can assume that y is t'_k.

If in the (ii) derivation we replace all non-variable terms by new variables, we obtain a new derivation : (iii) : $P_2(x_1,\ldots,x_p) \leadsto^k_\sigma G''_1, P_2(x'_1,\ldots,x'_p), G''_2$ and there exists i, k such that $\sigma(x_i) = x'_i$ (at least one non-variable term in the (ii) derivation), $\sigma(x_k) = t_k$, and x'_k is a variable of t_k. We conclude that Prog is empty-recursive. ◀

Deciding the empty-recursiveness of a CS-program seems to be a difficult problem (undecidable?). Nevertheless, we propose a sufficient syntactic condition to ensure that a CS-program is not empty-recursive.

▶ **Definition 21.** The clause $P(t_1,\ldots,t_n) \leftarrow A_1,\ldots,Q(\ldots),\ldots,A_m$ is *pseudo-empty over* Q if there exist i,j s.t.
- t_i is a variable,
- and t_j is not a variable,
- and $\exists x \in Var(t_j)$, $x \neq t_i \wedge \{x,t_i\} \subseteq Var(Q(\ldots))$.

Roughly speaking, when making a resolution step issued from the flat atom $P(y_1,\ldots,y_n)$, the variable y_i is not instantiated, and y_j is instantiated by something that is synchronized with y_i (in $Q(\ldots)$).

The clause $H \leftarrow B$ is *pseudo-empty* if there exists some Q s.t. $H \leftarrow B$ is pseudo-empty over Q.

The CS-clause $P(t_1,\ldots,t_n) \leftarrow A_1,\ldots,Q(x_1,\ldots,x_k),\ldots,A_m$ is *empty over* Q if for all x_i, ($\exists j,\ t_j = x_i$ or $x_i \notin Var(P(t_1,\ldots,t_n))$).

▶ **Example 22.** The CS-clause $P(x, f(x), z) \leftarrow Q(x, z)$ is both pseudo-empty (thanks to the second and the third argument of P) and empty over Q (thanks to the first and the third argument of P).

▶ **Definition 23.** Using Definition 21, let us define two relations over predicate symbols.

- $P_1 \trianglerighteq_{Prog} P_2$ if there exists in $Prog$ a clause empty over P_2 of the form $P_1(\ldots) \leftarrow A_1,\ldots,P_2(\ldots),\ldots,A_n$. The reflexive-transitive closure of $\trianglerighteq_{Prog}$ is denoted by $\trianglerighteq^*_{Prog}$.
- $P_1 >_{Prog} P_2$ if there exist in $Prog$ predicates P'_1, P'_2 s.t. $P_1 \trianglerighteq^*_{Prog} P'_1$ and $P'_2 \trianglerighteq^*_{Prog} P_2$, and a clause pseudo-empty over P'_2 of the form $P'_1(\ldots) \leftarrow A_1,\ldots,P'_2(\ldots),\ldots,A_n$. The transitive closure of $>_{Prog}$ is denoted by $>^+_{Prog}$.

$>_{Prog}$ is *cyclic* if there exists a predicate P s.t. $P >^+_{Prog} P$.

▶ **Example 24.** Let $\Sigma = \{f^{\backslash 1}, h^{\backslash 1}, a^{\backslash 0}\}$ Let $Prog$ be the following CS-program:

$$Prog = \{P(x, h(y), f(z)) \leftarrow Q(x,z), R(y).\ \ Q(x, g(y,z)) \leftarrow P(x,y,z).\ \ R(a) \leftarrow .\ \ Q(a,a) \leftarrow .\}$$

One has $P >_{Prog} Q$ and $Q >_{Prog} P$. Thus, $>_{Prog}$ is cyclic.

The lack of cycles is the key point of our technique since it ensures the finiteness of the number of critical pairs.

