Towards an Efficient Implementation of Tree Automata Completion *

E. Balland[1], Y. Boichut[1], T. Genet[2], and P.-E. Moreau[1]

[1] Loria, INRIA Nancy – Grand Est, France
[2] Irisa, INRIA Rennes-Bretagne Atlantique, France

Abstract. Term Rewriting Systems (TRSs) are now commonly used as a modeling language for applications. In 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. Using a tree automata completion technique, it has been shown that the non reachability of a term t can be verified by computing an over-approximation of the set of reachable terms and proving that t is not in the over-approximation. Since the verification of real programs gives rise to rewrite models of significant size, efficient implementations of completion are essential. We present in this paper a TRS transformation preserving the reachability analysis by tree automata completion. This transformation makes the completion implementation based on rewriting techniques possible. Thus, the reduction of a term to a state by a tree automaton is fully handled by rewriting. This approach has been prototyped in Tom, a language extension which adds rewriting primitives to Java. The first experiments are very promising relative to the state-of-the-art tool Timbuk.

1 Introduction

In the context of infinite state systems verification, a rising approach uses Term Rewriting Systems (TRSs) as a model and reachability analysis as a verification technique. In comparison with some other modeling techniques, TRSs have a great advantage: they can be both executed *and* verified. On one hand, comparing the execution of a TRS with the execution of a program gives a pragmatic way to check the coherence between the formal model and the program to be analyzed. On the other hand, most of the verification techniques have their Term Rewriting counterpart: model-checking [10], static analysis and abstract interpretation [13, 12, 19] or even interactive proofs [7]. Hence, it permits to use any of them on the TRS model. Furthermore, since all those techniques operate on a *common formal model*, i.e. TRS, this may lead to an elegant way to combine their verification power. However, like in the general verification setting, efficiency problems occur when trying to apply those rewriting techniques to real-size applications. This is the case when using model-checking on TRS models of Java programs [18, 11] or when using TRS based static analysis on cryptographic protocols [14] or on Java Bytecode programs [5]. Thus, having efficient verification tools on TRS models is crucial to guarantee their success as a modeling technique.

* This work was partially supported by ANR funding number ANR-06-SETI-14.

In this paper, we aim at improving significantly the static analysis part. The state of the art implementation is called Timbuk [16] and has been used to prove properties on TRS models of cryptographic protocols [14, 6] and Java Bytecode programs [5]. This tool constructs approximations of reachable terms using the so-called *tree automata completion* algorithm. Starting from a set of initial terms (representing respectively all possible function calls, initial configuration for parallel processes, etc.) it computes a regular super-set of all terms reachable by rewriting initial terms. This over-approximation, recognized by a tree automaton, represents either a super-set of all possible evaluations (partial or completed) for functions, or a super-set of all possible processes' behaviors for parallel processes. Then, it is possible to check some properties related to reachability (in particular safety and security properties) on this approximation. The work reported here improves by a factor 10 in general, and up to 100 on some Java examples, the efficiency of the tree automata completion. First, the proposed technique consists of decomposing each rewrite rule of the TRS in several simpler rules and to apply the completion on the transformed TRS. We show that the resulting automaton is also an approximation and thus the reachability is preserved by this TRS transformation. Second, an efficient implementation is obtained using compilation techniques, thanks to Tom [3, 4], a Java extension that offers powerful pattern-matching features.

After presenting the classical approach in Section 2, we present the transformation and prove that the reachability analysis after completion is preserved. Then, we detail in Section 4 how it has been implemented in Tom and show especially how some of the Tom features make the development painless. To conclude, we present in Section 5 promising experimental results on the verification of cryptographic protocols and Java program analysis.

2 Preliminaries

2.1 Terms and TRSs

Comprehensive surveys can be found in [9, 2] for term rewriting systems, and in [8, 17] for tree automata and tree language theory.

Let $\mathcal{F}$ be a possibly infinite set of symbols, associated with an arity function $ar : \mathcal{F} \rightarrow \mathbb{N}$, and let $\mathcal{X}$ be a countable set of variables. Let $<_{\mathcal{X}}$ be a total order relation on 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 $\mathcal{P}os(f(t_1, \ldots, t_n)) = \{\epsilon\} \cup \{i.p \mid 1 \leq i \leq n \text{ and } p \in \mathcal{P}os(t_i)\}$, and $\mathcal{P}os(t) = \{\epsilon\}$ when $t \in \mathcal{X}$. The depth of a term t, denoted by $depth(t)$ is the length of the maximal sequence in $\mathcal{P}os(t)$. 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 $t \in \mathcal{T}(\mathcal{F}, \mathcal{X})$ and A a set of symbols, let $\mathcal{P}os_A(t) = \{p \in \mathcal{P}os(t) \mid t(p) \in A\}$. Thus $\mathcal{P}os_{\mathcal{F}}(t)$ is the set of positions of t, at each of which a function symbol appears.

A term t is said to be *flat* if t is either a simple constant or a term of the form $f(x_1, \dots, x_n)$ where $x_1, \dots, x_n \in \mathcal{X}$. We say a term t is *almost flat* if t is of the form $f(t_1, \dots, t_n)$ and the t_i's are flat terms or variables.

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 terms $E \subseteq \mathcal{T}(\mathcal{F}, \mathcal{X})$ is $\mathcal{R}^*(E) = \{t \in \mathcal{T}(\mathcal{F}, \mathcal{X}) \mid \exists s \in E \text{ s.t. } s \to^{\star}_{\mathcal{R}} t\}$.

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 [17]. 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 that will be used to represent set E and over-approximation of $\mathcal{R}^*(E)$.

Let $\mathcal{Q}$ be an infinite set of symbols, with 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 normalized transition). *A* transition *is a rewrite rule $c \to q$, where c is a configuration i.e. $c \in \mathcal{T}(\mathcal{F} \cup \mathcal{Q})$ and $q \in \mathcal{Q}$. A* normalized transition *is a transition $c \to q$ where $c = f(q_1, \dots, q_n)$, $f \in \mathcal{F}$ whose arity is n, and $q_1, \dots, q_n \in \mathcal{Q}$.*

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

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

Definition 3 (Recognized language). *The tree language recognized by $\mathcal{A}$ in a state q is $\mathcal{L}(\mathcal{A}, q) = \{t \in \mathcal{T}(\mathcal{F}) \mid t \to^{\star}_{\mathcal{A}} q\}$. The language recognized 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 can be recognized by a tree automaton.*

Example 1. Let $\mathcal{A}$ be the tree automaton such that $\mathcal{Q}_f = \{q_0\}$ and $\Delta = \{a \to q_4, b \to q_5, c \to q_6, d \to q_7, f(q_4, q_5) \to q_1, h(q_6) \to q_2, h(q_7) \to q_3, g(q_1, q_2) \to q_0, g(q_1, q_3) \to q_0\}$. The language recognized by $\mathcal{A}$ is $\mathcal{L}(\mathcal{A}) = \{g(f(a,b), h(c)), g(f(a,b), h(d))\}$. This example is used throughout this paper to explain the concepts and algorithms presented in the paper.

