Finer is better: Abstraction Refinement for Rewriting Approximations*

Y. Boichut[1], R. Courbis[2] and P.-C. Héam[2], and O. Kouchnarenko[2]

INRIA/PAREO
615 rue du Jardin Botanique
BP-101
F-54602 Villers-Lès Nancy Cedex
boichut@loria.fr

INRIA/CASSIS
LIFC / University of Franche-Comté
16 route de Gray
F-25030 Besançon Cedex
lastname.firstname@lifc.univ-fcomte.fr

Abstract. Term rewriting systems are now commonly used as a modeling language for programs or systems. On those rewriting based models, reachability analysis, i.e. proving or disproving that a given term is reachable from a set of input terms, provides an efficient verification technique. For disproving reachability (i.e. proving non reachability of a term) on non terminating and non confluent rewriting models, Knuth-Bendix completion and other usual rewriting techniques do not apply. Using the tree automaton completion technique, it has been shown that the non reachability of a term t can be shown by computing an over-approximation of the set of reachable terms and prove that t is not in the over-approximation. However, when the term t is in the approximation, nothing can be said.
In this paper, we improve this approach as follows: given a term t, we try to compute an over-approximation which does not contain t by using an approximation refinement that we propose. If the approximation refinement fails then t is a reachable term. This semi-algorithm has been prototyped in the Timbuk tool. We present some experiments with this prototype showing the interest of such an approach w.r.t. verification on rewriting models.

1 Introduction

In the rewriting theory, the reachability problem is the following: given a term rewriting system (TRS) $\mathcal{R}$ and two terms s and t, can we decide whether $s \to_{\mathcal{R}}^{*} t$ or not? This problem, which can easily be solved on strongly terminating TRSs (by rewriting s into all its possible reduced forms and compare them to t), is undecidable on non terminating TRSs. There exists several syntactic classes of TRSs for which this problem becomes decidable: some are surveyed in [12], more recent ones are [16, 20]. In general, the decision procedures for those classes compute a finite tree automaton recognising the possibly infinite set of terms reachable from a set $E \subseteq \mathcal{T}(\mathcal{F})$ of initial terms, by $\mathcal{R}$, denoted by $\mathcal{R}^{*}(E)$. Then, provided that $s \in E$, those procedures check whether $t \in \mathcal{R}^{*}(E)$ or not. On

* This work has been funded by the French ANR-06-SETI-014 RAVAJ project.

the other hand, outside of those decidable classes, one can prove $s \not\rightarrow^*_{\mathcal{R}} t$ using over-approximations of $\mathcal{R}^*(E)$ [17, 12] and proving that t does not belong to this approximation.

Recently, reachability analysis turned out to be a very efficient verification technique for proving properties on infinite systems modeled by TRSs. Some of the most successful experiments, using proofs of $s \not\rightarrow^*_{\mathcal{R}} t$, were done on cryptographic protocols [19, 13, 4] where protocols and intruders are described using a TRS $\mathcal{R}$, E represents the set of initial configurations of the protocol and t a possible flaw. Some other have been carried out on Java byte code programs [2] and in this context, $\mathcal{R}$ encodes the byte code instructions and the evolution of the Java Virtual Machine (JVM), E specifies the set of initial configurations of the JVM and t a possible flaw.

Then reachability analysis can prove the absence of flaws (if $\forall s \in E : s \not\rightarrow^*_{\mathcal{R}} t$). In [12], the method we propose to improve, given a TRS $\mathcal{R}$, a set of terms E and an abstraction function γ, a sequence of sets of terms $App^\gamma_0, App^\gamma_1, \ldots, App^\gamma_k$ is built such that $App^\gamma_0 = E$ and $\mathcal{R}(App^\gamma_i) \subseteq App^\gamma_{i+1}$. The role of the abstraction γ is to define equivalence classes of terms and to allot each term to an equivalence class. The computation stops when on the one hand, the number of equivalence classes introduced by the abstraction function is bounded, and on the other hand, each equivalence class is $\mathcal{R}-$closed, i.e. when there exists $N \in \mathbb{N}$ such that $\mathcal{R}(App^\gamma_N) = App^\gamma_N$. Then, App^γ_N represents an over-approximation of terms reachable by $\mathcal{R}$ from E. The abstraction function γ should be well designed in such a way that on one hand App^γ_N exists, and on the other hand $t \notin App^\gamma_N$. However, the main drawback of this technique based on tree automata, is that if $t \notin \mathcal{R}^*(E)$ then it is not trivial (when it is possible) to compute a such fix-point over-approximation App^γ_N. Indeed, a high-level expertise in this technique is required for defining a pertinent abstraction function. At the same time, it is easy to define simple abstraction functions leading to inconclusive analyses. So, the question is : *Is it possible to obtain conclusive analyses starting from simple abstraction functions?* This problem becomes crucial when approximations are used to prove security and safety properties and when a large community of users is targeted.

This paper addresses this question and proposes a solution that automatically attempts to show that a term t is not a term of $\mathcal{R}^*(E)$. We proceed as follows. For a simple abstraction function γ, we compute a sequence $App^\gamma_1, \ldots, App^\gamma_k$ such that: either App^γ_k is a fix-point automaton whose language is an over-approximation of reachable terms and $t \notin App^\gamma_k$, or App^γ_k recognises t. For the former, everything is fine and we are done. For the latter, we first detect in the sequence $App^\gamma_1, \ldots, App^\gamma_k$ where the abstraction function has been too coarse. Second, we automatically refine γ, i.e., we fix γ in order to remove t from the over-approximation. The construction of the sequence restarts from the problematic set App^γ_i with the refined abstraction function and so on. Moreover, if the algorithm fails for finding the reason concerning the abstraction function which makes t be in App^γ_k then the term t is a term in $\mathcal{R}^*(E)$.

Note that this solution is a semi-algorithm and it has been prototyped in the Timbuk tool [14]. For a lack of space, the proofs of this paper are available at `http://lifc.univ-fcomte.fr/~kouchna/Kouchnarenko-english.html`.

Layout of the paper The paper is organised as follows. After giving preliminary notions on tree-automata and TRSs, we introduce in Section 2 the completion technique we want to improve. Section 3 presents the main contributions concerning the refinement of abstraction functions, the backward completion for the computation of the ancestors of a set of terms by rewriting. A semi-algorithm including those both processes is also given. Finally, before concluding, Section 4 reports on experimental results showing the feasibility and the interest of the proposed approach.

2 Preliminaries

2.1 Terms and TRSs

Comprehensive surveys can be found in [11, 1] for term rewriting systems, and in [9, 15] for tree automata and tree language theory.