▶ **Lemma 25.** *If $>_{Prog}$ is not cyclic, then Prog is not empty-recursive, consequently the number of critical pairs is finite.*

Proof. By contrapositive. Let us suppose that $Prog$ is empty recursive. So there exist P and distinct variables $x_1, \dots, x_n$ s.t. $P(x_1, \dots, x_n) \leadsto^+_\sigma A_1, \dots, P(x'_1, \dots, x'_n), \dots, A_k$ where $x'_1, \dots, x'_n$ are variables and there exist i, j s.t. $x'_i = \sigma(x_i)$ and $\sigma(x_j)$ is not a variable and $x'_j \in Var(\sigma(x_j))$. We can extract from the previous derivation the following derivation which has p steps ($p \geq 1$). $P(x_1, \dots, x_n) = Q^0(x_1, \dots, x_n) \leadsto_{\alpha_1} B^1_1 \dots Q^1(x^1_1, \dots, x^1_{n_1}) \dots B^1_{k_1} \leadsto_{\alpha_2} B^1_1 \dots B^2_1 \dots Q^2(x^2_1, \dots, x^2_{n_2}) \dots B^2_{k_2} \dots B^1_{k_1} \leadsto_{\alpha_3} \dots \leadsto_{\alpha_p} B^1_1 \dots B^p_1 \dots Q^p(x^p_1, \dots, x^p_{n_p}) \dots B^p_{k_p} \dots B^1_{k_1}$ where $Q^p(x^p_1, \dots, x^p_{n_p}) = P(x'_1, \dots, x'_n)$.

For each k, $\alpha_k \circ \alpha_{k-1} \dots \circ \alpha_1(x_i)$ is a variable of $Q^k(x^k_1, \dots, x^k_{n_k})$ and $\alpha_k \circ \alpha_{k-1} \dots \circ \alpha_1(x_j)$ is either a variable of $Q^k(x^k_1, \dots, x^k_{n_k})$ or a non-variable term containing a variable of $Q^k(x^k_1, \dots, x^k_{n_k})$.

Each derivation step issued from Q^k uses either a clause pseudo-empty over Q^{k+1} and we deduce $Q^k >_{Prog} Q^{k+1}$, or an empty clause over Q^{k+1} and we deduce $Q^k \trianglerighteq_{Prog} Q^{k+1}$. At least one step uses a pseudo-empty clause otherwise no variable from $x_1, \dots, x_n$ would be instantiated by a non-variable term containing at least one variable in $x'_1, \dots, x'_n$. We conclude that $P = Q^0 \; op_1 \; Q^1 \; op_2 \; Q^2 \dots Q^{p-1} \; op_p \; Q^p = P$ with each op_i is $>_{Prog}$ or $\trianglerighteq_{Prog}$ and there exists k such that op_k is $>_{Prog}$. Therefore $P >^+_{Prog} P$, so $>_{Prog}$ is cyclic. ◀

So, if a CS-program $Prog$ does not involve $>_{Prog}$ to be cyclic, then all is fine. Otherwise, we have to transform $Prog$ into another CS-program $Prog'$ such as $>_{Prog'}$ is not cyclic and $Mod(Prog) \subseteq Mod(Prog')$.

The transformation is based on the following observation. If $>_{Prog}$ is cyclic, there is at least one pseudo-empty clause over a given predicate that participates in a cycle. Note that this remark can be checked in Example 24 where $P(x, h(y), f(z)) \leftarrow Q(x, z), R(y)$ is a pseudo-empty clause over Q involving the cycle. To remove cycles, we transform some pseudo-empty clauses into clauses that are not pseudo-empty anymore. It boils down to unsynchronize some variables. The process is mainly described in Definition 28. Definitions 26 and 27 are intermediary definitions involved in Definition 28.

▶ **Definition 26** (simplify)**.** Let $H \leftarrow A_1, \dots, A_n$ be a CS-clause, and for each i, let us write $A_i = P_i(\dots)$.
If there exists P_i s.t. $L(P_i) = \emptyset$ then $\mathsf{simplify}(H \leftarrow A_1, \dots, A_n)$ is the empty set, otherwise it is the set that contains only the clause $H \leftarrow B_1, \dots, B_m$ such that

- $\{B_i \mid 0 \leq i \leq m\} \subseteq \{A_i \mid 0 \leq i \leq n\}$ and
- $\forall i \in \{1, \dots, n\}$, $(\neg(\exists j, B_j = A_i) \Leftrightarrow Var(A_i) \cap Var(H) = \emptyset)$.

In other words, simplify deletes unproductive clauses, or it removes the atoms of the body that contain only extra-variables.

▶ **Definition 27** (unSync)**.** Let $P(t_1, \dots, t_n) \leftarrow B$ be a pseudo-empty CS-clause.