Given a tree automaton $\mathcal{A}$ and a TRS $\mathcal{R}$, the tree automata completion algorithm, proposed in [13, 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 $\mathcal{A} = \mathcal{A}^0_{\mathcal{R}}$ 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 $s \rightarrow_{\mathcal{R}} t$ then $t \in \mathcal{L}(\mathcal{A}^{i+1}_{\mathcal{R}})$. If we find 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 we have $\mathcal{L}(\mathcal{A}^k_{\mathcal{R}}) = \mathcal{R}^*(\mathcal{L}(\mathcal{A}^0_{\mathcal{R}}))$ (or $\mathcal{L}(\mathcal{A}^k_{\mathcal{R}}) \supseteq \mathcal{R}^*(\mathcal{L}(\mathcal{A}))$ if $\mathcal{R}$ is not in the class of [12]). To build $\mathcal{A}^{i+1}_{\mathcal{R}}$ from $\mathcal{A}^i_{\mathcal{R}}$, we achieve a *completion step* which consists of finding *critical pairs* between $\rightarrow_{\mathcal{R}}$ and $\rightarrow_{\mathcal{A}^i_{\mathcal{R}}}$. To define the notion of critical pair, we extend the definition of substitutions to terms of $\mathcal{T}(\mathcal{F} \cup \mathcal{Q})$. If there exists a substitution $\sigma : \mathcal{X} \mapsto \mathcal{Q}$, a rule $l \rightarrow r \in \mathcal{R}$, and $q \in \mathcal{Q}$ satisfying $l\sigma \rightarrow^*_{\mathcal{A}^i_{\mathcal{R}}} q$ and $l\sigma \rightarrow_{\mathcal{R}} r\sigma$, we say that $\langle r\sigma, q\rangle$ is a critical pair. Note that since $\mathcal{R}$, $\mathcal{A}^i_{\mathcal{R}}$, and the set of states of $\mathcal{Q}$ used in $\mathcal{A}^i_{\mathcal{R}}$ are finite, there is only a finite number of critical pairs. Note also that, in our case, it is enough to consider only root overlap between rules of $\mathcal{R}$ and transitions of $\mathcal{A}^i_{\mathcal{R}}$. For every critical pair detected between $\mathcal{R}$ and $\mathcal{A}^i_{\mathcal{R}}$ such that $r\sigma \not\rightarrow^*_{\mathcal{A}^i_{\mathcal{R}}} q$, the tree automaton $\mathcal{A}^{i+1}_{\mathcal{R}}$ is constructed by adding a new transition $r\sigma \rightarrow q$ to $\mathcal{A}^i_{\mathcal{R}}$ such that $\mathcal{A}^{i+1}_{\mathcal{R}}$ recognizes $r\sigma$ in q, i.e. $r\sigma \rightarrow_{\mathcal{A}^{i+1}_{\mathcal{R}}} q$.

$$\begin{tikzcd}
l\sigma \arrow[r, "\mathcal{R}"'] \arrow[d, "\mathcal{A}^i_{\mathcal{R}}"', "*"] & r\sigma \arrow[dl, bend left, "*"', "\mathcal{A}^{i+1}_{\mathcal{R}}"] \\
q &
\end{tikzcd}$$

However, the transition $r\sigma \rightarrow q$ is not necessarily a normalized transition of the form $f(q_1, \ldots, q_n) \rightarrow q$ and so it has to be normalized first. Since normalization consists of associating state symbols to subterms of the left-hand side of the new transition, it always succeeds. Note that, when using new states to normalize the transitions, completion is as precise as possible. However, without approximation, completion is likely not to terminate (because of general undecidability results [17]). To enforce termination, and produce an over-approximation, the completion algorithm is parametrized by a set N of *approximation rules*. When the set N is used during completion to normalize transitions, the obtained tree automata are denoted by $\mathcal{A}^1_{N,\mathcal{R}}, \ldots, \mathcal{A}^k_{N,\mathcal{R}}$. Each such rule describes a context in which a list of rules can be used to normalize a term. For all $s, l_1, \ldots, l_n \in \mathcal{T}(\mathcal{F} \cup \mathcal{Q}, \mathcal{X})$ and for all $x, x_1, \ldots, x_n \in \mathcal{Q} \cup \mathcal{X}$, the general form for an approximation rule is:

$$[s \rightarrow x] \rightarrow [l_1 \rightarrow x_1, \ldots, l_n \rightarrow x_n].$$

The expression $[s \rightarrow x]$ is a pattern to be matched with the new transitions $t \rightarrow q'$ obtained by completion. The expression $[l_1 \rightarrow x_1, \ldots, l_n \rightarrow x_n]$ is a set of rules used to normalize t. To normalize a transition of the form $t \rightarrow q'$, we match s with t and x with q', obtain a substitution σ from the matching and then we normalize t with the rewrite system $\{l_1\sigma \rightarrow x_1\sigma, \ldots, l_n\sigma \rightarrow x_n\sigma\}$. Furthermore, if $\forall i \in [1..n]$, $x_i \in \mathcal{Q}$ or $x_i \in \mathcal{V}ar(l_i) \cup \mathcal{V}ar(s) \cup \{x\}$ then, since $\sigma : \mathcal{X} \mapsto \mathcal{Q}$, $x_1\sigma, \ldots, x_n\sigma$ are necessarily states. If a transition cannot be fully normalized using approximation rules N, normalization is finished using some new states, see Example 2. Such normalization rules can either be defined by hand, in order to prove precise properties on specific systems or can be automatically generated in more specific settings [5, 6].

The main property of the tree automata completion algorithm is that, whatever the state labels used to normalize the new transitions are, if completion terminates then it produces an over-approximation of reachable terms [12].

Theorem 1 ([12]). *Let $\mathcal{R}$ be a left-linear TRS, $\mathcal{A}$ be a tree automaton, and N be a set of approximation rules. If completion terminates on $\mathcal{A}^k_{N,\mathcal{R}}$ then*

$$\mathcal{L}(\mathcal{A}^k_{N,\mathcal{R}}) \supseteq \mathcal{R}^*(\mathcal{L}(\mathcal{A})).$$

Here is a simple example illustrating completion and the use of approximation rules when the language $\mathcal{R}^*(E)$ is not regular.