Let $\mathcal{F}$ be a finite set of symbols, associated with an arity function $ar : \mathcal{F} \to \mathbb{N}$, and let $\mathcal{X}$ be a countable set of variables. $\mathcal{T}(\mathcal{F}, \mathcal{X})$ denotes the set of terms, and $\mathcal{T}(\mathcal{F})$ denotes the set of ground terms (terms without variables). The set of variables of a term t is denoted by $\mathcal{V}ar(t)$. A substitution is a function σ from $\mathcal{X}$ into $\mathcal{T}(\mathcal{F}, \mathcal{X})$, which can be extended uniquely to an endomorphism of $\mathcal{T}(\mathcal{F}, \mathcal{X})$. A position p for a term t is a word over $\mathbb{N}$. The empty sequence ϵ denotes the top-most position. The set $\mathcal{P}os(t)$ of positions of a term t is inductively defined by $\mathcal{P}os(t) = \{\epsilon\}$ if $t \in \mathcal{X}$ and by $\mathcal{P}os(f(t_1, \ldots, t_n)) = \{\epsilon\} \cup \{i.p \mid 1 \leq i \leq n \text{ and } p \in \mathcal{P}os(t_i)\}$ otherwise. If $p \in \mathcal{P}os(t)$, then $t|_p$ denotes the subterm of t at position p and $t[s]_p$ denotes the term obtained by replacement of the subterm $t|_p$ at position p by the term s. We also denote by $t(p)$ the symbol occurring in t at position p. Given a term $t \in \mathcal{T}(\mathcal{F}, \mathcal{X})$, we denote $\mathcal{P}os_A(t) \subseteq \mathcal{P}os(t)$ the set of positions of t such that $\mathcal{P}os_A(t) = \{p \in \mathcal{P}os(t) \mid t(p) \in A\}$. Thus $\mathcal{P}os_{\mathcal{F}}(t)$ is the set of functional positions of t. A TRS $\mathcal{R}$ is a set of *rewrite rules* $l \to r$, where $l, r \in \mathcal{T}(\mathcal{F}, \mathcal{X})$ and $l \notin \mathcal{X}$. A rewrite rule $l \to r$ is *left-linear* (resp. right-linear) if each variable of l (resp. r) occurs only once within l (resp. r). A TRS $\mathcal{R}$ is left-linear (resp. right-linear) if every rewrite rule $l \to r$ of $\mathcal{R}$ is left-linear (resp. right-linear). A TRS $\mathcal{R}$ is linear if it is right and left-linear. The TRS $\mathcal{R}$ induces a rewriting relation $\to_{\mathcal{R}}$ on terms whose reflexive transitive closure is written $\to^\star_{\mathcal{R}}$. The set of $\mathcal{R}$-descendants of a set of ground terms E is $\mathcal{R}^*(E) = \{t \in \mathcal{T}(\mathcal{F}) \mid \exists s \in E \text{ s.t. } s \to^\star_{\mathcal{R}} t\}$. Symmetrically, the set of $\mathcal{R}$-ancestors of a set of ground terms E is $\mathcal{R}^{-1\star}(E) = \{s \in \mathcal{T}(\mathcal{F}) \mid \exists t \in E \text{ s.t. } s \to^\star_{\mathcal{R}} t\}$.

2.2 Tree Automata Completion

Note that $\mathcal{R}^*(E)$ is possibly infinite: $\mathcal{R}$ may not terminate and/or E may be infinite. The set $\mathcal{R}^*(E)$ is generally not computable [15]. However, it is possible

to over-approximate it [12] using tree automata, i.e. a finite representation of infinite (regular) sets of terms. We next define tree automata.

Let $\mathcal{Q}$ be a finite set of symbols, of arity 0, called *states* such that $\mathcal{Q} \cap \mathcal{F} = \emptyset$. $\mathcal{T}(\mathcal{F} \cup \mathcal{Q})$ is called the set of *configurations*.

Definition 1 (Transition and normalised transition). *A* transition *is a rewrite rule $c \to q$, where $c \in \mathcal{T}(\mathcal{F} \cup \mathcal{Q})$ is a configuration and $q \in \mathcal{Q}$. A* normalised transition *is a transition $c \to q$ where $c = f(q_1, \ldots, q_n)$, $f \in \mathcal{F}$, $ar(f) = n$, and $q_1, \ldots, q_n \in \mathcal{Q}$.*

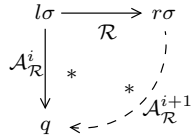
Definition 2 (Bottom-up non-deterministic finite tree automaton). *A bottom-up non-deterministic finite tree automaton (tree automaton for short) is a quadruple $\mathcal{A} = \langle \mathcal{F}, \mathcal{Q}, \mathcal{Q}_f, \Delta \rangle$, $\mathcal{Q}_f \subseteq \mathcal{Q}$ and Δ is a finite set of normalised transitions.*

The *rewriting relation* on $\mathcal{T}(\mathcal{F} \cup \mathcal{Q})$ induced by the transition set Δ of $\mathcal{A}$ is denoted $\to_\Delta$. When Δ is clear from the context, $\to_\Delta$ is also written $\to_\mathcal{A}$.

Definition 3 (Recognised language). *The tree language recognised by $\mathcal{A}$ in a state q is $\mathcal{L}(\mathcal{A}, q) = \{t \in \mathcal{T}(\mathcal{F}) \mid t \to_\mathcal{A}^\star q\}$. The language recognised by $\mathcal{A}$ is $\mathcal{L}(\mathcal{A}) = \bigcup_{q \in \mathcal{Q}_f} \mathcal{L}(\mathcal{A}, q)$. A tree language is regular if and only if it is recognised by a tree automaton.*

Let us now recall how tree automata and TRSs can be used for term reachability analysis. Given a tree automaton $\mathcal{A}$ and a TRS $\mathcal{R}$, the tree automata completion algorithm proposed in [12] computes a tree automaton $\mathcal{A}^k_\mathcal{R}$ such that $\mathcal{L}(\mathcal{A}^k_\mathcal{R}) = \mathcal{R}^*(\mathcal{L}(\mathcal{A}))$ when it is possible (for the classes of TRSs where an exact computation is possible, see [12]), and such that $\mathcal{L}(\mathcal{A}^k_\mathcal{R}) \supseteq \mathcal{R}^*(\mathcal{L}(\mathcal{A}))$ otherwise.

The tree automata completion works as follows. From $\mathcal{A} = \mathcal{A}^0_\mathcal{R}$ the completion builds a sequence $\mathcal{A}^0_\mathcal{R}, \mathcal{A}^1_\mathcal{R} \ldots \mathcal{A}^k_\mathcal{R}$ of automata such that if $s \in \mathcal{L}(\mathcal{A}^i_\mathcal{R})$ and $s \to_\mathcal{R} t$ then $t \in \mathcal{L}(\mathcal{A}^{i+1}_\mathcal{R})$. If there is a fix-point automaton $\mathcal{A}^k_\mathcal{R}$ such that $\mathcal{R}^*(\mathcal{L}(\mathcal{A}^k_\mathcal{R})) = \mathcal{L}(\mathcal{A}^k_\mathcal{R})$, then $\mathcal{L}(\mathcal{A}^k_\mathcal{R}) = \mathcal{R}^*(\mathcal{L}(\mathcal{A}^0_\mathcal{R}))$ (or $\mathcal{L}(\mathcal{A}^k_\mathcal{R}) \supseteq \mathcal{R}^*(\mathcal{L}(\mathcal{A}))$ if $\mathcal{R}$ is in no class of [12]). To build $\mathcal{A}^{i+1}_\mathcal{R}$ from $\mathcal{A}^i_\mathcal{R}$, a *completion step* is achieved. It consists of finding *critical pairs* between $\to_\mathcal{R}$ and $\to_{\mathcal{A}^i_\mathcal{R}}$. To define the notion of critical pair, the substitution definition is extended to terms in $\mathcal{T}(\mathcal{F} \cup \mathcal{Q})$. For a substitution $\sigma : \mathcal{X} \mapsto \mathcal{Q}$ and a rule $l \to r \in \mathcal{R}$ such that $\mathcal{V}ar(r) \subseteq \mathcal{V}ar(l)$, if there exists $q \in \mathcal{Q}$ satisfying $l\sigma \to^*_{\mathcal{A}^i_\mathcal{R}} q$ then $l\sigma \to^*_{\mathcal{A}^i_\mathcal{R}} q$ and $l\sigma \to_\mathcal{R} r\sigma$ is a critical pair. Note that since $\mathcal{R}$ and $\mathcal{A}^i_\mathcal{R}$ are finite, there is only a finite number of critical pairs. Thus, for every critical pair detected between $\mathcal{R}$ and $\mathcal{A}^i_\mathcal{R}$ such that $r\sigma \not\to^*_{\mathcal{A}^i_\mathcal{R}} q$, the tree automaton $\mathcal{A}^{i+1}_\mathcal{R}$ is constructed by adding a new transition $r\sigma \to q$ to $\mathcal{A}^i_\mathcal{R}$. Consequently, $\mathcal{A}^{i+1}_\mathcal{R}$ recognises $r\sigma$ in q, i.e. $r\sigma \to_{\mathcal{A}^{i+1}_\mathcal{R}} q$.