$\mathsf{unSync}(P(t_1, \dots, t_n) \leftarrow B) = \mathsf{simplify}(P(t_1, \dots, t_n) \leftarrow \sigma_0(B), \sigma_1(B))$ where σ_0, σ_1 are substitutions built as follows:

$$\sigma_0(x) = \begin{cases} x & \text{if } \exists i,\, t_i = x \\ \text{a fresh variable otherwise} \end{cases} \qquad \sigma_1(x) = \begin{cases} x & \text{if } \exists i,\, t_i \notin Var \wedge x \in Var(t_i) \\ & \quad \wedge \neg(\exists j,\, t_j = x) \\ \text{a fresh variable otherwise} \end{cases}$$

▶ **Definition 28** (removeCycles)**.** Let $Prog$ be a CS-program.

$$\mathsf{removeCycles}(Prog) = \begin{cases} Prog & \text{if } >_{Prog} \text{ is not cyclic} \\ \mathsf{removeCycles}(\{\mathsf{unSync}(H \leftarrow B)\} \cup Prog') & \text{otherwise} \end{cases}$$

where $H \leftarrow B$ is a pseudo-empty clause involved in a cycle and $Prog' = Prog \setminus \{H \leftarrow B\}$.

▶ **Example 29.** Let $Prog$ be the CS-program of Example 24. Since $Prog$ is cyclic, let us compute removeCycles($Prog$). The pseudo-empty CS-clause $P(x, h(y), f(z)) \leftarrow Q(x, z), R(y)$ is involved in the cycle. Consequently, unSync is applied on it. According to Definition 27, one obtains σ_0 and σ_1 where $\sigma_0 = [x/x, y/x_1, z/x_2]$ and $\sigma_1 = [x/x_3, y/y, z/z]$. Thus, one obtains the CS-clause $P(x, h(y), f(z)) \leftarrow Q(x, x_2), R(x_1), Q(x_3, z), R(y)$. Note that according to Definition 27, simplify has to be applied on the CS-clause above-mentioned. Following Definitions 26 and 28, one has to remove $P(x, h(y), f(z)) \leftarrow Q(x, z), R(y)$ from $Prog$ and to add $P(x, h(y), f(z)) \leftarrow Q(x, x_2), Q(x_3, z), R(y)$ instead. Note that the atom $R(x_1)$ has been removed using simplify. Note also that there is no cycle anymore.

Lemma 30 describes that our transformation preserves at least and may extend the initial least Herbrand Model.

▶ **Lemma 30.** *Let $Prog$ be a CS-program and $Prog'$ =* removeCycles($Prog$)*.*
Then $>_{Prog'}$ is not cyclic, and $Mod(Prog) \subseteq Mod(Prog')$. Moreover, if $Prog$ is normalized and preserving, then so is $Prog'$.

Proof. Proof are detailled in [3]. ◀

At this point, given a CS-program $Prog$, if $>_{Prog}$ is not cyclic then the number of critical pairs is finite. Otherwise, we transform $Prog$ into another CS-program $Prog'$ in such a way that $>_{Prog'}$ is not cyclic and $Mod(Prog) \subseteq Mod(Prog')$. Since $Prog'$ is not cyclic, the finiteness of the number of critical pairs is ensured.

3.3 Normalizing critical pairs

In Section 3.1, we have defined the notion of critical pair and we have shown in Theorem 14 that this notion is useful for a matter of rewriting closure. Moreover, as mentioned at the very beginning of Section 3, non-convergent critical pairs correspond to the CS-clauses that we would like to add in the current CS-program. Unfortunately, these CS-clauses are not necessarily in the expected form (normalized).

Definition 34 describes the normalization process that transforms a non-normalized CS-clause into several normalized ones. For example, consider the non-normalized CS-clause $P(f(g(x)), b) \leftarrow P'(x)$. We want to generate a set of normalized CS-clauses covering at least the same Herbrand model. The following set of CS-clauses $\{P(f(x_1), b) \leftarrow P_{new_1}(x_1).\ P_{new_1}(g(x_1)) \leftarrow P'(x_1).\}$ is a good candidate with P_{new_1} a new predicate symbol.

Definition 31 introduces tools for manipulating parameters of predicates (tuple of terms). Definition 32 formalizes a way for cutting a clause head, at depth 1. An example is given after Definition 34.