Example 2. Let $\mathcal{A}$ be the tree automaton given in Example 1. In the following we illustrate the effect of approximation rules. First, we consider the TRS $\mathcal{R} = \{g(f(a,x), h(y)) \to g(f(a,\ f(a,x)), h(h(y)))\}$, composed of a single rule. The set of $\mathcal{R}$-descendants of $\mathcal{L}(\mathcal{A})$ is $\mathcal{R}^*(\mathcal{L}(\mathcal{A})) = \{g(f^n(a,b), h^n(\{c,d\})) \mid n \geq 0\}$.
Let N be the set of approximation rules such that $N = \{[g(f(a, f(a,x)), h(h(y))) \to z] \to [a \to q_4,\ f(q_4,x) \to q_5,\ f(q_4,q_5) \to q_5,\ h(y) \to q_6,\ h(q_6) \to q_6,\ g(q_5,q_6) \to z]\}$. Intuitively, the *approximated* set of descendants will be the following: $\mathcal{L}(\mathcal{A}^2_{N,\mathcal{R}}) = \{g(f^n(a,b), h^m(\{c,d\})) \mid n,m \geq 0\}$.

To get this result, we first compute the critical pairs. Let us consider $\sigma_1 = \{x \mapsto q_5, y \mapsto q_6\}$ and $\sigma_2 = \{x \mapsto q_5, y \mapsto q_7\}$, we have:

1. $g(f(a,q_5), h(q_6)) \to^*_{\mathcal{A}} q_0$ and $g(f(a,q_5), h(q_6)) \to_{\mathcal{R}} g(f(a,\ f(a,q_5)), h(h(q_6)))$,
2. $g(f(a,q_5), h(q_7)) \to^*_{\mathcal{A}} q_0$ and $g(f(a,q_5), h(q_7)) \to_{\mathcal{R}} g(f(a, f(a,q_5)), h(h(q_7)))$.

Let us call $l = g(f(a,x), h(y))$ and $r = g(f(a,\ f(a,x)), h(h(y)))$ the respective left-hand side and right-hand side of the rule of $\mathcal{R}$. The transitions (1) $r\sigma_1 \to q_0$ and (2) $r\sigma_2 \to q_0$ have to be normalised using N. To normalize the transition (1), we match the pattern of the approximation rule , i.e. $g(f(a, f(a,x)), h(h(y)))$, with $r\sigma_1$ and match z with q_0, and thus obtain a substitution $\sigma = \{x \mapsto q_5, y \mapsto q_6, z \mapsto q_0\}$. Applying σ to $[a \to q_4,\ f(q_4,x) \to q_5,\ f(q_4,q_5) \to q_5,\ h(y) \to q_6,\ h(q_6) \to q_6,\ g(q_5,q_6) \to z]$ gives $[a \to q_4, f(q_4,q_5) \to q_5, h(q_6) \to q_6,\ g(q_5,q_6) \to q_0]$. This last system is used to normalize the transition $r\sigma_1 \to q_0$ into the set $S_1 = \{a \to q_4,\ f(q_4,q_5) \to q_5,\ h(q_6) \to q_6,\ g(q_5,q_6) \to q_0\}$. At the same time, the same process is performed for the transition (2), resulting in $S_2 = \{a \to q_4,\ f(q_4,q_5) \to q_5,\ h(q_7) \to q_6,\ h(q_6) \to q_6,\ g(q_5,q_6) \to q_0\}$. The tree automaton $\mathcal{A}^1_{N,\mathcal{R}}$ is then obtained by adding $S_1 \cup S_2$ to $\mathcal{A}$.

The completion process continues for another step. As there are no more critical pair, it ends on $\mathcal{A}^2_{N,\mathcal{R}} = \mathcal{A}^1_{N,\mathcal{R}}$ whose set of transitions is $\{a \to q_4,\ b \to q_5,\ c \to q_6,\ d \to q_7,\ f(q_4,q_5) \to q_1,\ h(q_6) \to q_2,\ h(q_7) \to q_3,\ g(q_1,q_2) \to q_0,\ g(q_1,q_3) \to q_0,\ f(q_4,q_5) \to q_5,\ h(q_6) \to q_6,\ g(q_5,q_6) \to q_0,\ h(q_7) \to q_6\}$. We have $\mathcal{L}(\mathcal{A}^2_{N,\mathcal{R}}) = \{g(f^n(a,b), h^m(\{c,d\})) \mid n,m \geq 0\}$ which is an over-approximation of $\mathcal{R}^*(\mathcal{L}(\mathcal{A})) = \{g(f^n(a,b), h^n(\{c,d\})) \mid n \geq 0\}$.

The tree automata completion algorithm and the approximation mechanism are implemented in the Timbuk [16] tool. In the previous example, once the fixpoint automaton $\mathcal{A}^k_{N,\mathcal{R}}$ has been computed, it is possible to check whether some terms are reachable, i.e. recognized by $\mathcal{A}^k_{N,\mathcal{R}}$ or not. This can be done using tree automata intersections [12]. Another way to do that is to search for instances of a pattern t, where t is a linear term of $\mathcal{T}(\mathcal{F},\mathcal{X})$, in the tree automaton. Given t it is possible to check if there exists a substitution $\sigma : \mathcal{X} \mapsto \mathcal{Q}$ and a state $q \in \mathcal{Q}$ such that $t\sigma \rightarrow^*_{\mathcal{A}^k_{N,\mathcal{R}}} q$. If such a solution exists then it proves that there exists a term $s \in \mathcal{T}(\mathcal{F})$, a position $p \in \mathcal{P}os(s)$ and a substitution $\sigma' : \mathcal{X} \mapsto \mathcal{T}(\mathcal{F})$ such that $s[t\sigma']_p \in \mathcal{L}(\mathcal{A}^k_{N,\mathcal{R}}) \supseteq \mathcal{R}^*(\mathcal{L}(\mathcal{A}))$, i.e. that $t\sigma'$ occurs as a subterm in the language recognized by $\mathcal{L}(\mathcal{A}^k_{N,\mathcal{R}})$. On the other hand, if there is no solution then it proves that no such term is in the over-approximation, hence it is not in $\mathcal{R}^*(\mathcal{L}(\mathcal{A}))$, i.e. it is not reachable. For instance, the pattern $g(h(x), f(y,z))$ has no solution on $\mathcal{A}^2_{N,\mathcal{R}}$ of Example 2, meaning that no term containing any ground instance of this pattern is reachable from $g(f(a,b),h(c))$ and from $g(f(a,b),h(d))$.