However, the transition $r\sigma \to q$ is not necessarily normalised. Then, we use abstraction functions whose goal is to define a set of normalised transitions

$Norm$ such that $r\sigma \rightarrow^*_{Norm} q$. Thus, instead of adding the transition $r\sigma \rightarrow q$ which is not normalised, the set of transitions $Norm$ is added to Δ, i.e., the transition set of the current automaton $\mathcal{A}^i_{\mathcal{R}}$. Notice that the completion process introduces new states. We give below a very general definition of abstraction functions which allot a state in $\mathcal{Q}$ to each functional position of $r\sigma$. The role of an abstraction function is to define equivalence classes of terms where one class corresponds to one state in $\mathcal{Q}$.

Definition 4 (Abstraction Function). *An abstraction function γ is a function $\gamma : ((\mathcal{R} \times (\mathcal{X} \rightarrow \mathcal{Q}) \times \mathcal{Q}) \mapsto \mathbb{N}^*) \mapsto \mathcal{Q}$ such that $\gamma(l \rightarrow r, \sigma, q)(\epsilon) = q$.*

Thus, given an abstraction function γ, the normalisation of a transition $r\sigma \rightarrow q$ is defined as follows.

Definition 5 ($\gamma-$normalisation). *Let γ be an abstraction function, Δ be a transition set, $l \rightarrow r \in R$ with $\mathcal{V}ar(r) \subseteq \mathcal{V}ar(l)$ and $\sigma : \mathcal{X} \rightarrow \mathcal{Q}$ such that $l\sigma \rightarrow^*_{\Delta} q$. The $\gamma-$normalisation of the transition $r\sigma \rightarrow q$, written $Norm_\gamma(l \rightarrow r, \sigma, q)$, is defined by:*

$$
\begin{aligned}
Norm_\gamma(l \rightarrow r, \sigma, q) = \{ & r(p)(\beta_{p.1}, \ldots, \beta_{p.n}) \rightarrow \beta \mid \\
& p \in \mathcal{P}os_{\mathcal{F}}(r), \\
& \beta = \gamma(l \rightarrow r, \sigma, q)(p) \\
& \beta_{p.i} = \begin{cases} \sigma(r(p.i)) \ if \ r(p.i) \in \mathcal{X} \\ \gamma(l \rightarrow r, \sigma, q)(p.i) \ otherwise. \end{cases}
\end{aligned}
$$

Example 1 (Normalisation of a transition using an abstraction function).
Let $\mathcal{A} = \langle \mathcal{F}, \mathcal{Q}, \mathcal{Q}_f, \Delta \rangle$ be the tree automaton such that $\mathcal{F} = \{a, b, c, d, e, f, \omega\}$ with $ar(s) = 1$ with $s \in \{a, b, c, d, e, f\}$ and $ar(\omega) = 0$, $\mathcal{Q} = \{q_b, q_f, q_\omega\}$, $\mathcal{Q}_f = \{q_f\}$ and $\Delta = \{\omega \rightarrow q_\omega, b(q_\omega) \rightarrow q_b, a(q_b) \rightarrow q_f\}$. Thus, $\mathcal{L}(\mathcal{A}) = \{a(b(\omega))\}$. Given the TRS $\mathcal{R} = \{a(x) \rightarrow c(d(x)),\ b(x) \rightarrow e(f(x))\}$, two critical pairs are computed: $a(q_b) \rightarrow^*_{\mathcal{A}} q_f$, $a(q_b) \rightarrow_{\mathcal{R}} c(d(q_b))$ and $b(q_\omega) \rightarrow^*_{\mathcal{A}} b(q_\omega) \rightarrow_{\mathcal{R}} e(f(q_\omega))$. Let γ be the abstraction function such that $\gamma(a(x) \rightarrow c(d(x)), \{x \rightarrow q_b\}, q_f)(\epsilon) = q_f$, $\gamma(a(x) \rightarrow c(d(x)), \{x \rightarrow q_b\}, q_f)(1) = q_f$, $\gamma(b(x) \rightarrow e(f(x)), \{x \rightarrow q_\omega\}, q_b)(\epsilon) = q_b$ and $\gamma(b(x) \rightarrow e(f(x)), \{x \rightarrow q_\omega\}, q_b)(1) = q_b$.

So, $Norm_\gamma(a(x) \rightarrow c(d(x)), \{x \rightarrow q_b\}, q_f) = \{d(q_b) \rightarrow q_f, c(q_f) \rightarrow q_f\}$ and $Norm_\gamma(b(x) \rightarrow e(f(x)), \{x \rightarrow q_\omega\}, q_b) = \{f(q_\omega) \rightarrow q_b, e(q_b) \rightarrow q_b\}$.

Now we formally define what a completion step is.

Definition 6 (One Completion Step). *Let $\mathcal{A} = \langle \mathcal{F}, \mathcal{Q}, \mathcal{Q}_f, \Delta \rangle$ be a tree automaton, γ an abstraction function and $\mathcal{R}$ a left-linear TRS. We define a tree automaton $C^{\mathcal{R}}_\gamma(\mathcal{A}) = \langle \mathcal{F}, \mathcal{Q}', \mathcal{Q}'_f, \Delta' \rangle$ with $\mathcal{Q}' = \{q \mid c \rightarrow q \in \Delta'\}$ (and $\mathcal{Q} \subseteq \mathcal{Q}'$), $\mathcal{Q}'_f = \mathcal{Q}_f$ and $\Delta' = \Delta \ \cup \bigcup_{l \rightarrow r \in \mathcal{R},\ \sigma : \mathcal{X} \mapsto \mathcal{Q},\ l\sigma \rightarrow^*_{\mathcal{A}} q, r\sigma \not\rightarrow^*_{\mathcal{A}} q} Norm_\gamma(l \rightarrow r, \sigma, q)$.*

Example 2. Given $\mathcal{A}$, $\mathcal{R}$ and γ of Example 1, performing one completion step on $\mathcal{A}$ gives the automaton $C^{\mathcal{R}}_\gamma(\mathcal{A})$ such that $C^{\mathcal{R}}_\gamma(\mathcal{A}) = \langle \mathcal{F}, \mathcal{Q}, \mathcal{Q}_f, \Delta' \rangle$ where $\Delta' = \Delta \cup Norm_\gamma(a(x) \rightarrow c(d(x)), \{x \rightarrow q_b\}, q_f) \cup Norm_\gamma(b(x) \rightarrow e(f(x)), \{x \rightarrow q_\omega\}, q_b) = \{\omega \rightarrow q_\omega, b(q_\omega) \rightarrow q_b, a(q_b) \rightarrow q_f, d(q_b) \rightarrow q_f, c(q_f) \rightarrow q_f, f(q_\omega) \rightarrow$

$q_b, e(q_b) \to q_b\}$. Notice that $C^{\mathcal{R}}_\gamma(\mathcal{A})$ is $\mathcal{R}$-close, and in fact an over-approximation of $\mathcal{R}^*(\mathcal{L}(\mathcal{A}))$ is computed. Indeed, the tree automaton $C^{\mathcal{R}}_\gamma(\mathcal{A})$ recognises the term $a(e(e(f(\omega))))$ when $\mathcal{R}^*(\mathcal{L}(\mathcal{A})) = \{a(b(\omega)), a(e(f(\omega))), c(d(b(\omega))), c(d(e(f(\omega))))\}$.

Proposition 1 (Adaptation of [12, Theorem 1]). *Let $\mathcal{A}$ be a tree automaton and $\mathcal{R}$ be a TRS such that $\mathcal{A}$ is deterministic or $\mathcal{R}$ is left-linear, and for every $l \to r \in \mathcal{R}$, $\mathcal{V}ar(r) \subseteq \mathcal{V}ar(l)$. One has $\mathcal{L}(\mathcal{A}) \cup \mathcal{R}(\mathcal{L}(\mathcal{A})) \subseteq C^{\mathcal{R}}_\gamma(\mathcal{A})$, for any abstraction function γ.*

However, abstraction functions can be defined in such a way that only actually reachable terms are computed. We call this class of abstraction functions $(\mathcal{A}, \mathcal{R})-$exact abstraction functions.