▶ **Definition 31.** A tree-tuple $(t_1, \ldots, t_n)$ is *normalized* if for all i, t_i is a variable or contains only one function-symbol.
We define tuple concatenation by $(t_1, \ldots, t_n).(s_1, \ldots, s_k) = (t_1, \ldots, t_n, s_1, \ldots, s_k)$.
The arity of the tuple $(t_1, \ldots, t_n)$ is $ar(t_1, \ldots, t_n) = n$.

▶ **Definition 32.** Consider a tree-tuple $\overrightarrow{t} = (t_1, \ldots, t_n)$. We define :

- $\overrightarrow{t}^{cut} = (t_1^{cut}, \ldots, t_n^{cut})$, where $t_i^{cut} = \left| \begin{array}{l} x'_{i,1} \text{ if } t_i \text{ is a variable} \\ t_i \text{ if } t_i \text{ is a constant} \\ t_i(\epsilon)(x'_{i,1}, \ldots, x'_{i,ar(t_i(\epsilon))}) \text{ otherwise} \end{array} \right.$

  and variables $x'_{i,k}$ are new variables that do not occur in $\overrightarrow{t}$.

- for each i, $\overrightarrow{Var}(t_i^{cut})$ is the (possibly empty) tuple composed of the variables of t_i^{cut} (taken in the left-right order).
- $\overrightarrow{Var}(\overrightarrow{t}^{cut}) = \overrightarrow{Var}(t_1^{cut}) \ldots \overrightarrow{Var}(t_n^{cut})$ (concatenation of tuples).
- for each i, t_i^{rest} is the tree-tuple $t_i^{rest} = \left| \begin{array}{l} (t_i) \text{ if } t_i \text{ is a variable} \\ \text{the empty tuple if } t_i \text{ is a constant} \\ (t_i|_1, \ldots, t_i|_{ar(t_i(\epsilon))}) \text{ otherwise} \end{array} \right.$
- $\overrightarrow{t}^{rest} = (t_1^{rest} \ldots t_n^{rest})$ (concatenation of tuples).

▶ **Example 33.** Let $\overrightarrow{t}$ be a tree-tuple such that $\overrightarrow{t} = (x_1, x_2, g(x_3, h(x_1)), h(x_4), b)$ where x_i's are variables. Thus,

- $\overrightarrow{t}^{cut} = (y_1, y_2, g(y_3, y_4), h(y_5), b)$ with y_i's new variables;
- $\overrightarrow{Var}(\overrightarrow{t}^{cut}) = (y_1, y_2, y_3, y_4, y_5)$;
- $\overrightarrow{t}^{rest} = (x_1, x_2, x_3, h(x_1), x_4)$.

Note that $\overrightarrow{t}^{cut}$ is normalized, $\overrightarrow{Var}(\overrightarrow{t}^{cut})$ is linear, $\overrightarrow{Var}(\overrightarrow{t}^{cut})$ and $\overrightarrow{t}^{rest}$ have the same arity.

Notation: $card(S)$ denotes the number of elements of the finite set S.

▶ **Definition 34** (norm)**.** Let $Prog$ be a normalized CS-program.
Let $Pred$ be the set of predicate symbols of $Prog$, and for each positive integer i, let $Pred_i = \{P \in Pred \mid ar(P) = i\}$ where ar means *arity*.
Let *arity-limit* and *predicate-limit* be positive integers s.t. $\forall P \in Pred$, $arity(P) \leq$ *arity-limit*, and $\forall i \in \{1, \ldots,$ *arity-limit*$\}$, $card(Pred_i) \leq$ *predicate-limit*. Let $H \leftarrow B$ be a CS-clause.