3 TRS Transformation Preserving Reachability

In this section, we propose a TRS transformation preserving the reachability analysis of the original one. This transformation makes the completion implementation in Section 4 simpler because of the particular form of the resulting rules.

3.1 Definition of the TRS Transformation ϕ

We first propose in Definition 4 a function which associates to a term over $\mathcal{F}$ and $\mathcal{X}$ a term which we consider as its context.

Definition 4. *Let $\mathcal{F}'$ be the set of function symbols C_t with $t \in \mathcal{T}(\mathcal{F},\mathcal{X})$ and $\mathcal{F} \cap \mathcal{F}' = \emptyset$. We define the function $\rho : \mathcal{T}(\mathcal{F},\mathcal{X}) \rightarrow \mathcal{T}(\mathcal{F}',\mathcal{X})$ such that $\forall t \in \mathcal{T}(\mathcal{F},\mathcal{X})$:*

$$\rho(t) = \begin{cases} C_t(x_1,\ldots,x_n) & \textit{where } \mathcal{V}ar(t) = \{x_1,\ldots,x_n\} \textit{ and } x_i <_{\mathcal{X}} x_{i+1} \textit{ if} \\ & t = f(t_1,\ldots,t_n) \\ t & \textit{if } t \in \mathcal{X} \end{cases}$$

Example 3. Consider $l = g(f(a,x),h(y))$ which is the left hand-side of the rule in Example 2. Then, $\rho(l)$ is equal to $C_{g(f(a,x),h(y))}(x,y)$. Note that for any ground term t, $\rho(t) = C_t$.

The following definition allows the construction of a set of rules from a given term. This TRS allows the rewriting of the given term t into its context C_t. This definition is central for the construction of the transformation $\phi(\mathcal{R})$ given in Definition 6.

Definition 5. *Given a term $t \in \mathcal{T}(\mathcal{F},\mathcal{X})$ and the function $\rho : \mathcal{T}(\mathcal{F},\mathcal{X}) \rightarrow \mathcal{T}(\mathcal{F}',\mathcal{X})$:*

$$\textsc{Trs}_\rho(t) = \begin{cases} \{f(\rho(t_1),\ldots,\rho(t_n)) \rightarrow \rho(t)\} \cup \bigcup_{i=1}^n \textsc{Trs}_\rho(t_i) & \textit{if } t = f(t_1,\ldots,t_n) \\ \emptyset & \textit{if } t \in \mathcal{X}. \end{cases}$$

Example 4. Let $l = g(f(a,x),h(y))$ be the left hand-side of the rule in Example 2. Then, $\textsc{Trs}_\rho(l) = \{a \rightarrow C_a,\ f(C_a,x) \rightarrow C_{f(a,x)}(x),\ h(y) \rightarrow C_{h(y)}(y),\ g(C_{f(a,x)}(x), C_{h(y)}(y)) \rightarrow C_{g(f(a,x),h(y))}(x,y)\}$.

Now we define the transformed TRS $\phi(\mathcal{R})$.

Definition 6 (TRS Transformation). *Given a set of rewrite rules $\mathcal{R}$:*

$$\phi(\mathcal{R}) = \bigcup_{l \to r \in \mathcal{R}} \textsc{Trs}_\rho(l) \cup \{\rho(l) \to r\}$$

Example 5. Let $\mathcal{R}$ be the TRS defined in Example 2. As there is only one rule $l \to r$ with $l = g(f(a,x), h(y))$ and $r = g(f(a, f(a,x)), h(h(y)))$, we have $\phi(\mathcal{R}) = \textsc{Trs}_\rho(l) \cup \{C_l(x,y) \to r\}$ where $\textsc{Trs}_\rho(l)$ is defined as in Example 4.

Note that for any TRS $\mathcal{R}$, the TRS $\phi(\mathcal{R})$ has two properties. First, for each rule $l \to r$ of $\phi(\mathcal{R})$, l is almost flat (see the definition in Section 2.1). So $depth(l)$ is in the worst case equal to three. And second, for a rule $l \to r \in \mathcal{R}$, r is reachable by rewriting from l and using $\phi(\mathcal{R})$. The latter is emphasized in the following proposition.

Proposition 1. *Let $\mathcal{R}$ be a left-linear TRS, for any rule $l \to r \in \mathcal{R}$:*

$$r \in \phi(\mathcal{R})^*(l)$$

Proof. Direct consequence of Definition 6. □

Another trivial property of $\phi(\mathcal{R})$ is about its linearity which is the same as $\mathcal{R}$.

Proposition 2. *If $\mathcal{R}$ is a left-linear TRS, then so is $\phi(\mathcal{R})$.*

Proof. Direct consequence of Definitions 6 and 4. □

3.2 Preservation of Reachability

As claimed at the very beginning of this section, the reachability analysis performed on $\phi(\mathcal{R})$ can be propagated to the one performed on $\mathcal{R}$ itself. In other words, given a set of terms E, an over-approximation of the set of terms reachable from E and using $\phi(\mathcal{R})$ is also an over-approximation of the set of reachable terms which can be computed from E using $\mathcal{R}$.

Theorem 2. *Let $\mathcal{R}$ and $\mathcal{A}$ be respectively a left-linear TRS and a tree automaton such that $\mathcal{A} = \langle \mathcal{F}, \mathcal{Q}, \mathcal{Q}_f, \Delta \rangle$. Let $\mathcal{A}'$ be a tree automaton such that $\mathcal{A}' = \langle \mathcal{F} \cup \mathcal{F}', \mathcal{Q}, \mathcal{Q}_f, \Delta \rangle$ where $\mathcal{F}'$ is specified as in Definition 4. For any set of approximation rules N, if completion terminates on $\mathcal{A}'^{k}_{N,\phi(\mathcal{R})}$ then*

$$\mathcal{R}^*(\mathcal{L}(\mathcal{A})) \subseteq \mathcal{L}(\mathcal{A}'^{k}_{N,\phi(\mathcal{R})}).$$

