Towards More Precise Rewriting Approximations

Yohan Boichut(✉), Jacques Chabin, and Pierre Réty

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

Abstract. To check a system, some verification techniques consider a set of terms I that represents the initial configurations of the system, and a rewrite system R that represents the system behavior. To check that no undesirable configuration is reached, they compute an over-approximation of the set of descendants (successors) issued from I by R, expressed by a tree language. Their success highly depends on the quality of the approximation. Some techniques have been presented using regular tree languages, and more recently using non-regular languages to get better approximations: using context-free tree languages [16] on the one hand, using synchronized tree languages [2] on the other hand. In this paper, we merge these two approaches to get even better approximations: we compute an over-approximation of the descendants, using synchronized-context-free tree languages expressed by logic programs. We give several examples for which our procedure computes the descendants in an exact way, whereas the former techniques compute a strict over-approximation.

Keywords: Term rewriting · Tree languages · Logic programming · Reachability

1 Introduction

To check systems like cryptographic protocols or Java programs, some verification techniques consider a set of terms I that represents the initial configurations of the system, and a rewrite system R that represents the system behavior [1,13,14]. To check that no undesirable configuration is reached, they compute an over-approximation of the set of descendants[1] (successors) issued from I by R, expressed by a tree language. 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, one computes an over-approximation App of $R^*(I)$ (i.e. $App \supseteq R^*(I)$), and checks that $App \cap Bad = \emptyset$, which ensures that the protocol or the program is correct.

However, I, Bad and App have often been considered as regular tree languages, recognized by finite tree automata. In the general case, $R^*(I)$ is not

[1] I.e. terms obtained by applying arbitrarily many rewrite steps on the terms of I.

A.-H. Dediu et al. (Eds.): LATA 2015, LNCS 8977, pp. 652–663, 2015.
DOI: 10.1007/978-3-319-15579-1_51

regular, even if I is. Moreover, the expressiveness of regular languages is poor. Then 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) to conclude as often as possible [1,4,6]. However, in some cases, no regular over-approximation works [5].

To overcome this theoretical limit, the idea is 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, if we 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 approximation technique using context-free tree languages has been proposed in [16]. On the other hand, the class of synchronized tree languages [17] also has these properties, and an approximation technique using synchronized tree languages has been proposed in [2]. Both classes include regular languages, but they are incomparable. Context-free tree languages cannot express dependencies between different branches, except in some cases, whereas synchronized tree languages cannot express vertical dependencies.

We want to use a more powerful class of languages that can express the two kinds of dependencies together: the class of synchronized-context-free tree-(tuple) languages [21,22], which has the same properties as context-free languages and as synchronized languages, i.e. closure under union, closure under intersection with a regular language, decidability of membership and emptiness.

In this paper, we propose a procedure that always terminates and that computes an over-approximation of the descendants obtained by a linear rewrite system, using synchronized-context-free tree-(tuple) languages expressed by logic programs. Compared to our previous work [2], we introduce "input arguments" in predicates, which is a major technical change that highly improves the quality of the approximation, and that requires new results and new proofs. This work is a first step towards a verification technique offering more than regular approximations. Some on-going work is discussed in Section 5 in order to make this technique be an accepted verification technique.

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, is explained in Section 3. Finally, in Section 4 our technique is applied to examples, in particular when $R^*(I)$ can be expressed in an exact way neither by a context-free language, nor by a synchronized language. For lack of space, all proofs are in [3].

Related Work: The class of tree-tuples whose overlapping coding is recognized by a tree automaton on the product alphabet [7] (called "regular tree relations" by some authors), is strictly included in the class of rational tree relations [19]. The latter is equivalent to the class of non-copying[2] synchronized languages [20], which is strictly included in the class of synchronized languages.

[2] Clause heads are assumed to be linear.

Context-free tree languages (i.e. without assuming a particular strategy for grammar derivations) [23] are equivalent to OI (outside-in strategy) context-free tree languages, but are incomparable with IO (inside-out strategy) context-free tree languages [11,12]. The IO class (and not the OI one) is strictly included in the class of synchronized-context-free tree languages. The latter is equivalent to the "term languages of hyperedge replacement grammars", which are equivalent to the tree languages definable by attribute grammars [9,10]. However, we prefer to use the synchronized-context-free tree languages, which use the well known formalism of pure logic programming, for its implementation ease.

Much other work computes the descendants in an exact way using regular tree languages (in particular the recent paper [8]), assuming strong restrictions.