Definition 7 ($(\mathcal{A}, \mathcal{R})-$exact abstraction function). *Let $\mathcal{A} = \langle \mathcal{F}, \mathcal{Q}, \mathcal{Q}_f, \Delta \rangle$ be a tree automaton and $\mathcal{R}$ be a TRS. Let $Im(\gamma) = \{q' \mid \exists l \to r \in \mathcal{R},\ \exists p \in \mathcal{P}os_{\mathcal{F}}(r),\ p \neq \epsilon,\ \exists \sigma : \mathcal{X} \to \mathcal{Q}\ .\ \gamma(l \to r, \sigma, q)(p) = q'\}$. An abstraction function γ is $(\mathcal{A}, \mathcal{R})-$exact if γ is injective and $Im(\gamma) \cap \mathcal{Q} = \emptyset$.*

By adapting the proof of Theorem 2 in [12] to the new class of abstractions, we show that with such abstraction functions, only reachable terms are computed.

Proposition 2 ([12, Theorem 2]). *Let $\mathcal{A}$ be a tree automaton and $\mathcal{R}$ be a left-linear TRS such that $\mathcal{A}$ is deterministic or $\mathcal{R}$ is also right-linear. Let α be an $(\mathcal{A}, \mathcal{R})-$exact abstraction function. One has: $C^{\mathcal{R}}_\alpha(\mathcal{A}) \subseteq \mathcal{R}^*(\mathcal{L}(\mathcal{A}))$.*

Example 3 (Exact automaton with $(\mathcal{A}, \mathcal{R})-$exact abstraction functions).
Let $\mathcal{A}$, $\mathcal{R}$ be the tree automaton and the TRS from Example 1. Let α be the $(\mathcal{A}, \mathcal{R})-$abstraction function such that $Norm_\alpha(a(x) \to c(d(x)), \{x \to q_b\}, q_f) = \{c(q_1) \to q_f,\ d(q_b) \to q_1\}$ and $Norm_\alpha(b(x) \to e(f(x)), \{x \to q_\omega\}, q_b) = \{e(q_2) \to q_b, f(q_\omega) \to q_2\}$. Note that q_1 and q_2 are not states of $\mathcal{A}$. Then, $C^{\mathcal{R}}_\alpha(\mathcal{A})$ is the tree automaton $\langle \mathcal{F}, \mathcal{Q} \cup \{q_1, q_2\}, \mathcal{Q}_f, \Delta' \rangle$ where $\Delta' = \{\omega \to q_\omega, b(q_\omega) \to q_b, a(q_b) \to q_f, d(q_b) \to q_1, c(q_1) \to q_f, f(q_\omega) \to q_2, e(q_2) \to q_b\}$.

Figure 1 gives a graphical representation of Example 3 and Example 2 using word automata. Indeed, considering the symbol ω as the empty word, the term $a(b(\omega))$ can be read as the word ab. The state q_f of $\mathcal{A}$ becomes the initial state of the word automaton, and q_ω is its final state. Note that we do not consider the empty word ω in our representation. So, $C^{\mathcal{R}}_\gamma(\mathcal{A})$ from Example 2 is represented by the word automaton without non dashed transitions implying q_1 and q_2. It recognises, for example, the word $aeef$ because of the abstraction. Using $(\mathcal{A}, \mathcal{R})-$exact approximation function α gives $C^{\mathcal{R}}_\alpha(\mathcal{A})$, the word automaton in Fig. 1 without dashed transitions.

We now give the general result in [12] saying that, if there exists a fix-point automaton, then its language contains all the terms actually reachable by rewriting, at least.

Theorem 1 ([12, Theorem 1]). *Let $\mathcal{A}$ and $\mathcal{R}$ be respectively a tree automaton and a left-linear TRS. For any abstraction function, if there exists $N \in \mathbb{N}$ and $N \geq 0$ such that $(C^{\mathcal{R}}_\gamma)^{(N)}(\mathcal{A}) = (C^{\mathcal{R}}_\gamma)^{(N+1)}(\mathcal{A})$, then $\mathcal{R}^*(\mathcal{L}(\mathcal{A})) \subseteq \mathcal{L}((C^{\mathcal{R}}_\gamma)^{(N)}(\mathcal{A}))$.*

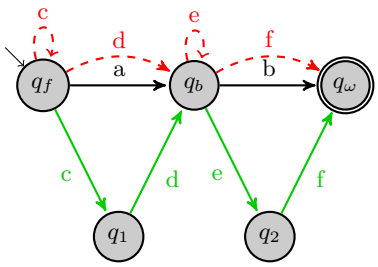


Fig. 1. $C_\gamma^{\mathcal{R}}(\mathcal{A})$ and $C_\alpha^{\mathcal{R}}(\mathcal{A})$ as simple word automata

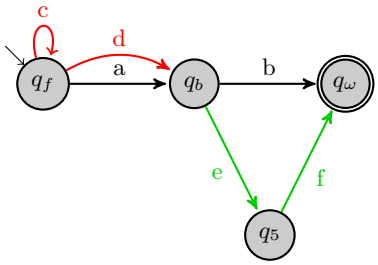


Fig. 2. Word automaton for $C_{\gamma_{\text{Disc}}}^{\mathcal{R}}$

3 Approximation Refinement

Let consider a tree automaton $\mathcal{A}$, a TRS $\mathcal{R}$ and an abstraction function γ such that there exists $N \in \mathbb{N}$ for which $(C_\gamma^{\mathcal{R}})^{(N)}(\mathcal{A}) = (C_\gamma^{\mathcal{R}})^{(N+1)}(\mathcal{A})$. Let $\mathcal{A}_p$ be a tree automaton recognising a set of unwanted terms, i.e., we want that terms to be unreachable. Let suppose that the intersection between the languages of $(C_\gamma^{\mathcal{R}})^{(N)}(\mathcal{A})$ and $\mathcal{A}_p$ is not empty; the method in [12] cannot conclude. In this section we refine γ to show that: either a term recognised by $\mathcal{A}_p$ is actually reachable by rewriting from a term in $\mathcal{L}(\mathcal{A})$, either all terms recognised by $\mathcal{A}_p$ are actually unreachable. For the former, we present in Section 3.1 how to refine an abstraction function when that function gives rise to the non-empty intersection. For the latter, in Section 3.2, we describe the computation of the ancestors of a set of terms by rewriting, using a completion on TRSs whose rewrite rules are reversed. Then, in Section 3.3, the backward analysis is used for refining abstraction functions when the assumptions in Section 3.1 are not satisfied.

3.1 Abstraction Refinement

In this section, $\mathcal{A} = \langle \mathcal{F}, \mathcal{Q}, \mathcal{Q}_f, \Delta \rangle$ and $\mathcal{A}_P$ are two tree automata, $\mathcal{R}$ is a TRS, γ is an abstraction function and α is a $(\mathcal{A}, \mathcal{R})-$exact abstraction function. The tree automaton $\mathcal{A}_p$ recognises a set of forbidden terms. We assume that: (1) $\mathcal{L}(C_\gamma^{\mathcal{R}}(\mathcal{A})) \cap \mathcal{L}(\mathcal{A}_p) \neq \emptyset$, $\mathcal{L}(\mathcal{A}) \cap \mathcal{L}(\mathcal{A}_p) = \emptyset$ and (2) $\mathcal{L}(C_\alpha^{\mathcal{R}}(\mathcal{A})) \cap \mathcal{L}(\mathcal{A}_p) = \emptyset$. In other words, these assumptions mean that the abstraction function is too coarse since a term of $\mathcal{L}(\mathcal{A}_p)$ is reachable using γ, while it is not with an exact computation. So, it means that some transitions introduced by γ are problematic, i.e., lead to the non-empty intersection. The following definition allows the refinement of a given abstraction function according to a given set of problematic transitions.