Function $\mathsf{norm}_{Prog}(H \leftarrow B)$
Res = Prog
If $H \leftarrow B$ is normalized
then Res = Res $\cup \{H \leftarrow B\}$ (a)
else If $H \rightarrow_{Res} A$ by a synchronizing and non-empty clause
  then (note that A is an atom) Res = $\mathsf{norm}_{Res}(A \leftarrow B)$ (b)
  else let us write $H = P(\overrightarrow{t})$
    If $ar(\overrightarrow{Var}(\overrightarrow{t}^{cut})) \leq$ *arity-limit*
    then let c' be the clause $P(\overrightarrow{t}^{cut}) \leftarrow P'(\overrightarrow{Var}(\overrightarrow{t}^{cut}))$
      where P' is a new or an existing predicate symbol[6]
      $Res = \mathsf{norm}_{Res \cup \{c'\}}(P'(\overrightarrow{t}^{rest}) \leftarrow B)$ (c)
    else choose tuples $\overrightarrow{vt_1}, \ldots, \overrightarrow{vt_k}$ and tuples $\overrightarrow{tt_1}, \ldots, \overrightarrow{tt_k}$ s.t.
      $\overrightarrow{vt_1} \ldots \overrightarrow{vt_k} = \overrightarrow{Var}(\overrightarrow{t}^{cut})$ and $\overrightarrow{tt_1} \ldots \overrightarrow{tt_k} = \overrightarrow{t}^{rest}$,
      and for all j, $ar(\overrightarrow{vt_j}) = ar(\overrightarrow{tt_j})$ and $ar(\overrightarrow{vt_j}) \leq$ *arity-limit*
     let c' be the clause $P(\overrightarrow{t}^{cut}) \leftarrow P'_1(\overrightarrow{vt_1}), \ldots, P'_k(\overrightarrow{vt_k})$
      where $P'_1, \ldots, P'_k$ are new or existing predicate symbols[7]
     Res = Res $\cup \{c'\}$
     For j=1 to k **do** $Res = \mathsf{norm}_{Res}(P'_j(\overrightarrow{tt_j}) \leftarrow B)$ **EndFor** (d)
    EndIf
  EndIf
EndIf
return Res

---

[6] If $card(Pred_{ar(\overrightarrow{Var}(\overrightarrow{t}^{cut}))}(Res)) <$ *predicate-limit*, then P' is new, otherwise P' is arbitrarily chosen in $Pred_{ar(\overrightarrow{Var}(\overrightarrow{t}^{cut}))}(Res)$.

[7] For all j, P'_j is new iff $card(Pred_{ar(\overrightarrow{vt_j})}(Res)) + j - 1 <$ *predicate-limit*.

▶ **Example 35.** Consider the CS-program $Prog =$

$\{P_0(f(x)) \leftarrow P_1(x).\ \ P_1(a) \leftarrow .\ \ P_0(u(x)) \leftarrow P_2(x).\ \ P_2(f(x)) \leftarrow P_3(x).\ \ P_3(v(x,x)) \leftarrow P_1(x).\}$

Let *arity-limit* $= 1$ and *predicate-limit* $= 5$. Let $P_2(u(f(v(x,x)))) \leftarrow P_3(x)$ be a CS-clause to normalize. According to Definition 34, we are not in case (a) nor in (b), we are in case (c). Then, according to Definition 32, $\overrightarrow{u(f(v(x,x)))}^{cut} = u(x_1)$ with x_1 a new variable. Since for now the number of predicates with arity 1 is equal to $4 <$ *predicate-limit*, a new predicate P_4 can be created and then one has to add the CS-clause $P_2(u(x_1)) \leftarrow P_4(x_1)$. Then we have to solve the recursive call $\mathsf{norm}_{Prog \cup \{P_2(u(x_1)) \leftarrow P_4(x_1)\}}(P_4(f(v(x,x))) \leftarrow P_3(x))$. The same process is applied except for the creation of a new predicate, because *predicate-limit* would be exceeded. Consequently, no new predicate with arity 1 can be generated. One has to choose an existing one. Let us try with P_3. So, the CS-clause $P_4(f(x_2)) \leftarrow P_3(x_2)$ is added into $Prog$ (because $\overrightarrow{f(v(x,x))}^{cut} = f(x_2)$) and then, norm is called with the parameter $P_3(v(x,x)) \leftarrow P_3(x)$. Finally, $P_3(v(x,x)) \leftarrow P_3(x)$ is also added into $Prog$ since this clause is already normalized. To summarize, the normalization of the CS-clause $P_2(u(f(v(x,x)))) \leftarrow P_3(x)$ has produced three new clauses, which are $P_2(u(x_1)) \leftarrow P_4(x_1)$, $P_4(f(x_2)) \leftarrow P_3(x_2)$ and $P_3(v(x,x)) \leftarrow P_3(x)$.

Obviously, termination of norm is guaranteed according to Lemma 36.