2 Preliminaries

Consider a *finite ranked alphabet* $\Sigma = \{a, b, f, g, h, \ldots\}$ and a set of variables $Var = \{x, y, z, \ldots\}$. 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 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 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.

In the following, we consider the framework of *pure logic programming*, and the class of synchronized-context-free tree-tuple[3] languages [21,22], which is presented as an extension of the class of synchronized tree-tuple languages defined by CS-clauses [17,18]. 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.

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

[3] For simplicity, "tree-tuple" is sometimes omitted.

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

A Horn clause $P(t_1, \ldots, t_n) \leftarrow B$ is normalized *if $\forall i \in \{1, \ldots, n\}$, t_i is a variable or contains only one occurrence of function-symbol. A program is* normalized *if all its clauses are normalized.*

Example 2. Let x, y, z be variables. The sequence of atoms $P_1(x, y), P_2(z)$ is flat, whereas $P_1(x, f(y))$, $P_2(z)$ is not flat. The clause $P(x, y) \leftarrow Q(x, y)$ is normalized (x, y are variables). The clause $P(f(x), y) \leftarrow Q(x, y)$ is normalized whereas $P(f(f(x)), y) \leftarrow Q(x, y)$ is not.

Definition 3. A *logic program with modes* is a logic program such that a mode-tuple $\boldsymbol{m} \in \{I, O\}^n$ is associated to each predicate symbol P (n is the arity of P). In other words, each predicate argument has mode I (Input) or O (Output). To distinguish them, **output arguments will be covered by a hat**.

Notation: Let P be a predicate symbol. $ArIn(P)$ is the number of input arguments of P, and $ArOut(P)$ is the number of output arguments. Let B be a sequence of atoms (possibly containing only one atom). $In(B)$ is the input part of B, i.e. the tuple composed of the input arguments of B. $ArIn(B)$ is the arity of $In(B)$. $Var^{in}(B)$ is the set of variables that appear in $In(B)$. $Out(B)$, $ArOut(B)$, and $Var^{out}(B)$ are defined in a similar way. We also define $Var(B) = Var^{in}(B) \cup Var^{out}(B)$.

Example 4. Let $B = P(\widehat{t_1}, \widehat{t_2}, t_3), Q(\widehat{t_4}, t_5, t_6)$. Then, $Out(B) = (t_1, t_2, t_4)$ and $In(B) = (t_3, t_5, t_6)$.

Definition 5. Let $B = A_1, \ldots, A_n$ be a sequence of atoms. We say that $A_j \succ A_k$ (possibly $j = k$) if $\exists y \in Var^{in}(A_j) \cap Var^{out}(A_k)$. In other words an input of A_j depends on an output of A_k. We say that B has a *loop* if $A_j \succ^+ A_j$ for some A_j ($\succ^+$ is the transitive closure of $\succ$).

Example 6. $Q(\widehat{x}, s(y))$, $R(\widehat{y}, s(x))$ (where x, y are variables) has a loop because $Q(\widehat{x}, s(y)) \succ R(\widehat{y}, s(x)) \succ Q(\widehat{x}, s(y))$.

Definition 7. A *Synchronized-Context-Free (S-CF) program* $Prog$ is a logic program with modes, whose clauses $H \leftarrow B$ satisfy:

- $In(H).Out(B)$ ($.$ is the tuple concatenation) is a linear tuple of variables, i.e. each tuple-component is a variable, and each variable occurs only once,
- and B does not have a loop.

A clause of an S-CF program is called *S-CF clause*.

Example 8. $Prog = \{P(\widehat{x}, y) \leftarrow P(\widehat{s(x)}, y)\}$ is not an S-CF program because $In(H).Out(B) = (y, s(x))$ is not a tuple of variables. $Prog' = \{P'(\widehat{s(x)}, y) \leftarrow P'(\widehat{x}, s(y))\}$ is an S-CF program because $In(H).Out(B) = (y, x)$ is a linear tuple of variables, and there is no loop in the clause body.

Definition 9. Let $Prog$ be an S-CF program. Given a predicate symbol P without input arguments, the tree-(tuple) language generated by P is $L_{Prog}(P) = \{\boldsymbol{t} \in (T_\Sigma)^{ArOut(P)} \mid P(\boldsymbol{t}) \in Mod(Prog)\}$, 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-Context-Free language (S-CF language).*

Example 10. $Prog = \{S(\widehat{c(x,y)}) \leftarrow P(\widehat{x},\widehat{y},a,b).$
$P(\widehat{f(x)},\widehat{g(y)},x',y') \leftarrow P(\widehat{x},\widehat{y},h(x'),i(y'))$. $P(\widehat{x},\widehat{y},x,y) \leftarrow\}$ is an S-CF program. The language generated by S is $L_{Prog}(S) = \{c(f^n(h^n(a)), g^n(i^n(b))) \mid n \in \mathbb{N}\}$, which is not synchronized (there are vertical dependencies) nor context-free.