Definition 8 (Refined abstraction function). *Let $l\sigma \to_{\mathcal{R}} r\sigma$, $l\sigma \to_{\mathcal{A}}^* q$ be a critical pair where $q \in \mathcal{Q}$, $l \to r$ is a rewrite rule and σ a substitution from $\mathcal{X}$ into $\mathcal{Q}$. Let Δ_0 be a set of transitions. For any functional position p of r, the*

refined abstraction function γ_{Δ_0} is built as follows:

$$\gamma_{\Delta_0}(l \to r, \sigma, q)(p) \stackrel{\text{def}}{=} \begin{cases} \alpha(l \to r, \sigma, q)(p) & if\ Norm_\gamma(l \to r, \sigma, q) \cap \Delta_0 \neq \emptyset \\ \gamma(l \to r, \sigma, q)(p) & otherwise. \end{cases}$$

For the abstraction function γ (resp. α), we denote by Δ_γ (resp. Δ_α) the set of transitions occurring in $C^{\mathcal{R}}_\gamma(\mathcal{A})$ (resp.$C^{\mathcal{R}}_\alpha(\mathcal{A})$) but not in $\mathcal{A}$, and by Q_γ (resp. Q_α) the set of states occurring in the transitions of Δ_γ (resp. Δ_α). The following proposition claims that, given γ and $\mathcal{A}$, we are able to refine γ in such a way that no unwanted term is recognised by the tree automaton resulting from one completion step on $\mathcal{A}$ when using $\mathcal{R}$ and the refined abstraction function.

Proposition 3 (Refined automaton existence). *Considering assumptions (1) and (2), there exists* $\text{Disc} \subseteq \Delta_\gamma$ *such that* $\mathcal{L}(C^{\mathcal{R}}_{\gamma_{\text{Disc}}}(\mathcal{A})) \cap \mathcal{L}(\mathcal{A}_P) = \emptyset$.

Example 4 (Refined Automaton). Let $\mathcal{A}$, γ and α be respectively the tree automaton, the two abstraction functions defined in Example 3. In Sect. 2.2, the word $aeef$ is recognised by the word automaton representing $C^{\mathcal{R}}_\gamma(\mathcal{A})$. Let $\text{Disc} \subseteq \Delta_\gamma$ be a set of transitions such that $\text{Disc} = \{e(q_b) \to q_b\}$. Indeed, this transition gives rise to infinite terms of the form $e^*(b(\omega))$ or $e^*(f(\omega))$ (infinite words of the form e^*b or e^*f). So, γ_{Disc} uses α and γ for normalising respectively the transitions $e(f(q_\omega)) \to q_b$ and $c(d(q_b)) \to q_f$. A word automaton representing $C^{\mathcal{R}}_{\gamma_{\text{Disc}}}$ is given in Fig 2. Note that the word $aeef$ is not recognised anymore.

For applying Proposition 3 in practice, it could be more efficient to fix only problematic transitions instead of performing an exact completion step by taking $\text{Disc} = \Delta_\gamma$.

3.2 Backward Reachability Analysis by Completion

The backward analysis we expose in this section can be viewed as an exact completion performed on an automaton $\mathcal{A}$, using a TRS $\mathcal{R}$ whose rules have been reversed and an $(\mathcal{A}, \mathcal{R})$-exact abstraction function. Let $\mathcal{R}^t$ be a TRS built from $\mathcal{R}$ in the following way: $\mathcal{R}^t = \{r \to l | l \to r \in \mathcal{R}\}$. So, for $\mathcal{R}^t$, some rules may not satisfy the conditions of Propositions 1 and 2, in particular that for each $l \to r \in \mathcal{R}^t$, $\mathcal{V}ar(r) \subseteq \mathcal{V}ar(l)$. This is why we extend the completion definition in Section 2.2. Before, we establish the relation between $\mathcal{R}^t(E)$ and $\mathcal{R}^{-1}(E)$ for a given set of terms $E \subseteq \mathcal{T}(\mathcal{F})$.

Proposition 4. *For every set of terms* $E \subseteq \mathcal{T}(\mathcal{F})$, $\mathcal{R}^t(E) = \mathcal{R}^{-1}(E)$.

A consequence of this proposition is that $(\mathcal{R}^t)^*(E) = (\mathcal{R}^{-1})^*(E)$. So, we can now define the completion algorithm in order to analyse $(\mathcal{R}^{-1})^*(E)$. For a given set of functional symbols $\mathcal{F}$, we introduce a set of transitions $T(q)$ reducing each term of $\mathcal{T}(\mathcal{F})$ to the single state q. $T(q)$ is built as follows: $T(q) = \{s \to q \mid s \in T(\mathcal{F}), s = f(q_1, \ldots, q_n), f \in \mathcal{F},\ ar(f) = n,\ n \geq 0,\ and\ q_i = q\ for\ i = 1 \ldots n\}$. This set is useful for handling variables occurring in the right-hand side of a rule but not in its left-hand side.

In the following definitions, a substitution is transformed according to a given set of variables and a given state.

Definition 9. *Let q be a state and $Y \subseteq \mathcal{X}$. Given a substitution $\sigma : \mathcal{X} \to \mathcal{Q}$ and $x \in \mathcal{X}$,*

$$Chg_Y^q(\sigma)(x) = \begin{cases} \sigma(x) & \text{if } x \in Y \\ q & \text{otherwise.} \end{cases}$$

In the remainder of this section we do not assume anymore that for each rule $l \to r \in \mathcal{R}$, $\mathcal{V}ar(r) \subseteq \mathcal{V}ar(l)$. Using Definition 9, Definition 10 extends the completion algorithm for handling such TRSs.

Definition 10 (Backward Completion). *Let $\mathcal{A} = \langle \mathcal{F}, \mathcal{Q}, \mathcal{Q}_f, \Delta \rangle$ be a tree automaton, γ an abstraction function and q_{all} a state such that $q_{all} \notin \mathcal{Q}$. Let $\mathcal{R}$ be a left-linear TRS. We define a tree automaton $K_\alpha^{\mathcal{R}}(\mathcal{A}) = \langle \mathcal{F}, \mathcal{Q}', \mathcal{Q}'_f, \Delta' \rangle$ where:*

$$\Delta' = \Delta \cup \bigcup_{l \to r \in \mathcal{R}, l\sigma \to_{\mathcal{A}}^* q, r(Chg_{\mathcal{V}ar(l)}^{q_{all}}(\sigma)) \not\to_{\mathcal{A}}^* q} (Norm_\gamma(l \to r, Chg_{\mathcal{V}ar(l)}^{q_{all}}(\sigma), q)) \cup T(q_{all}),$$

$\mathcal{Q}' = \{q \mid c \to q \in \Delta'\}$ *and* $\mathcal{Q}'_f = \mathcal{Q}_f$.

Note that there is a particular processing for variables occurring in the right-hand side of a rule which do not appear in its left-hand side. Indeed, each of these variables is substituted by the special state q_{all} because it is impossible to determine terms substituting these variables. Actually, considering the set $T(q_{all})$ of transitions, q_{all} is such that for each term $t \in \mathcal{T}(\mathcal{F})$, $t \to_{T(q_{all})}^* q_{all}$. So, for the rule $f(x) \to g(x, y)$, y is replaced by q_{all}, i.e., $f(x) \to g(x, q_{all})$. Roughly, we can say that a term of the form $f(x)$ can be rewritten into $g(x, t)$ where t is any term of $\mathcal{T}(\mathcal{F})$.

Finally, using an $(\mathcal{A}, \mathcal{R})-$exact abstraction function, we can perform a backward analysis of the set of reachable terms thanks to the new backward completion given above.