▶ **Lemma 36.** *Function* norm *always terminates.*

Proof. Consider a run of $\mathsf{norm}_{Prog}(H \leftarrow B)$, and any recursive call $\mathsf{norm}_{Prog'}(H' \leftarrow B')$. We can see that $|H'|_\Sigma < |H|_\Sigma$. Consequently a normalized clause is necessarily reached, and there is no recursive call in this case. ◀

Given a normalized CS-program $Prog$, Theorem 37 raises two important points:
1. given a non-normalized clause $H \leftarrow B$, one obtains $H \rightarrow_{\mathsf{norm}_{Prog}(H \leftarrow B)} B$, and 2. adding the CS-clauses provided by norm into $Prog$ may increase the least Herbrand model of $Prog$.

▶ **Theorem 37.** *Let c be a critical pair in $Prog$. Then c is convergent in* $\mathsf{norm}_{Prog}(c)$*. Moreover for any CS-clause c', we have* $Mod(Prog \cup \{c'\}) \subseteq Mod(\mathsf{norm}_{Prog}(c'))$.

Proof. The second item of the theorem is a consequence of the first item.
Let us now prove the first item. Let $c = (H \leftarrow B)$ and let us prove that $H \rightarrow^*_{Res} B$. The proof is by induction on recursive calls to Function norm (we write *ind-hyp* for "induction hypothesis"). We consider items (a), (b),... in Definition 34 :

(a) From Lemma 30.
(b) We have $H \rightarrow A \rightarrow^*_{ind-hyp} B$.
(c) $H = P(\overrightarrow{t}) \rightarrow_{c'} P'(\overrightarrow{t}^{rest}) \rightarrow^*_{ind-hyp} B$.
(d) $H = P(\overrightarrow{t}) \rightarrow_{c'} (P'_1(\overrightarrow{tt_1}), \ldots, P'_k(\overrightarrow{tt_k})) \rightarrow^*_{ind-hyp} (B, \ldots, B)$ (up to variable renamings).

◀

3.4 Completion

In Sections 3.1 and 3.3, we have described how to detect critical pairs and how to convert them into normalized clauses. Moreover, in a given finite CS-program the number of critical pairs is finite as shown in Section 3.2. Definition 38 explains precisely our technique for computing over-approximation using a CS-program completion.

▶ **Definition 38** (comp)**.** Let R be a left-linear rewrite system, and $Prog$ be a finite and normalized CS-program s.t.

- $>_{Prog}$ is not cyclic (otherwise apply removeCycles to remove cycles),
- and $\forall P \in Pred$, $arity(P) \leq arity\text{-}limit$,
- and $\forall i \in \{1, \dots, arity\text{-}limit\}$, $card(Pred_i) \leq predicate\text{-}limit$.

where $card(Pred_i)$ is the number of predicate symbols of arity i.

Function $\mathsf{comp}_R(Prog)$

while there exists a non-convergent critical pair $H \leftarrow B$ **do**

Prog = $\mathsf{removeCycles}(\mathsf{norm}_{Prog}(H \leftarrow B))$

end while

return Prog

Theorem 39 and Corollary 40 illustrate that our technique leads to a finite CS-program whose least Herbrand model over-approximates the descendants obtained by a left-linear rewrite system R.

▶ **Theorem 39.** *Function* comp *always terminates, and all critical pairs are convergent in* $\mathsf{comp}_R(Prog)$*. Moreover* $Mod(Prog) \subseteq Mod(\mathsf{comp}_R(Prog))$*.*

Proof. Proofs are detailled in [3]. ◀

Moreover, thanks to Theorem 14, $Mod(\mathsf{comp}_R(Prog))$ is closed under rewriting by R. Then:

▶ **Corollary 40.** *If in addition Prog is preserving,* $R^*(Mod(Prog)) \subseteq Mod(\mathsf{comp}_R(Prog))$*.*

4 Examples

In this section, our technique is applied on several examples. In Examples 41, 42 and 43, I is the initial set of terms and R is the rewrite system. Moreover, initially, we define a CS-program $Prog$ that generates I.

▶ **Example 41.** In this example, we define Σ as follows: $\Sigma = \{c^{\backslash 2}, a^{\backslash 0}\}$. Let I be the set of terms $I = \{f(t) \mid t \in T_\Sigma\}$. Let R be the rewrite system $R = \{f(x) \to b(x,x)\}$. Obviously, one can easily guess that $R^*(I) = \{b(t,t) \mid t \in T_\Sigma\} \cup I$. Note that $R^*(I)$ is not a regular, nor a context-free language [1, 13].