Definition 11. The S-CF clause $H \leftarrow B$ is *non-copying* if the tuple $Out(H)$. $In(B)$ is linear. A program is *non-copying* if all its clauses are non-copying.

Example 12. The clause $P(\widehat{d(x,x)},y) \leftarrow Q(\widehat{x},p(y))$ is copying whereas $P(\widehat{c(x)}, y) \leftarrow Q(\widehat{x},p(y))$ is non-copying.

Remark 13. An S-CF program without input arguments is actually a CS-program (composed of CS-clauses) [17], which generates a synchronized language[4]. A non-copying CS-program such that every predicate symbol has only one argument generates a regular tree language[5]. Conversely, every regular tree language can be generated by a non-copying CS-program.

Definition 14. *Given an S-CF program $Prog$ and a sequence of atoms G,*

- *G derives into G' by a* resolution *step if there exists a clause*[6] *$H \leftarrow B$ 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 $G \rightsquigarrow_\sigma G'$. We consider the transitive closure $\rightsquigarrow^+$ and the reflexive-transitive closure $\rightsquigarrow^*$ of $\rightsquigarrow$. If $G_1 \rightsquigarrow_{\sigma_1} G_2$ and $G_2 \rightsquigarrow_{\sigma_2} G_3$, we write $G_1 \rightsquigarrow^*_{\sigma_2 \circ \sigma_1} G_3$.*
- *G* rewrites *into G' (possibly in several steps) if $G \rightsquigarrow^*_\sigma G'$ s.t. σ does not instantiate the variables of G. It is written $G \rightarrow^*_\sigma G'$.*

Example 15. Let $Prog = \{P(\widehat{x_1},\widehat{g(x_2)}) \leftarrow P'(\widehat{x_1},\widehat{x_2})$. $P(\widehat{f(x_1)},\widehat{x_2}) \leftarrow P''(\widehat{x_1}, \widehat{x_2}).\}$, 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 $\sigma_2 = [x_1/x, x_2/y]$.

In the remainder of the paper, given an S-CF program $Prog$ and two sequences of atoms G_1 and G_2, $G_1 \rightsquigarrow^*_{Prog} G_2$ (resp. $G_1 \rightarrow^*_{Prog} G_2$) also denotes that G_2 can be derived (resp. rewritten) from G_1 using clauses of $Prog$. Note that for any atom A, if $A \rightarrow B$ then $A \rightsquigarrow B$. On the other hand, $A \rightsquigarrow_\sigma B$ implies $\sigma(A) \rightarrow B$. Consequently, if A is ground, $A \rightsquigarrow B$ implies $A \rightarrow B$.

It is well known that resolution is complete.

Theorem 16. *Let A be a ground atom. $A \in Mod(Prog)$ iff $A \rightsquigarrow^*_{Prog} \emptyset$.*

[4] Initially, synchronized languages were presented using constraint systems (sorts of grammars) [15], and later using logic programs. CS stands for "Constraint System".
[5] In this case, the S-CF program can easily be transformed into a finite tree automaton.
[6] We assume that the clause and G have distinct variables.

3 Computing Descendants

To make the understanding easier, we first give the completion algorithm in Definition 17. Given a normalized S-CF program $Prog$ and a linear rewrite system R, we propose an algorithm to compute a normalized S-CF program $Prog'$ such that $R^*(Mod(Prog)) \subseteq Mod(Prog')$, and consequently $R^*(L_{Prog}(P)) \subseteq L_{Prog'}(P)$ for each predicate symbol P. Some notions, as strong coherence, will be explained later.

Definition 17 (comp). *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 S-CF program strongly coherent with R. 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 $Prog$

Let us explain this algorithm.

The notion of critical pair is at the heart of the technique. Given an S-CF program $Prog$, a predicate symbol P and a rewrite rule $l \rightarrow r$, a critical pair, explained in details in Section 3.1, is a way to detect a possible rewriting by $l \rightarrow r$ for a term t in a tuple of $L_{Prog}(P)$. A convergent critical pair means that the rewrite step is already handled i.e. if $t \rightarrow_{l \rightarrow r} s$ then s is in a tuple of $L_{Prog}(P)$. Consequently, the language of a normalized CS-program involving only convergent critical pairs is closed by rewriting.