Proposition 5 (Extension of Propositions 1 and 2). *Let $\mathcal{A}$ be a tree automaton, $\mathcal{R}$ be TRS and α be an abstraction function. Thus, one has:*

1) *if $\mathcal{A}$ is deterministic or $\mathcal{R}$ is left-linear: $\mathcal{L}(\mathcal{A}) \cup \mathcal{R}(\mathcal{L}(\mathcal{A})) \subseteq \mathcal{L}(K_\alpha^{\mathcal{R}}(\mathcal{A}))$;*
2) *if $\mathcal{R}$ is linear and α be an $(\mathcal{A}, \mathcal{R})-$exact abstraction function: $\mathcal{L}(K_\alpha^{\mathcal{R}}(\mathcal{A})) \subseteq \mathcal{R}^*(\mathcal{L}(\mathcal{A}))$.*

3.3 Semi-Algorithm

In Sect. 2.2, reachability analysis can be performed by computing either an over-approximation of reachable terms with a fine tuned abstraction function or an under-approximation using an $(\mathcal{A}, \mathcal{R})-$exact abstraction function. The former may allow the proof of unreachability of terms and the latter may show that terms are reachable. Nevertheless, using the completion algorithm of [12], a choice must be done according to the kind of analysis we want to perform.

We propose in this section a new semi-algorithm which attempts to perform both analyses automatically using abstraction refinement. More precisely, let $\mathcal{A}$,

$\mathcal{R}$, γ, α and $\mathcal{A}_p$ be respectively a tree automaton, a linear TRS, an abstraction function, an $(\mathcal{A},\mathcal{R})-$exact abstraction function and a tree automaton recognising a set of unwanted terms. A sequence of automaton $\mathcal{A}_0,\ldots,\mathcal{A}_k$, where $\mathcal{A}_0=\mathcal{A}$, is computed by completion until $\mathcal{A}_k$ where $\mathcal{A}_k$ is either a fix-point automaton such that $\mathcal{L}(\mathcal{A}_k)\cap\mathcal{L}(\mathcal{A}_P)=\emptyset$, or $\mathcal{L}(\mathcal{A}_k)\cap\mathcal{L}(\mathcal{A}_P)\neq\emptyset$. For the former, by Proposition 1 each term of $\mathcal{L}(\mathcal{A}_P)$ is unreachable. For the latter, an exact completion step on $\mathcal{A}_{k-1}$ is performed using α $(C^{\mathcal{R}}_{\alpha}(\mathcal{A}_{k-1}))$. If $\mathcal{L}(C^{\mathcal{R}}_{\alpha}(\mathcal{A}_{k-1}))\cap\mathcal{L}(\mathcal{A}_P)=\emptyset$ then the abstraction function γ has been too coarse at k^{th} completion step. So, according to Proposition 3, a new abstraction function γ' is obtained by refining γ to ensure that $\mathcal{L}(C^{\mathcal{R}}_{\gamma'}(\mathcal{A}_{k-1}))\cap\mathcal{L}(\mathcal{A}_P)=\emptyset$. Otherwise, the backward analysis following Proposition 5 is performed from $\mathcal{A}_P$ in order to detect the completion step which is guilty of this non-empty intersection. The completion step i is guilty if $\mathcal{L}(\mathcal{A}_i)\cap\mathcal{L}((K^{\mathcal{R}}_{\alpha})^{(k-i)}(\mathcal{A}_P))\neq\emptyset$. As soon as the incriminated completion step is detected, the abstraction function γ is refined and the completion restarts from this completion step and using the new abstraction function, and so on. If no completion step is guilty then $\mathcal{R}^*(\mathcal{L}(\mathcal{A}))\cap\mathcal{L}(\mathcal{A}_P)\neq\emptyset$.

In Algorithm 1 and for the remainder of this section, $\mathcal{A}^e$ denotes $\mathcal{C}^{\mathcal{R}}_{\alpha}(\mathcal{A})$. $\mathcal{A}_i$ is the i^{th} element of the list *aut_list*. Let *aut_list* be a list of n elements $[e_1,\ldots,e_n]$, *aut_list[i]* denotes the sublist $[e_1,\ldots,e_i]$ with $i\leq n$. The function *aut_list::x* adds x at the end of the list *aut_list*.

Algorithm 1 (Refinement semi-algorithm) *Given $\mathcal{R}$ a linear TRS, $\mathcal{A}$ a tree automaton, $\mathcal{A}_P$ a tree automaton recognising a set of unwanted terms, γ an abstraction function and α an $(\mathcal{A},\mathcal{R})-$exact abstraction function, $Comp_{Ref}(\mathcal{A}, \mathcal{R},\mathcal{A}_P,\gamma,\alpha)$ is defined as follows:*

Variables
$\mathcal{A}_P^{temp}$:= $\mathcal{A}_P$;
aut_list := $[\mathcal{A};C^{\mathcal{R}}_{\gamma}(\mathcal{A})]$; (* list of automata *)
i := 1 ; (completion step number *)*
result := true;
00 ***Begin***
01 ***While*** $(\mathcal{A}_i\neq\mathcal{A}_{i-1})$ *and (result = true)* ***do***
02 ***If*** $\mathcal{L}(\mathcal{A}_i)\cap\mathcal{L}(\mathcal{A}_P)=\emptyset$ ***then*** — *If the intersection is empty between*
03 *aut_list := aut_list::* $C^{\mathcal{R}}_{\gamma}(\mathcal{A}_i)$; — $\mathcal{L}(\mathcal{A}_i)$ *and* $\mathcal{L}(\mathcal{A}_P)$ *then a normal*
04 *i := i + 1;* — *completion step is performed.*
05 ***Else***
06 *status :=* $\mathcal{L}(C^{\mathcal{R}}_{\alpha}(\mathcal{A}_{i-1}))\cap\mathcal{L}(\mathcal{A}_P)$; — *While the intersection between* $\mathcal{L}(\mathcal{A}^e_i)$
07 ***While*** *(status* $\neq\emptyset$*) and (*$i>0$*)* ***do*** — *and* $\mathcal{L}(\mathcal{A}_P)$ *is not empty, and while*
08 $\mathcal{A}_P^{temp}:=K^{\mathcal{R}^t}_{\alpha}(\mathcal{A}_P^{temp})$; — $i>0$, *Definition 10 is used*
09 *i := i - 1;* — *to compute a new automaton* A_P.
10 ***If*** $i>0$ ***then***
11 *status :=* $\mathcal{L}(C^{\mathcal{R}}_{\alpha}(\mathcal{A}_{i-1}))\cap\mathcal{L}(\mathcal{A}_P^{temp})$;
12 ***EndIf***
13 ***Done*** — *There are 2 cases to make* **while** *stop:*
14 $\mathcal{A}_P^{temp}$:= $\mathcal{A}_P$; — *- i = 0 (and* $\mathcal{L}(\mathcal{A}^e_i)\cap\mathcal{L}(\mathcal{A}_P)\neq\emptyset$*).*
15 ***If*** *(i = 0)* ***then*** — *In this case we can conclude that*
16 *result := false ;* — $\mathcal{R}^*(\mathcal{L}(\mathcal{A}_0))\cap\mathcal{L}(\mathcal{A}_P)\neq\emptyset$;

17 **Else** - $\mathcal{L}(\mathcal{A}_i^e) \cap \mathcal{L}(\mathcal{A}_P) = \emptyset$ *(and* $i \geq 0$*).*
18 *Find* $Disc \subseteq \Delta_i \setminus \Delta_{i-1}$*;*
19 $\gamma := \gamma_{Disc}$ *In this case* γ *is refined and*
20 *aut_list := aut_list[i-1]::*$C_\gamma^{\mathcal{R}}(\mathcal{A}_i)$*;* *a completion step is performed.*
21 **EndIf**
22 **EndIf**
23 **Done**
24 *return result;*
25 **End**

Theorems 2 shows that the semi-algorithm above is sound.

Theorem 2 (Soundness of Algorithm 1). *If* $Comp_{Ref}(\mathcal{A}, \mathcal{R}, \mathcal{A}_P, \gamma, \alpha) =$ *true then* $\mathcal{R}^*(\mathcal{L}(\mathcal{A})) \cap \mathcal{L}(\mathcal{A}_P) = \emptyset$ *else if* $Comp_{Ref}(\mathcal{A}, \mathcal{R}, \mathcal{A}_P, \gamma, \alpha) =$ *false then* $\mathcal{R}^*(\mathcal{L}(\mathcal{A})) \cap \mathcal{L}(\mathcal{A}_P) \neq \emptyset$.