Proof. Let t and u be two ground terms over $\mathcal{F}$ such that $t \to_{\mathcal{R}} u$. There exists a rule $l \to r$ in $\mathcal{R}$, a substitution $\mu : \mathcal{X} \to \mathcal{T}(\mathcal{F})$ and a position p of t such that $t|_p = l\mu$ and $u = t[r\mu]_p$. According to Proposition 1, for a rule $l \to r \in \mathcal{R}$, $r \in \phi(\mathcal{R})^*(l)$. Thus, $t \to^*_{\phi(\mathcal{R})} u$. Consequently, for $\mathcal{R}$ and $\mathcal{A}'$, $\mathcal{R}^*(\mathcal{L}(\mathcal{A}')) \subseteq \phi(\mathcal{R})^*(\mathcal{L}(\mathcal{A}'))$. Note that since $\mathcal{A}'$ differs from $\mathcal{A}$ only because of its set $\mathcal{F}'$ of symbols, we have $\mathcal{L}(\mathcal{A}) = \mathcal{L}(\mathcal{A}')$ and thus $\mathcal{R}^*(\mathcal{L}(\mathcal{A})) \subseteq \phi(\mathcal{R})^*(\mathcal{L}(\mathcal{A}'))$. According to Theorem 1 and Proposition 2, $\phi(\mathcal{R})^*(\mathcal{L}(\mathcal{A}')) \subseteq \mathcal{L}(\mathcal{A}'^{k}_{N,\phi(\mathcal{R})})$. So one can deduce that $\mathcal{R}^*(\mathcal{L}(\mathcal{A})) \subseteq \mathcal{L}(\mathcal{A}'^{k}_{N,\phi(\mathcal{R})})$. □

Note that given $\mathcal{A}$, $\mathcal{R}$ and N, one can perform completion using $\phi(\mathcal{R})$ and N without modifying N. Moreover, for each r' of $l' \to r' \in \phi(\mathcal{R})$, either r' occurs in the right-hand side of a rule in $\mathcal{R}$, or r' is a flat term. The former is normalized by the set of approximation rules if it is necessary. For the latter, any transition resulting from the application of such a rule is already normalised. For example, given an automaton $\mathcal{A}$ and the rule $g(C_{f(a,x)}(x), C_{h(y)}(y)) \to C_{g(f(a,x),h(y))}(x, y)$ introduced in Example 4, if there exists $\sigma : \mathcal{X} \to \mathcal{Q}$ and q a state of $\mathcal{A}$ such that $g(C_{f(a,x)}(\sigma(x)), C_{h(y)}(\sigma(y))) \to^*_{\mathcal{A}} q$, then $C_{g(f(a,x),h(y))}(\sigma(x), \sigma(y)) \to q$ is added to $\mathcal{A}$. And this transition is already normalised. Consequently, N does not act for this kind of rule.

We have shown in this section that the TRS transformation is sound from a reachability point of view: each term actually reachable by $\mathcal{R}$ can be computed using $\phi(\mathcal{R})$. Indeed, for an over-approximation *App* computed using $\phi(\mathcal{R})$ from a set of terms E ($\mathcal{R}^*(E) \subseteq App$), if a term $t \notin App$ then t is actually unreachable from E using either $\phi(\mathcal{R})$ or $\mathcal{R}$.

4 Implementation in Tom

In this section, we show how the completion with $\phi(\mathcal{R})$ has been implemented in the Tom language. The main principle of Tom is to integrate rewriting statements into Java. After the compilation, every Tom statement is translated into Java and we obtain a standalone Java program. In this section, we show how Tom pattern-matching features have been the key of the completion implementation with $\phi(\mathcal{R})$. See [3] for more details about the Tom language features.

4.1 General process

In Figure 1, we present the general process which leads to the implementation of the completion. In order to compare easily our implementation with Timbuk, we use a similar input format.

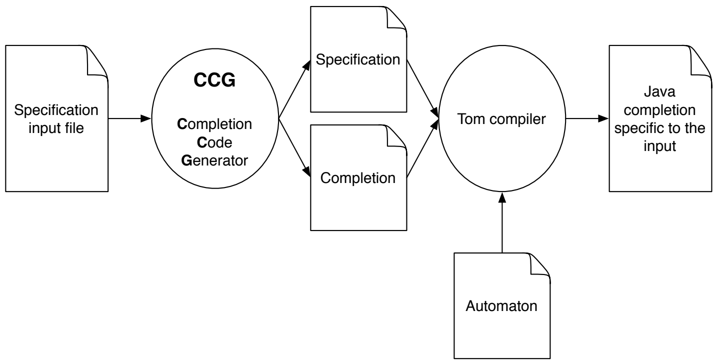

Fig. 1. CCG application: from a Specification to its dedicated Completion Program

For a given Timbuk specification, the application CCG generates a Tom program of the completion algorithm dedicated to this specification. Actually, this program is composed of three files. The file `Specification` defines the algebraic signature and the file `Completion` describes the completion implementation. These two files are specific to the input specification file. The last file `Automaton` is generic and corresponds to a Tom program in which all data structures handling tree automata can be found. These Tom files are finally compiled into Java files providing an efficient application dedicated to the completion on the given Timbuk specification.

To present how completion is encoded in Tom, we consider again the Example 2 in the following sections.

4.2 The `Specification` file

Tom provides several constructs to manipulate algebraic structures. In particular, it is possible to directly define an algebraic signature and from this signature, Tom generates a typed term structure that can be directly used by a Java programmer.

For the Example 2, the signature generated by CCG in the file `Specification` contains the set of symbols $\mathcal{F}$, a new constructor denoted `q(int)`, for specifying states. The transitions are represented using the constructor `transition` of arity 2. Each context symbol C_t of the set $\mathcal{F}'$ introduced by $\phi(\mathcal{R})$ is denoted `C_i`. The signature also contains a variadic operator called `sons` which is used to represent *expanded states*. The expanded states of a state `q` corresponds to the list, built using the variadic operator `sons`, of all `t_i` such that `transition(t_i,q)` is a transition of the current automaton. For example, for the tree automaton given in Example 2, the expanded state of q_0, denoted `q(0)` in the implementation, is `sons(g(q(1),q(2)), g(q(1),q(3)))`.

Below, we give the signature generated by our compiler CCG, for the automaton $\mathcal{A}$ and the set of rules $\mathcal{R}$ given in Example 2. The constructors C_a, $C_{f(a,x)}$, $C_{h(y)}$ and $C_{g(f(a,x),h(y))}$ have been respectively renamed into `C1`, `C2`, `C3` and `C4`.

```
Term = f(Term,Term)
     | g(Term,Term)
     | h(Term)
     | a() | b() | c() | d()
     | q(int)
     | sons(Term*)
     | C1() | C2(Term) | C3(Term)| C4(Term,Term)
     | transition(config:Term,state:Term)
```

4.3 The `Automaton` file

A tree automaton is an object of the class `Automaton`. This class is mainly composed of two fields: `transitionsByFunctionSymbol` and `expandedForms`, both of sort `HashTable`. The keys of the former are the function symbols of the generated signature and its values are the sets of transitions. Given a function symbol `f` (a key), the corresponding value is the set composed of transitions whose left-hand side is built from `f`. The latter stores the expanded form of states. Both hash-tables are updated during the completion process. Another field `newTransitions` stores the new transitions built by a completion step. Both data structures `expandedForms` and `transitionsByFunctionSymbol` are updated according to this field when the method `update()` implemented in this class is invoked. This method resets the set `newTransitions` to the empty set. There are also other fields specifying the final states, handling the new states introduced and so on.