To summarize, a non-convergent critical pair gives rise to an S-CF clause. Adding the resulting S-CF clause to the current S-CF program makes the critical pair convergent. But, let us emphasize on the main problems arising from Definition 17, i.e. the computation may not terminate and the resulting S-CF clause may not be normalized. Concerning the non-termination, there are mainly two reasons. Given a normalized S-CF program $Prog$, 1) the number of critical pairs may be infinite and 2) even if the number of critical pairs is finite, adding the critical pairs to $Prog$ may create new non-convergent critical pairs, and so on.

Actually, as in [2], there is a function called removeCycles whose goal is to get finitely many critical pairs from a given finite S-CF program. For lack of space, many details on this function are given in [3]. Basically, given an S-CF program $Prog$ having infinitely many critical pairs, $\mathsf{removeCycles}(Prog)$ is another S-CF program that has finitely many critical pairs, and such that for any predicate symbol P, $L_{Prog}(P) \subseteq L_{\mathsf{removeCycles}(Prog)}(P)$. The normalization process presented in Section 3.2 not only preserves the normalized nature of the computed S-CF programs but also allows us to control the creation of new non-convergent critical pairs. Finally, in Section 3.3, our main contribution, i.e. the computation of an over-approximating S-CF program, is fully described.

3.1 Critical Pairs

Definition 18. *Let Prog be a non-copying S-CF program and $l \to r$ be a left-linear rewrite rule. Let $x_1, \ldots, x_n$ be distinct variables such that $\{x_1, \ldots, x_n\} \cap Var(l) = \emptyset$. If there are P and k s.t. the k^{th} argument of P is an output, and $P(x_1, \ldots, x_{k-1}, l, x_{k+1}, \ldots, x_n) \leadsto_\theta^+ G$ where*[7]

1. *resolution steps are applied only on atoms whose output is not flat,*
2. *$Out(G)$ is flat and*
3. *the clause $P(t_1, \ldots, t_n) \leftarrow B$ used in the first step of this derivation satisfies t_k is not a variable*[8]

then the clause $\theta(P(x_1, \ldots, x_{k-1}, r, x_{k+1}, \ldots, x_n)) \leftarrow G$ is called critical pair. *Moreover, if θ does not instantiate the variables of $In(P(x_1, \ldots, x_{k-1}, l, x_{k+1}, \ldots, x_n))$ then the critical pair is said* strict.

Example 19. Let *Prog* be the S-CF program defined by:
$Prog = \{P(\widehat{x}) \leftarrow Q(\widehat{x}, a).\ \ Q(\widehat{f(x)}, y) \leftarrow Q(\widehat{x}, g(y)).\ \ Q(\widehat{x}, x) \leftarrow .\}$ and consider the rewrite system: $R = \{f(x) \to x\}$. Note that $L(P) = \{f^n(g^n(a)) \mid n \in \mathbb{N}\}$.

We have $Q(\widehat{f(x)}, y) \leadsto_{Id} Q(\widehat{x}, g(y))$ where Id denotes the substitution that leaves every variable unchanged. Since $Out(Q(\widehat{x}, g(y)))$ is flat, this generates the strict critical pair $Q(\widehat{x}, y) \leftarrow Q(\widehat{x}, g(y))$.

Lemma 20. *A strict critical pair is an S-CF clause. In addition, if $l \to r$ is right-linear, a strict critical pair is a non-copying S-CF clause.*

Definition 21. *A critical pair $H \leftarrow B$ is said* convergent *if $H \to_{Prog}^* B$.*

The critical pair of Example 19 is not convergent.

Let us recall that the completion procedure is based on adding the non-convergent critical pairs into the program. In order to preserve the nature of the S-CF program, the computed non-convergent critical pairs are expected to be strict. So we define a sufficient condition on R and *Prog* called *strong coherence*.

Definition 22. Let R be a rewrite system. We consider the smallest set of *consuming* symbols, recursively defined by: $f \in \Sigma$ is *consuming* if there exists a rewrite rule $f(t_1, \ldots, t_n) \to r$ in R s.t. some t_i is not a variable, or r contains at least one consuming symbol.

The S-CF program *Prog* is *strongly coherent* with R if 1) for all $l \to r \in R$, the top-symbol of l does not occur in input arguments of *Prog* and 2) no consuming symbol occurs in clause-heads having input arguments.