Proof. Algorithm 1 terminates when either a fix-point automaton is computed or *result* = *false*. If *result* = *false* then line 16 has been executed. Moreover, according to lines 7, 8, 9 and 11, $\mathcal{R}^*(\mathcal{L}(\mathcal{A})) \cap \mathcal{L}(\mathcal{A}_P) \neq \emptyset$. Indeed, if $i = 0$ at line 9 then i has been equal to 1 either at line 6 (before entering in the **while**) or at line 9 (at the previous iteration). Consequently, $\mathcal{L}(C_\alpha^{\mathcal{R}}(\mathcal{A})) \cap \mathcal{L}(\mathcal{A}_P^{temp}) \neq \emptyset$ since i becomes 0 next, at line 9. Consequently, as $\mathcal{A}_P^{temp}$ represents $(K_\alpha^{\mathcal{R}^t})^k(\mathcal{A}_P)$, according to Propositions 5 and 4, one deduces that $\mathcal{R}^*(\mathcal{L}(\mathcal{A})) \cap \mathcal{L}(\mathcal{A}_P) \neq \emptyset$.

If *result* = *true* and $\mathcal{A}_i$ is a fix-point automaton then line 15 has never been executed. So, to break **while** at line 7, *status* needs to be $\emptyset$. So, there exist $n \in \mathbb{N}$ and an abstraction function γ' built from γ such that $(C_{\gamma'}^{\mathcal{R}})^{(n)}(\mathcal{A}) = (C_{\gamma'}^{\mathcal{R}})^{(n+1)}(\mathcal{A})$. According to Proposition 1, $\mathcal{L}((C_{\gamma'}^{\mathcal{R}})^{(n)}(\mathcal{A})) \supseteq \mathcal{R}^*(\mathcal{L}(\mathcal{A}))$. Consequently, $\mathcal{R}^*(\mathcal{L}(\mathcal{A})) \cap \mathcal{L}(\mathcal{A}_P) = \emptyset$.

The case when *result* = *false* and $\mathcal{A}_i$ is a fix-point automaton, reduces to the first case handled in this proof.

Finally, Theorem 3 claims that our semi-algorithm is complete in the sense that if an unwanted term is reachable then Algorithm 1 returns false.

Theorem 3 (Partial Completeness of Algorithm 1). *If* $\mathcal{R}^*(\mathcal{L}(\mathcal{A})) \cap \mathcal{L}(\mathcal{A}_P) \neq \emptyset$ *then* $Comp_{Ref}(\mathcal{A}, \mathcal{R}, \mathcal{A}_P, \gamma, \alpha) =$ *false.*

Proof. Suppose each time, at line 18 *Disc* is set such that $Disc = \Delta_i \setminus \Delta_{i-1}$. Thus γ tends to behave as α. So, let γ be α. Since $\mathcal{R}^*(\mathcal{L}(\mathcal{A})) \cap \mathcal{L}(\mathcal{A}_P) \neq \emptyset$, there exists $n \in \mathbb{N}$ and $n > 0$ such that $\mathcal{L}((C_\gamma^{\mathcal{R}})^{(n)}(\mathcal{A})) \cap \mathcal{L}(\mathcal{A}_P) \neq \emptyset$ and $\mathcal{L}((C_\gamma^{\mathcal{R}})^{(n-1)}(\mathcal{A})) \cap \mathcal{L}(\mathcal{A}_P) = \emptyset$. Consequently, in this setting, *status* at line 6 is different from $\emptyset$ and $i = n$. Consequently, $K_\alpha^{\mathcal{R}^t}(\mathcal{A}_P)$ is computed. According to Propositions 2 and 5, one can deduce that $\mathcal{L}((C_\gamma^{\mathcal{R}})^{(n-1)}(\mathcal{A})) \cap \mathcal{L}(K_\alpha^{\mathcal{R}^t}(\mathcal{A}_P)) \neq \emptyset$. So, by a simple induction, one trivially obtains $\mathcal{L}((C_\gamma^{\mathcal{R}})^{(0)}(\mathcal{A})) \cap \mathcal{L}((K_\alpha^{\mathcal{R}^t})^{(n)}(\mathcal{A}_P)) \neq \emptyset$ that corresponds to the value stored in *status* at line 11 after $n-1$ iteration in the **while** at line 7. Since $i = 1$ and *status*$\neq \emptyset$, a new iteration is performed. Finally, $i = 0$ and *status*$\neq \emptyset$. Thus, *result* is set to false, the **while** at line 1 is broken and Alg. 1 returns false.

An advantage of our approach is that the abstraction function γ given as input to $Comp_{Ref}$ does not require to be very pertinent. As explained at the very beginning of this paper, it is very easy to generate abstraction functions leading to inconclusive analyses. Our algorithm attempts to fix this kind of abstraction functions in order to perform an unreachability analysis. If the inconclusive analysis is not of the abstraction function concern then our algorithm states that some of the unwanted terms are actually reachable. This algorithm has been prototyped in the Timbuk tool [14].

4 Experiments

Our abstraction refinement technique for completion has been applied for the verification of a simple two processes counting system. The following TRS describes the behaviour of two processes each one equipped with an input list and a FIFO. Each process receives a list of symbols '+' and '−' to count, as an input. One of the processes, say P_+, is counting the '+' symbols and the other one, say P_- is counting the '−' symbols. When P_+ receives a '+', it counts it and when it receives a '−', it adds the symbol to P_-'s FIFO. The behaviour of P_- is symmetric. When a process' input list and FIFO is empty then it stops and gives the value of its counter.

Here is a possible rewrite specification of this system, given in the Timbuk language, where `S(_,_,_,_)` represents a configuration with a process P_+, a process P_-, P_+'s FIFO and P_-'s FIFO. The term `Proc(_,_)` represents a process with an input list and a counter, `add(_,_)` implements adding of an element in a FIFO, and `cons`, `nil`, `s`, `o` are the usual constructors for lists and natural numbers.

```
Ops
        S:4 Proc:2 Stop:1 cons:2 nil:0 plus:0 minus:0 s:1 o:0 end:0 add:2
Vars    x y z u c m n
TRS R1
 add(x, nil) -> cons(x, nil)
 add(x, cons(y, z)) -> cons(y, add(x, z))
 S(Proc(cons(plus, y), c), z, m, n) -> S(Proc(y, s(c)), z, m, n)
 S(Proc(cons(minus, y), c), u, m, n) -> S(Proc(y, c), u, m, add(minus, n))
 S(x, Proc(cons(minus, y), c), m, n) -> S(x, Proc(y, s(c)), m, n)
 S(x, Proc(cons(plus, y), c), m, n) -> S(x, Proc(y, c), add(plus, m), n)
 S(Proc(x, c), z, cons(plus,m), n) -> S(Proc(x, s(c)), z, m, n)
 S(x, Proc(z, c), m, cons(minus,n)) -> S(x, Proc(z, s(c)), m ,n)
 S(Proc(nil, c), z, nil, n)  -> S(Stop(c), z, nil, n)
 S(x, Proc(nil, c), m, nil) -> S(x, Stop(c), m, nil)
```

The set of initial configurations of the system is described by the following tree automaton, where each process has a counter initialised to 0 and has an unbounded input list (with both '+' and '−') and with at least one symbol.

```
Automaton A1
States q0 qinit qzero qnil qlist qsymb
Final States q0
Transitions
 cons(qsymb, qnil) ->qlist    cons(qsymb, qlist) -> qlist       o -> qzero
 Proc(qlist, qzero) -> qinit  S(qinit, qinit, qnil, qnil) -> q0  nil -> qnil
 plus -> qsymb                minus -> qsymb
```

On this specification, we aim at proving that, for any input lists, there is no possible deadlock. In this example, a deadlock is a configuration where a process has stopped but there are still symbols to count in its FIFO. This property is specified by a tree automaton `Bad_state` recognising a system for which one of the two processes has stopped and whose FIFO is not empty.