Initially, $Prog = \{P_0(f(x)) \leftarrow P_1(x).\ \ P_1(c(x,y)) \leftarrow P_1(x), P_1(y).\ \ P_1(a) \leftarrow .\}$. Using our approach, the critical pair $P_0(b(x,x)) \leftarrow P_1(x)$ is detected. This critical pair is already normalized, then it is immediately added into $Prog$. Then, there is no more critical pair and the procedure stops. Note that we get exactly the set of descendants, i.e. $L(P_0) = R^*(I)$. So, given $t, t' \in T_\Sigma$ such that $t \neq t'$, one can show that $b(t,t') \notin R^*(I)$.

The example right above shows that non-context-free descendants can be handled in a conclusive manner with our approach. Such example cannot be handled by [14] in an exact way, because they use context-free languages. Actually, the classes of languages covered by our approach and theirs are in some sense orthogonal. However, the examples below shows that our approach can also be relevant for other problems.

▶ **Example 42.**
Let I be the set of terms $I = \{f(a,a)\}$, and R be the rewrite system $R = \{f(x,y) \to u(f(v(x), w(y)))\}$. Intuitively, the exact set of descendants is $R^*(I) = \{u^n(f(v^n(a), w^n(a))) \mid n \in \mathbb{N}\}$. We define $Prog = \{P_0(f(x,y)) \leftarrow P_1(x), P_1(y).\ \ P_1(a) \leftarrow .\}$. We choose $predicate\text{-}limit = 4$ and $arity\text{-}limit = 2$.

First, the following critical pair is detected: $P_0(u(f(v(x), w(y)))) \leftarrow P_1(x), P_1(y)$. According to Definition 34, the normalization of this critical pair produces three new CS-clauses: $P_0(u(x)) \leftarrow P_2(x)$, $P_2(f(x,y)) \leftarrow P_3(x,y)$ and $P_3(v(x), w(y)) \leftarrow P_1(x), P_1(y)$. Adding these three CS-clauses into *Prog* produces the new critical pair $P_2(u(f(v(x), w(y))))$ $\leftarrow P_3(x,y)$. This critical pair can be normalized without exceeding *predicate-limit*. So, we add: $P_2(u(x)) \leftarrow P_4(x)$. $P_4(f(x,y)) \leftarrow P_5(x,y)$. and $P_5(v(x), w(y)) \leftarrow P_3(x,y)$.

Once again, a new critical pair has been introduced: $P_4(u(f(v(x), w(y)))) \leftarrow P_5(x,y)$. Note that, from now, we are not allowed to introduce any new predicate of arity 1. Let us proceed the normalization of $P_4(u(f(v(x), w(y)))) \leftarrow P_5(x,y)$ step by step. We choose to re-use the predicate P_4. Thus, we first generate the following CS-clause: $P_4(u(x)) \leftarrow P_4(x)$. So, we have to normalize now $P_4(f(v(x), w(y))) \leftarrow P_5(x,y)$. Note that $P_4(f(v(x), w(y))) \rightarrow^+_{Prog}$ $P_3(x,y)$. Consequently, the CS-clause $P_3(x,y) \leftarrow P_5(x,y)$ is added into *Prog*.

Note that there is no critical pair anymore.

To summarize, we obtain the final CS-program $Prog_f$ composed of the following CS-clauses:

$$Prog_f = \left\{ \begin{array}{lll} P_0(f(x,y)) \leftarrow P_1(x), P_1(y). & P_1(a) \leftarrow . & P_0(u(x)) \leftarrow P_2(x) \\ P_2(f(x,y)) \leftarrow P_3(x,y). & P_3(v(x), w(y)) \leftarrow P_1(x), P_1(y). & P_2(u(x)) \leftarrow P_4(x). \\ P_4(f(x,y)) \leftarrow P_5(x,y). & P_5(v(x), w(y)) \leftarrow P_3(x,y). & P_4(u(x)) \leftarrow P_4(x). \\ P_3(x,y) \leftarrow P_5(x,y) & & \end{array} \right\}$$

For $Prog_f$, note that $L(P_0) = \{u^n(f(v^m(a), w^m(a))) \mid n, m \in \mathbb{N}\}$ and $R^*(I) \subseteq L(P_0)$.

In Example 42, the approximation computed is still a non-regular language. Nevertheless, it is a strict over-approximation since a synchronization is broken between the three counters.