In $R = \{f(x) \to g(x),\ g(s(x)) \to h(x)\}$, g is consuming and so is f. Thus $Prog = \{P(\widehat{f(x)}, x) \leftarrow .\}$ is not strongly coherent with R. Note that a CS-program (no input arguments) is strongly coherent with any rewrite system.

[7] Here, we do not use a hat to indicate output arguments because they may occur anywhere depending on P.

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

Lemma 23. *If Prog is a normalized S-CF program strongly coherent with R, then every critical pair is strict.*

So, we come to our main result that ensures to get the rewriting closure when every computable critical pair is convergent.

Theorem 24. *Let R be a linear rewrite system, and Prog be a non-copying normalized S-CF program strongly coherent with R. If all strict critical pairs are convergent, then for every predicate symbol P without input arguments, $L(P)$ is closed under rewriting by R, i.e.* $(\boldsymbol{t} \in L(P) \wedge \boldsymbol{t} \rightarrow_R^* \boldsymbol{t'}) \implies \boldsymbol{t'} \in L(P)$.

3.2 Normalizing Critical Pairs – norm_{Prog}

If a critical pair is not convergent, we add it into *Prog*, and the critical pair becomes convergent. However, in the general case, a critical pair is not normalized, whereas all clauses in *Prog* should be normalized. In the case of CS-clauses (i.e. without input arguments), a procedure that transforms a non-normalized clause into normalized ones has been presented [2]. For example, $P(\widehat{f(g(x))}, \widehat{b}) \leftarrow Q(\widehat{x})$ is normalized into $\{P(\widehat{f(x_1)}, \widehat{b}) \leftarrow P_1(\widehat{x_1}).\ \ P_1(\widehat{g(x_1)}) \leftarrow Q(\widehat{x_1}).\}$ (P_1 is a new predicate symbol). Since only output arguments should be normalized, this procedure still works even if there are also input arguments. As new predicate symbols are introduced, possibly with bigger arities, the procedure may not terminate. To make it terminate in every case, two positive integers are used: *predicate-limit* and *arity-limit*. If the number of predicate symbols having the same arity as P_1 (including P_1) exceeds *predicate-limit*, an existing predicate symbol (for example Q) must be used instead of the new predicate P_1. This may enlarge $Mod(Prog)$ in general and may lead to a strict over-approximation. If the arity of P_1 exceeds *arity-limit*, P_1 must be replaced in the clause body by several predicate symbols[9] whose arities are less than or equal to *arity-limit*. This may also enlarge $Mod(Prog)$. See [2] for more details.

In other words $\mathsf{norm}_{Prog}(H \leftarrow B)$ builds a set of normalized S-CF clauses N such that $Mod(Prog \cup \{H \leftarrow B\}) \subseteq Mod(Prog \cup N)$.

However, when starting from a CS-program (i.e. without input arguments), it could be interesting to normalize by introducing input arguments, in order to profit from the bigger expressiveness of S-CF programs, and consequently to get a better approximation of the set of descendants, or even an exact computation, like in Examples 26 and 27 presented in Section 4.

3.3 Completion

At the very beginning of Section 3, we have presented in Definition 17 the completion algorithm i.e. comp_R. In Sections 3.1 and 3.2, we have described how to

[9] For instance, if P_1 is binary and *arity-limit* $= 1$, then $P_1(t_1, t_2)$ should be replaced by the sequence of atoms $P_2(t_1), P_3(t_2)$. Note that the dependency between t_1 and t_2 is lost, which may enlarge $Mod(Prog)$. Symbols P_2 and P_3 are new if it is compatible with *predicate-limit*. Otherwise former predicate symbols should be used instead of P_2 and P_3.

detect non-convergent critical pairs and how to convert them into normalized clauses using norm_{Prog}.

Theorem 25 illustrates that our technique leads to a finite S-CF program whose language over-approximates the descendants obtained by a linear rewrite system R.

Theorem 25. *Function* comp *always terminates, and all critical pairs are convergent in* $\mathsf{comp}_R(Prog)$. *Moreover, for each predicate symbol* P *without input arguments,* $R^*(L_{Prog}(P)) \subseteq L_{\mathsf{comp}_R(Prog)}(P)$.

4 Examples

In this section, our technique is applied on several examples. I is the initial set of terms and R is the rewrite system. Initially, we define an S-CF program $Prog$ that generates I and that satisfies the assumptions of Definition 17. For lack of space, the examples should be as short as possible. To make the procedure terminate shortly, we suppose that *predicate-limit*=1, which means that for all i, there is at most one predicate symbol having i arguments, except for $i = 1$ we allow two predicate symbols having one argument.