4.4 The `Completion` file

The code of this class is automatically generated from the automaton $\mathcal{A}$ and the TRS $\mathcal{R}$ given in the input specification. This class contains one field `currentA` of sort `Automaton` representing the current automaton, as well as methods to implement the completion with $\phi(R)$.

In Example 2, the main method `completeAllSteps` iterates by applying every rule and updating `currentA` until reaching a fix-point (if it exists). The function `hasNewTransitions()` returns `true` if the set `newTransitions` is not empty. An execution of this algorithm is detailed in Appendix A. Note that for each rule of $\phi(\mathcal{R})$, a method `completeStepWithRule[i]` is generated. Such a method performs one completion step for a given rule.

```
public void completeAllSteps(){
  do {
    // Current automaton update
     currentA.update();
    // Completion step with a -> C1
     completeStepWithRule1();
    // Completion step with f(C1,x) -> C2(x)
     completeStepWithRule2();
    // Completion step with h(y) -> C3(y)
     completeStepWithRule3();
    // Completion step with g(C2(x),C3(y)) -> C4(x,y)
     completeStepWithRule4();
    // Completion step with C4(x,y) -> g(f(a,f(a,x)),h(h(y)))
     completeStepWithRule5();
  } while(currentA.hasNewTransitions());
}
```

We recall that, to perform a completion step, we need to find, for each rule $l \to r \in \phi(\mathcal{R})$, all substitutions $\sigma : \mathcal{X} \to \mathcal{Q}$ such that $l\sigma \to_{\mathcal{A}}^{*} q$ and $r\sigma \not\to_{\mathcal{A}}^{*} q$. Here, we use uttermost Tom matching on variadic operators to implement efficiently this operation. This is possible because in $\phi(\mathcal{R})$, the left-hand sides are almost flat (see the definition in Section 2.1).

In particular, according to Definition 6, l can be only of the form

(1) $f(t_1, \ldots, t_n)$ with $t_i \in \mathcal{X}$ or $t_i = C_k(x_1, \ldots, x_m)$, for $i \in [1, n]$, $C_k \in \mathcal{F}'$, and $x_1, \ldots, x_m \in \mathcal{X}$, or
(2) $C_{k'}(x_1, \ldots, x_n)$, $C_{k'} \in \mathcal{F}'$ and $x_1, \ldots, x_n \in \mathcal{X}$.

When l is of the form (2), finding a substitution σ such that $l\sigma \to_{\mathcal{A}}^{*} q$ consists of looking for every transition of $\mathcal{A}$ of the form $C_{k'}(q_1, \ldots, q_n) \to q$ and matching their left-hand side with $C_{k'}(x_1, \ldots, x_n)$.

When l is of the form (1), every substitution σ satisfying $l\sigma \rightarrow^*_{\mathcal{A}} q$ is such that $\exists q_1, \ldots q_n$, n states of $\mathcal{A}$, for which $\forall t_i$, either $t_i\sigma = q_i$, or $t_i\sigma \rightarrow q_i$ is a transition of $\mathcal{A}$, and $f(q_1, \ldots, q_n) \rightarrow q$ is a transition of $\mathcal{A}$. In the following we show how this operation can be realized using list matching on variadic operators.

Let l' be $l = f(t_1, \ldots, t_n)$ where every t_i of the form $C_k(x_1, \ldots, x_m)$ has been replaced by the Tom pattern `sons(_*,C_k(x_1,...,x_m),_*)`. Let us consider a transition $f(q_1, \ldots, q_n) \rightarrow q \in \mathcal{A}$. Remember that we look for σ such that $f(t_1, \ldots, t_n)\sigma \rightarrow^*_{\mathcal{A}} q$. Such a σ exists if $t_1\sigma \rightarrow^*_{\mathcal{A}} q_1, \ldots, t_n\sigma \rightarrow^*_{\mathcal{A}} q_n$. Thanks to our encoding using `sons`, we can expand every q_i such that the corresponding t_i in l is not a variable. Finding σ simply consists of matching l' against the expanded form of $f(q_1, \ldots, q_n)$. Moreover, when the right-hand side r is of the form $C_k(x_1, \ldots, x_m)$, verifying that $r\sigma \not\rightarrow^*_{\mathcal{A}} q$ consists just of checking that $r\sigma$ is in the expanded form of q.

Let us consider the following rule $g(C_{f(a,x)}(x), C_{h(y)}(y)) \rightarrow C_{g(f(a,x),h(y))}(x, y)$. Its completion method is `completeStepWithRule4` and it is implemented in Tom as follows.

```
public void completeStepWithRule4() {
  for (Term tr: currentA.getExpectedTransitions("g")) {
    Term t = tr.getconfig();
    Term te = t.expandForRule4(currentA);
    Term q = tr.getstate();
    Term qe = q.expand(currentA);
    %match(te) {
      g(sons(_*,C2(x),_*),sons(_*,C3(y),_*)) -> {
        if (! qe.contains(‘C4(x,y)) {
          currentA.addNewTransitions(‘transition(C4(x,y),q));
        }
      }
    }
  }
}
```

The method `getExpectedTransitions("g")` returns a set of transitions whose topmost function symbol of its left-hand side is `g`. The function symbol `g` is the symbol occurring at the root of the left-hand side of the considered rule i.e. $g(C_{f(a,x)}(x), C_{h(y)}(y)) \rightarrow C_{g(f(a,x),h(y))}(x, y)$. The instruction `tr.getconfig()` (resp. `tr.getstate()`) returns the value stored in the first (resp. second) subterm of `tr` (see the signature in Section 4.2). According to the current tree automaton, the function `expand` returns the expanded form of a state and the function `expandForRule4` builds the expanded form of `t` required for this rule. As in this rule, each child of the left-hand side is of the form `C_k(...)`, this form corresponds to `t` where each child (state) has been replaced by its expanded form.

In this function, two new Tom constructs are used. The first one is the ‘ (back-quote), whose action is to build a term in memory from the algebraic signature described in Section 4.3. For instance, the last instruction of the method `completeStepWithRule4` builds the ground term `‘transition(C4(x,y),q)` (with the variables `x` and `y` at this program point) and stores it into the set `newTransitions` of `currentA`.