Let us also show the application of our technique on an example introduced in [5]. In [5] authors propose an example that cannot be handled by regular approximations. Example 43 shows that this limitation can now be overcome.

▶ **Example 43.** Let I be the set of terms $I = \{f(a,a)\}$ and R be the rewrite system $R = \{f(x,y) \rightarrow f(g(x), g(y)),\ f(g(x), g(y)) \rightarrow f(x,y),\ f(a, g(a)) \rightarrow error\}$. Obviously, $R^*(I) = \{f(g^n(a), g^n(a)) \mid n \in \mathbb{N}\}$. Consequently, *error* is not a reachable term.

We start with the CS-program $Prog = \{P_0(f(x,y)) \leftarrow P_1(x), P_1(y).\ \ P_1(a) \leftarrow .\}$. After applying Function `comp`, we obtain the following CS-program for any *predicate-limit* ≥ 2:

$$Prog_f = \left\{ \begin{array}{lll} P_0(f(x,y)) \leftarrow P_1(x), P_1(y). & P_0(f(x,y)) \leftarrow P_2(x,y) & P_1(a) \leftarrow . \\ P_2(g(x), g(y)) \leftarrow P_1(x), P_1(y). & P_2(g(x), g(y)) \leftarrow P_2(x,y). & \end{array} \right\}$$

Note that $L(P_0)$ is exactly $R^*(I)$. Note also that $error \notin L(P_0)$. Consequently, we have proved that *error* is not reachable from I.

5 Further Work

We have presented a procedure that always terminates and that computes an over-approximation of the set of descendants, expressed by a synchronized tree language. This is the first attempt using synchronized tree languages. It could be improved or extended:

- In Definition 34, when *predicate-limit* is reached (in items (c) and (d)), an (several in item (d)) existing predicate of the right arity is chosen arbitrarily and re-used, instead of creating a new one. Of course, if there are several existing predicates of the right arity, the achieved choice affects the quality of the approximation. When using regular languages [8], a similar difficulty happens: to make the procedure terminate, it is sometimes necessary to chose and re-use an existing state instead of creating a new one. Some ideas have been proposed to make this choice in a smart way [11]. We are going to extend these ideas in order to improve the choice of existing predicates.

- A similar problem arises when *arity-limit* is reached (item (d)): a tuple is divided into several smaller tuples in an arbitrary way, and there may be several possibilities, which may affect the quality of the approximation.
- To compute descendants, we have used synchronized tree languages, whereas context-free languages have been used in [14]. Each approach has advantages and drawbacks. Therefore, it would be interesting to mix the two approaches to get the advantages of both.

References

1 A. Arnold and M. Dauchet. Un théorème de duplications pour les forêts algébriques. In *Journal of Computer and System Sciences*, pages 13:223–244, 1976.

2 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.

3 Y. Boichut, J. Chabin, and P. Rety. Over-approximating Descendants by Synchronized Tree Languages. Technical Report RR-2013-04, University of Orléans, LIFO, 2013.

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 M. Dauchet. Simulation of Turing Machines by a Left-Linear Rewrite Rule. In *RTA*, volume 355 of *LNCS*, pages 109–120. Springer, 1989.

8 Th. Genet. Decidable Approximations of Sets of Descendants and Sets of Normal Forms. In *RTA*, volume 1379 of *LNCS*, pages 151–165. Springer-Verlag, 1998.

9 Th. Genet, Th. P. Jensen, V. Kodati, and D. Pichardie. A Java Card CAP Converter in PVS. *ENTCS*, 82(2):426–442, 2003.

10 Th. Genet and F. Klay. Rewriting for Cryptographic Protocol Verification. In *CADE*, volume 1831 of *LNAI*, pages 271–290. Springer-Verlag, 2000.

11 Th. Genet and V. Viet Triem Tong. Reachability Analysis of Term Rewriting Systems with Timbuk. In *LPAR*, volume 2250 of *LNCS*, pages 695–706. Springer, 2001.

12 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.

13 D. Hofbauer, M. Huber, and G. Kucherov. Some Results on Top-context-free Tree Languages. In *CAAP*, volume 787 of *LNCS*, pages 157–171. Springer-Verlag, 1994.

14 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.

15 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.

16 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.

17 Sébastien Limet and Gernot Salzer. Tree Tuple Languages from the Logic Programming Point of View. *Journal of Automated Reasoning*, 37(4):323–349, 2006.