When the following example is dealt with synchronized languages, i.e. with CS-programs [2, Example 42], we get a strict over-approximation of the descendants. Now, thanks to the bigger expressive power of S-CF programs, we compute the descendants in an exact way.

Example 26. Let $I = \{f(a, a)\}$ and $R = \{f(x, y) \to u(f(v(x), w(y)))\}$. Intuitively, the exact set of descendants is $R^*(I) = \{u^n(f(v^n(a), w^n(a))) \mid n \in \mathbb{N}\}$ where u^n means that u occurs n times. We define $Prog = \{P_f(\widehat{f(x, y)}) \leftarrow P_a(\widehat{x}), P_a(\widehat{y}).,\ P_a(\widehat{a}) \leftarrow .\}$. Note that $L_{Prog}(P_f) = I$. The run of the completion is given in Fig. 1. The reader can refer to [3] for a detailed explanation. In Fig. 1, the left-most column reports the detected non-convergent critical pairs and the right-most column describes how they are normalized. Note that for the resulting program $Prog$, i.e. clauses appearing in the right-most column, $L_{Prog}(P_f) = R^*(I)$ indeed.

The previous example could probably be dealt in an exact way using the technique of [16] as well, since $R^*(I)$ is a context-free language. It is not the case for the following example, whose language of descendants $R^*(I)$ is not context-free (and not synchronized). It can be handled by S-CF programs in an exact way thanks to their bigger expressive power.

Example 27. Let $I = \{d_1(a, a, a)\}$ and

$$R = \left\{ \begin{array}{ll} d_1(x, y, z) \xrightarrow{1} d_1(h(x), i(y), s(z)), & d_1(x, y, z) \xrightarrow{2} d_2(x, y, z) \\ d_2(x, y, s(z)) \xrightarrow{3} d_2(f(x), g(y), z), & d_2(x, y, a) \xrightarrow{4} c(x, y) \end{array} \right\}$$

Detected non-convergent critical pairs	New clauses obtained by norm_{Prog}
	Starting S-CF program $P_f(\widehat{f(x,y)}) \leftarrow P_a(\widehat{x}), P_a(\widehat{y}).$ $P_a(\widehat{a}) \leftarrow .$
$P_f(u(f(\widehat{v(x)}, w(y)))) \leftarrow P_a(\widehat{x}), P_a(\widehat{y}).$	$P_f(\widehat{z}) \leftarrow P_1(\widehat{z}, x, y), P_a(\widehat{x}), P_a(\widehat{y}).$ $P_1(\widehat{u(z)}, x, y) \leftarrow P_1(\widehat{z}, v(x), w(y)).$ $P_1(\widehat{f(x,y)}, x, y) \leftarrow .$
$\emptyset$	

Fig. 1. Run of comp_R on Example 26

$R^*(I)$ is composed of all terms appearing in the following derivation:

$$d_1(a,a,a) \xrightarrow{1}{}^{n} d_1(h^n(a), i^n(a), s^n(a)) \xrightarrow{2} d_2(h^n(a), i^n(a), s^n(a)) \\ \xrightarrow{3}{}^{k} d_2(f^k(h^n(a)), g^k(i^n(a)), s^{n-k}(a)) \xrightarrow{4} c(f^n(h^n(a)), g^n(i^n(a)))\,.$$

Note that the last rewrite step by rule 4 is possible only when $k = n$. The run of the completion on this example is given in Fig. 2. Black arrows means that the non-convergent critical pair is directly added to $Prog$ since it is already normalized. The reader can find a full explanation of this example in [3].

Note that the subset of descendants $d_2(f^k(h^n(a)), g^k(i^n(a)), s^{n-k}(a))$ can be seen (with $p = n - k$) as $d_2(f^k(h^{k+p}(a)), g^k(i^{k+p}(a\ )), s^p(a))$. Let $Prog'$ be the S-CF program composed of all the clauses except the blue one occurring in the right-most column in Fig. 2. Thus, the reader can check by himself that $L_{Prog'}(P_d)$ is exactly $R^*(I)$.

5 Further Work

Computing approximations more precise than regular approximations is a first step towards a verification technique. However, there are at least two steps before claiming this technique as a verification technique: 1) automatically handling the choices done during the normalization process and 2) extending our technique to any rewrite system. The quality of the approximation is closely related to those choices. On one hand, it depends on the choice of the predicate symbol to be reused when *predicate-limit* is reached. On the other hand, the choice of generating function-symbols as output or as input is also crucial. According to the verification context, some automated heuristics will have to be designed in order to obtain well-customized approximations.