The second construct provided by Tom is the `%match`, which executes an action associated to a pattern, when this one matches the subject. In the example above, the first part of the pattern `sons(_*,C2(x),_*)` means that we try to find each element of the list, placed under the symbol sons, which is of the form `C2(x)`. The second part `sons(_*,C3(y),_*))` is interpreted similarly. The complete pattern `g(sons(_*,C2(x),_*),sons(_*,C3(y),_*))` matches for every couple of elements of the form `C2(x)` and `C3(y)` and computes the corresponding substitution for `x` and `y`. At the right-hand side of this pattern, there is a Java action that calls the method `addNewTransitions` on `currentA`. `addNewTransitions` takes as parameter either a transition or a collection of transitions and adds it (or them) into the set `newTransitions`. This action is executed as many times as the pattern matches.
Note that the methods `completeStepWithRule1`, `completeStepWithRule2` and `completeStepWithRule3` can be defined similarly.

Below, we consider the method `completeStepWithRule5` corresponding to the rule `C4(x,y) -> g(f(a,f(a,x)),h(h(y)))`. This method differs from the other ones mainly because the right-hand side is neither flat, nor almost flat. So to test if $r\sigma \not\rightarrow^*_{\mathcal{A}} q$, we use the function `reduceIn(t,q)` that returns `true` if `t` can be reduced to `q` by the current automata `currentA`. Moreover, the resulting transition must be normalized using the approximation rules `N`. Note that this normalization is not necessary for the other rules because the resulting transitions are already normalized.

```
public void completeStepWithRule5() {
  for (Term tr: currentA.getExpectedTransitions("C4")) {
    Term t = tr.getconfig();
    Term te = t.expandForRule5();
    Term q = tr.getstate();
    %match(te) {
      C4(x,y) -> {
        if (! currentA.reduceIn(‘g(f(a,f(a,x)),h(h(y))),q)) {
          currentA.addANewTransitions(Norm(N,
                ‘transition(g(f(a,f(a,x)),h(h(y))),q)));
        }
      }
    }
  }
}
```

In this example, as no child of the rule left-hand side is of the form `C_k(...)`, `expandForRule5` returns simply `t`.

Thus, the files `Completion`, `Specification` and `Automaton` are generated and compiled with Tom. The resulting Java files are then compiled and the file `Completion.class` can be executed using the command `java`.

5 Experiments

We give in this section significant examples to demonstrate the efficiency of this technique. In the table below, the automaton size is given as (nb of transitions / nb of states). The benchmarks were done on a intel based platform (2×Pentium 3GHz, running under FreeBSD), using Tom 2.5, Timbuk 2.2 and Java 1.5.

	Combinatory	NSPK	View-Only	Java prog. 1	Java prog. 2
TRS size (nb of rules)	1	13	15	279	303
Initial Automaton size	43 / 23	14 / 4	21 / 18	26 / 49	33 / 33
Tom:					
Final Automaton size	8043 / 23	171 / 21	938 / 89	1974 / 637	1611 / 672
Time (secs)	**5.9**	**5.9**	**150**	**360**	**303**
Timbuk:					
Final Automaton size	8043 / 23	151 / 16	730 / 74	1127 / 334	751 / 335
Time (secs)	**51.1**	**19.7**	**6420**	**25266**	**37387**

Combinatory example: This tiny example emphasizes that our prototype is better than Timbuk in particular when a large number of substitutions is computed during the completion. Let $\mathcal{R} = \{g(f(x_1), h(h(h(x_2, x_3), x_4), x_5)) \to u(x_1, x_2, x_3, x_4, x_5)\}$ and $\mathcal{A}$ be the tree automaton whose transition set is the following: $\{nil \to q_h, f(q_{a_1}) \to q_f, g(q_f, q_h) \to q_g\} \cup \{t \to q_t, h(q_h, q_t) \to q_h \mid t \in \{a_i, b_i, c_i, d_i \mid i = 1, \ldots, 5\}\}$. For the variables x_1, x_3, x_4 and x_5 there are twenty possible instantiations during the completion. The variables x_1 and x_2 take only and respectively the values q_{a_1} and q_h. So, there are 20^3 transitions to compute by completion.

Needham-Schröeder Public Key Protocol: NSPK is a security protocol whose goal is to ensure the mutual authentication of two participants. The first version established in 1976 has been corrected by G. Lowe in 1995. Indeed, in this first version, a man-in-the-middle attack was possible. The second version of NSPK was already verified using Timbuk in [14]. Using the same approximation rules, our prototype leads also to an over-approximation allowing us to verify this protocol. The computation time of our prototype is better than Timbuk was.

The View-Only protocol: Let us now focus on the *View-Only* protocol. This protocol is a component of the *Smartright* system [20] designed by Thomson. In the context of home digital network, this system prevents users from unlawfully copying movie broadcasts on the network. The *view-only* devices are a decoder (DC) and a digital TV set (TVS). They share a secret key Kab securely sealed in both of them. The goal of this protocol is to periodically change a secret control word (CW) enabling to decode the current broadcast program. The Timbuk specification of this protocol is described in [15]. The same properties have been successfully verified with our encoding i.e. secrecy of CW, authentication of CW (no control word sent by the intruder has been accepted) and no replay attack on CW. The fix-point automaton has been obtained within a couple of minutes, while Timbuk terminates within 107 minutes.

Java programs: In [5], Java program analysis is performed using approximations, i.e. tree automata completion. Starting from a Java byte code program P, a TRS encoding the Java Virtual Machine and the semantics of P is automatically produced. The Java program 1 is the one detailed in [5]. For this program, a fix-point automaton is obtained with Tom within 360 seconds whereas it takes several hours for Timbuk to obtain the result. On this fix-point, the same analysis as in [5] have been successfully performed.

The Java program 2 represents the construction of two linear chained lists of integers. One is supposed to contain only positive integers, and the other only negative ones. Integers are entered and stored in the corresponding list while their value is different from 0. For verifying this program, we define approximation rules in such a way that all integers are abstracted into three equivalence classes (equal to 0, strictly positive and strictly negative), the input stream is specified as an infinite stack of integers and abstractions are performed on the memory in order to handle an infinite number of object creations. On both fix-point automaton, we can conclude that there are: no positive integer in the list of negative ones and no negative integer in the list of positive ones.

Our experiments show that the new implementation is faster than the last version of Timbuk. And this is the case for examples dealing with numerous combinations and computations. It is often the case when we are dealing with approximations. We have also applied the transformation ϕ on the input TRS of the Timbuk tool. Thanks to these experiments, we are convinced that ϕ is not the only reason for our better performance. In Timbuk, the most time consuming operation is the computation of critical pairs. Indeed, for every rule $l \to r$, we need to find every ground instance of l which can be reduced to a state q of the current automaton. In [12], a solution based on tree automata intersections is proposed but it remains inefficient when the number of rules and the size of their left-hand side are huge. In our case, as CCG generates a completion algorithm dedicated to a given specification, we do not need to implement a general matching algorithm as in Timbuk. Moreover, since the left-hand sides of $\phi(\mathcal{R})$ rules are almost flat, we use only the Tom pattern-matching features to compute critical pairs.