Before computing an over-approximation of the reachable configurations of the two processes and according to Def. 4, we give the simple abstraction function γ that satisfies the following property: Let σ_1, σ_2 be two substitutions from $\mathcal{X}$ into $\mathcal{Q}$ such that $\sigma_1 \neq \sigma_2$. Let $l \to r$ be a rule of the TRS `R` of the Timbuk specification. Then, for any state q and for each functional position p of r, $\gamma(l \to r, \sigma_1, q)(p) = \gamma(l \to r, \sigma_2, q)(p)$. With such an abstraction function, the completion converges to a fix-point automaton $\mathcal{A}$ within 8 completion steps. Unfortunately, the intersection between the tree automata $\mathcal{A}$ and `Bad_state` is non-empty. No conclusion can be drawn since $\mathcal{A}$ is an over-approximation.

Using the abstraction refinement technique, the following scenario sets: Three completion steps are performed and a non-empty intersection is found. So, the backward analysis is run and finally reaches the initial tree automaton. In conclusion, there exists a deadlock for this system.

The problem found here can be fixed by adding an additional symbol: 'end' which has to be added by process P_+ to P_- FIFO when P_+ has reached the end of its list, and symmetrically for P_-. Then, a process can stop if and only if it has reached the end of its list and if it has read the 'end' symbol in its FIFO. Then, the TRS of the previous specification is modified a little. For the new specification, we use the same kind of abstraction function as in the first experiment. Timbuk finds a fix-point automaton $\mathcal{A}'$ within 9 completion steps. Unfortunately, the intersection between the tree automata $\mathcal{A}'$ and `Bad_state` is non-empty anew. Using the abstraction refinement, Timbuk finds another fix-point automaton $\mathcal{A}''$ whose language contains no term recognised by the tree automaton `Bad_state`. Consequently, we have managed to prove that the patched system is actually deadlock free.

5 Conclusion

The paper describes a new approach for automatically generating abstraction-based over-approximations guided by a set of unwanted terms. In the infinite state system verification framework, our work can be considered as an abstraction-based approach guided by a safety/security property which either can conclude that a safety property is satisfied or can detect a violation of the given property or may not terminate. The last point is not surprising since the reachability problem is undecidable on non terminating TRSs. Furthermore, in [3], we show that in some cases, unreachable terms are in all computable over-approximations. So, refinement may be unfruitful in the rewriting approximation framework. However, our first experimental results are promising. Moreover, the positive results obtained in the framework of security protocols and Java programs analysis, let us think that our refinement approach can work in practice

and then, can make the reachability analysis detailed in [12] available to a larger community of users. More experimentations are needed to compare the technique in this paper and the abstraction technique in [5], and to determine our interactive backward reachability analysis efficiency and pertinence on, e.g., Java programs. In many system analyses, backward analysis provides better results than forward analysis.

Related works The notion of abstraction function presented in [12] is not so far from the basic definition used in the framework of the abstraction-based verification of infinite-state systems.

Abstraction refinement is already used to make the definition of a good abstraction function easier and, consequently, to make the system verification easier. In [6], the CEGAR (*Counterexample-Guided Abstraction Refinement*) paradigm has been summarised and a general algorithm consisting in refining an abstraction function by analysing a spurious counterexample has been given. Our work fits almost exactly with this framework.

In [8, 7], the authors use abstraction refinement on Kripke structure and ACTL specification. When an abstract counterexample is found, the corresponding concrete counterexample is computed. If it does not correspond to a path in the concrete model, it means that it is a spurious counterexample, and the abstraction function is then refined to make the concrete model correspond to the given specification. The spurious counterexample analysis is done with a forward method.

In [18], an abstraction refinement method is used on transition systems for verifying invariants with a technique combining model checking, abstraction and deductive verification. Contrary to the three previous articles, the authors do not consider liveness properties, and the spurious counterexample analysis is done with a backward method. In [10], as part of predicate abstraction, predicates are automatically discovered by analysing spurious counterexamples. The method exposed in this paper is close to the above methods, but it works on different data structures.

In the field of tree automata, [5] computes an over-approximation with the help of an initial tree automata, tree transducers, and by merging states which either recognise the same language for a given depth or satisfy a given predicate. This merging is the origin of the over-approximation. A refinement can be done either by increasing the depth or by extending the predicate with a spurious counterexample. In our case, term rewriting systems are used instead of transducers. Moreover, the states fusion is guided by a safety/security property together with an abstraction function.

References

1. F. Baader and T. Nipkow. *Term Rewriting and All That.* Cambridge University Press, 1998.
2. Y. Boichut, Th. Genet, Th. Jensen, and L. Le Roux. Rewriting approximations for fast prototyping of static analyzers. In *proceedings of RTA*, LNCS 4533, pages 48–62. Springer, 2007.

3. Y. Boichut and P.-C. Héam. A Theoretical Limit for Safety Verification Techniques with Regular Fix-point Computations. *Information Processing Letters*, 2008. to appear.
4. Y. Boichut, P.-C. Héam, and O. Kouchnarenko. Automatic Verification of Security Protocols Using Approximations. Technical Report RR-5727, INRIA, 2005.
5. A. Bouajjani, P. Habermehl, A. Rogalewicz, and T. Vojnar. Abstract regular tree model checking. In *proceedings of INFINITY*, number 4 in BRICS Notes Series, pages 15–24, 2005.
6. E. Clarke. Counterexample-guided abstraction refinement. *proceedings of TIME*, 2003.
7. E. M. Clarke, O. Grumberg, S. Jha, Lu, and H. Veith. Counterexample-Guided Abstraction Refinement. In *proceedings of CAV*, pages 154–169, 2000.
8. E. M. Clarke, Y. Lu, S. Jha, and H. Veith. Tree-like counterexamples in model checking. In *proceedings of LICS*, pages 19–29, 2002.
9. H. Comon, M. Dauchet, R. Gilleron, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi. Tree Automata Techniques and Applications, 2002.
10. S. Das and D. L. Dill. Counter-example based predicate discovery in predicate abstraction. In *FMCAD*. Springer-Verlag, November 2002.
11. N. Dershowitz and J.-P. Jouannaud. *Handbook of Theoretical Computer Science*, volume B, chapter 6: Rewrite Systems, pages 244–320. Elsevier Science Publishers B. V, 1990.
12. G. Feuillade, Th. Genet, and V. VietTriemTong. Reachability analysis over term rewriting systems. *Journal of Automated Reasonning*, 33 (3-4), 2004.
13. Th. Genet and F. Klay. Rewriting for Cryptographic Protocol Verification. In *proceedings of CADE*, volume 1831 of *LNCS*, pages 271–290. Springer-Verlag, 2000.
14. Th. Genet and Valérie Viet Triem Tong. Reachability Analysis of Term Rewriting Systems with *timbuk*. In *proceedings of LPAR*, volume 2250 of *LNAI*, pages 691–702. Springer-Verlag, 2001.
15. R. Gilleron and S. Tison. Regular tree languages and rewrite systems. *Fundamenta Informatica*, 24(1/2):157–174, 1995.
16. P. Gyenizse and S. Vágvölgyi. Linear Generalized Semi-Monadic Rewrite Systems Effectively Preserve Recognizability. *Theoretical Computer Science*, 194(1-2):87–122, 1998.
17. F. Jacquemard. Decidable approximations of term rewriting systems. In *proceedings of RTA*, volume 1103, pages 362–376. Springer Verlag, 1996.
18. Y. Lakhnech, S. Bensalem, S. Berezin, and S. Owre. Incremental Verification by Abstraction. In *proceedings of TACAS 2001*, LNCS 2031. Springer-Verlag, 2001.
19. D. Monniaux. Abstracting cryptographic protocols with tree automata. In *proceedings of SAS*, volume 1694 of *LNCS*. Springer-Verlag, 1999.
20. T. Takai, Y. Kaji, and H. Seki. Right-linear finite-path overlapping term rewriting systems effectively preserve recognizability. In *proceedings of RTA*, volume 1833 of *LNCS*. Springer-Verlag, 2000.