Ongoing work tends to show that the linear restriction concerning the rewrite system can be tackled. A non right-linear rewrite system makes the computed S-CF program copying. Consequently, Theorem 24 does not hold anymore. To get rid of the right-linearity restriction, we are studying the transformation of a copying S-CF clause into non-copying ones that will generate an over-approximation.

Detected non-convergent critical pairs	New clauses obtained by norm_{Prog}
	Starting S-CF program $P_d(d_1(\widehat{x,y,z})) \leftarrow P_a(\widehat{x}), P_a(\widehat{y}), P_a(\widehat{z})$. $P_a(\widehat{a}) \leftarrow .$
$P_d(d_1(h(x),\widehat{i(y)},s(z))) \leftarrow P_a(\widehat{x}), P_a(\widehat{y}), P_a(\widehat{z})$	$P_d(d_1(\widehat{x,y,z})) \leftarrow P_1(\widehat{x},\widehat{y},\widehat{z})$. $P_1(\widehat{h(x)},\widehat{i(y)},\widehat{s(z)}) \leftarrow P_a(\widehat{x}), P_a(\widehat{y}), P_a(\widehat{z})$.
$P_d(d_2(\widehat{x,y,z})) \leftarrow P_a(\widehat{x}), P_a(\widehat{y}), P_a(\widehat{z})$.	⟶
$P_d(d_1(h(x),\widehat{i(y)},s(z))) \leftarrow P_1(\widehat{x},\widehat{y},\widehat{z})$	$P_1(\widehat{h(x)},\widehat{i(y)},\widehat{s(z)}) \leftarrow P_1(\widehat{x},\widehat{y},\widehat{z})$.
$P_d(d_2(\widehat{x,y,z})) \leftarrow P_1(\widehat{x},\widehat{y},\widehat{z})$.	⟶
$P_d(c(\widehat{x,y})) \leftarrow P_a(\widehat{x}), P_a(\widehat{y})$.	⟶
$P_d(d_2(f(h(x)),\widehat{g(i(y))},z)) \leftarrow P_a(\widehat{x}), P_a(\widehat{y}), P_a(\widehat{z})$	$P_d(d_2(\widehat{x,y,z})) \leftarrow P_2(\widehat{x},\widehat{y},\widehat{z},x',y',z'), P_a(\widehat{x'}), P_a(\widehat{y'}), P_a(\widehat{z'})$. $P_2(\widehat{f(x)},\widehat{g(y)},\widehat{z},x',y',z') \leftarrow P_2(\widehat{x},\widehat{y},\widehat{z},h(x'),i(y'),z')$ $P_2(\widehat{x},\widehat{y},\widehat{z},x,y,z) \leftarrow .$
A cycle is detected – removeCycles replaces the blue clause by the red one.	$P_2(\widehat{f(x)},\widehat{g(y)},\widehat{z},x',y',z') \leftarrow P_2(\widehat{x},\widehat{y},\widehat{z_1},h(x'),i(y'),z'_1),$ $P_2(\widehat{x_1},\widehat{y_1},\widehat{z},h(x'_1),i(y'_1),z')$
$P_d(d_2(f(h(x)),\widehat{g(i(y))},z)) \leftarrow P_1(\widehat{x},\widehat{y},\widehat{z})$	$P_d(d_2(\widehat{x,y,z})) \leftarrow P_2(\widehat{x},\widehat{y},\widehat{z},x',y',z'), P_1(\widehat{x'},\widehat{y'},\widehat{z'})$.
$P_d(c(f(\widehat{x}),g(y))) \leftarrow P_2(\widehat{x},\widehat{y},\widehat{z},h(x'),i(y'),z'),$ $P_a(\widehat{x'}), P_a(\widehat{y'})$.	$P_3(\widehat{f(x)},\widehat{g(y)}) \leftarrow P_2(\widehat{x},\widehat{y},\widehat{z},h(x'),i(y'),z'), P_a(\widehat{x'}), P_a(\widehat{y'})$. $P_d(c(\widehat{x,y})) \leftarrow P_3(\widehat{x},\widehat{y})$.

Fig. 2. Run of comp_R on Example 27

On the other hand, to get rid of the left-linearity restriction, we are studying a technique based on the transformation of any Horn clause into CS-clauses [17]. However, the method of [17] does not always terminate. We want to ensure termination thanks to an additional over-approximation.

References

1. Boichut, Y., Boyer, B., Genet, T., Legay, A.: Equational Abstraction Refinement for Certified Tree Regular Model Checking. In: Aoki, T., Taguchi, K. (eds.) ICFEM 2012. LNCS, vol. 7635, pp. 299–315. Springer, Heidelberg (2012)
2. Boichut, Y., Chabin, J., Réty, P.: Over-approximating descendants by synchronized tree languages. RTA. LIPIcs **21**, 128–142 (2013)
3. Boichut, Y., Chabin, J., Réty, P.: Towards more precise rewriting approximations (full version). Tech. Rep. RR-2014-02, LIFO, Université d'Orléans (2014)
4. Boichut, Y., Courbis, R., Héam, P.-C., Kouchnarenko, O.: Finer Is Better: Abstraction Refinement for Rewriting Approximations. In: Voronkov, A. (ed.) RTA 2008. LNCS, vol. 5117, pp. 48–62. Springer, Heidelberg (2008)

5. Boichut, Y., Héam, P.C.: A Theoretical Limit for Safety Verification Techniques with Regular Fix-point Computations. IPL **108**(1), 1–2 (2008)
6. Bouajjani, A., Habermehl, P., Rogalewicz, A., Vojnar, T.: Abstract Regular (Tree) Model Checking. STTT **14**(2), 167–191 (2012)
7. Comon, H., Dauchet, M., Gilleron, R., Lugiez, D., Tison, S., Tommasi, M.: Tree Automata Techniques and Applications (TATA)
8. Durand, I., Sylvestre, M.: Left-linear bounded trss are inverse recognizability preserving. RTA. LIPIcs **10**, 361–376 (2011)
9. Engelfriet, J., Heyker, L.: Context-free Hypergraph Grammars have the same Term-generating Power as Attribute Grammars. Acta Informatica 29 (1992)
10. Engelfriet, J., Vereijken, J.: Context-free Grammars and Concatenation of Graphs. Acta Informatica **34**, 773–803 (1997)
11. Engelfriet, J., Schmidt, E.M.: IO and OI (I). Journal of Computer and System Sciences **15**(3), 328–353 (1977)
12. Engelfriet, J., Schmidt, E.M.: IO and OI (II). Journal of Computer and System Sciences **16**(1), 67–99 (1978)
13. Genet, T.: Decidable Approximations of Sets of Descendants and Sets of Normal Forms. In: Nipkow, T. (ed.) RTA 1998. LNCS, vol. 1379, pp. 151–165. Springer, Heidelberg (1998)
14. Genet, T., Klay, F.: Rewriting for cryptographic protocol verification. In: McAllester, D. (ed.) Automated Deduction - CADE-17. LNCS, vol. 1831, pp. 271–290. Springer, Heidelberg (2000)
15. Gouranton, V., Réty, P., Seidl, H.: Synchronized Tree Languages Revisited and New Applications. In: Honsell, F., Miculan, M. (eds.) FOSSACS 2001. LNCS, vol. 2030, pp. 214–229. Springer, Heidelberg (2001)
16. Kochems, J., Ong, C.H.L.: Improved Functional Flow and Reachability Analyses Using Indexed Linear Tree Grammars. RTA. LIPIcs **10**, 187–202 (2011)
17. Limet, S., Salzer, G.: Proving Properties of Term Rewrite Systems via Logic Programs. In: van Oostrom, V. (ed.) RTA 2004. LNCS, vol. 3091, pp. 170–184. Springer, Heidelberg (2004)
18. Limet, S., Salzer, G.: Tree Tuple Languages from the Logic Programming Point of View. Journal of Automated Reasoning **37**(4), 323–349 (2006)
19. Raoult, J.: Rational Tree Relations. Bulletin of the Belgian Mathematical Society Simon Stevin **4**, 149–176 (1997)
20. Réty, P.: Langages synchronisés d'arbres et applications. Habilitation Thesis (in French). LIFO, Université d'Orléans. Tech. rep., June 2001
21. Réty, P., Chabin, J., Chen, J.: R-Unification thanks to Synchronized-Contextfree Tree Languages. In: UNIF (2005)
22. Réty, P., Chabin, J., Chen, J.: Synchronized ContextFree Tree-tuple Languages. Tech. Rep. RR-2006-13 (LIFO, 2006)
23. Rounds, W.C.: Context-free grammars on trees. In: Fischer, P.C., Ginsburg, S., Harrison, M.A. (eds.) STOC. ACM (1969)