6 Conclusion

In this paper, we have developed an original and efficient implementation of tree automata completion. The first contribution is to have shown that decomposing the TRS into smaller rules preserves the over-approximation property. The second contribution is to have shown that, because of the special form of the decomposed rules, it is possible to define completion in a non-standard way. Instead of sophisticated and heavy algorithms over tree automata, our completion is built using simple rewriting techniques. Finally, another contribution of this paper is to propose an implementation taking advantage of the list-matching compilation feature of Tom to greatly improve the efficiency. On Java programs, which are now our main concerns, the obtained results are up to 100 times faster than the current state of the art implementation, i.e. Timbuk. The implementation proposed in this paper is a first promising step towards efficient verification tools for infinite state systems. We plan to apply this tool to the static analysis of industrial Java applications in the context of the RAVAJ project [1]. Since Tom generates *thread-safe code* – code supporting simultaneous execution by multiple threads – we are currently studying a multi-threaded implementation of the completion. This could also be a way to greatly improve the overall performance of our tree automata completion tool.

References

1. RAVAJ project (ANR-06-SETI-014)– Rewriting and Approximations for Java Applications Verification. `http://www.irisa.fr/lande/genet/RAVAJ/index.html`.
2. F. Baader and T. Nipkow. *Term Rewriting and All That.* Cambridge University Press, 1998.
3. E. Balland, P. Brauner, R. Kopetz, P.-E. Moreau, and A. Reilles. Tom: Piggybacking rewriting on java. In *Proceedings of the 18th Conference on Rewriting Techniques and Applications*, volume 4533 of *LNCS*, pages 36–47. Springer-Verlag, 2007.
4. E. Balland and P.-E. Moreau. Optimizing pattern matching compilation by program transformation. In *3rd Workshop on Software Evolution through Transformations (SeTra'06)*. Electronic Communications of EASST, 2006. ISSN 1863-2122.
5. Y. Boichut, T. Genet, T. Jensen, and L. Le Roux. Rewriting Approximations for Fast Prototyping of Static Analyzers. In *Proceedings of the 18th Conference on Rewriting Techniques and Applications*, volume 4533 of *LNCS*, pages 48–62, 2007.
6. Y. Boichut, P.-C. Héam, and O. Kouchnarenko. Automatic Verification of Security Protocols Using Approximations. Research Report RR-5727, INRIA-Lorraine - CASSIS Project, October 2005.
7. M. Clavel, M. Palomino, and A. Riesco. Introducing the ITP tool: a tutorial. *J. UCS*, 12(11):1618–1650, 2006.
8. H. Comon, M. Dauchet, R. Gilleron, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi. Tree automata techniques and applications. `http://www.grappa.univ-lille3.fr/tata/`, 2002.
9. 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. (North-Holland), 1990. Also as: Research report 478, LRI.
10. S. Eker, J. Meseguer, and A. Sridharanarayanan. The Maude LTL Model Checker and its Implementation. In *SPIN*, volume 2648 of *LNCS*, pages 230–234. Springer, 2003.
11. A. Farzan, C. Chen, J. Meseguer, and G. Rosu. Formal analysis of java programs in javafan. In *CAV*, volume 3114 of *LNCS*, pages 501–505. Springer, 2004.
12. G. Feuillade, T. Genet, and V. Viet Triem Tong. Reachability Analysis over Term Rewriting Systems. *JAR*, 33 (3-4):341–383, 2004.
13. T. Genet. Decidable approximations of sets of descendants and sets of normal forms. In *Proc. 9th RTA Conf., Tsukuba (Japan)*, volume 1379 of *LNCS*, pages 151–165. Springer-Verlag, 1998.
14. T. Genet and F. Klay. Rewriting for Cryptographic Protocol Verification. In *Proc. 17th CADE Conf., Pittsburgh (Pen., USA)*, volume 1831 of *LNAI*. Springer-Verlag, 2000.
15. T. Genet, Y.-M. Tang-Talpin, and V. Viet Triem Tong. Verification of Copy Protection Cryptographic Protocol using Approximations of Term Rewriting Systems. In *In Proceedings of Workshop on Issues in the Theory of Security*, 2003.
16. T. Genet and V. Viet Triem Tong. Timbuk 2.0 – a Tree Automata Library. IRISA / Université de Rennes 1, 2000. `http://www.irisa.fr/lande/genet/timbuk/`.
17. R. Gilleron and S. Tison. Regular tree languages and rewrite systems. *Fundamenta Informaticae*, 24:157–175, 1995.
18. J. Meseguer and G. Rosu. Rewriting logic semantics: From language specifications to formal analysis tools. In *IJCAR*, pages 1–44, 2004.
19. T. Takai. A Verification Technique Using Term Rewriting Systems and Abstract Interpretation. In *Proc. 15th RTA Conf., Aachen (Germany)*, volume 3091 of *LNCS*, pages 119–133. Springer, 2004.
20. Thomson. Smartright technical white paper v1.0. Technical report, october 2001. `http://www.smartright.org`.

A Implementation of Example 2: detailed steps

Let us detail how the functions generated for the example 2 are applied. The transitions are of the form $f(q_1, \ldots, q_n) \rightarrow q_m$ and then their Tom representations are of the form `transition(f(q(1),...,q(n)),q(m))`. In the following, to make the presentation simpler, a state `q(i)` is denoted $\mathtt{q_i}$. Similarly, `C_k` is denoted $\mathtt{C_k}$. Let us start with the tree automaton $\mathcal{A}$ given in Example 1.

Step 1. First, the method `completeStepWithRule1` applies the rule $a \rightarrow C_1$ on the transition $\mathtt{transition}(\mathtt{a}(), \mathtt{q_4})$. As in the classical completion, the transition $\mathtt{transition}(\mathtt{C_1}(), \mathtt{q_4})$ is added to the set `newTransitions`. At the same time, we can apply the third rule $h(y) \rightarrow C_3(y)$ on the transitions $\mathtt{transition}(\mathtt{h}(\mathtt{q_6}), \mathtt{q_2})$ and $\mathtt{transition}(\mathtt{h}(\mathtt{q_7}), \mathtt{q_3})$ with the method `completeStepWithRule3`. Indeed the configurations $\mathtt{h}(\mathtt{q_6})$ and $\mathtt{h}(\mathtt{q_7})$ are not changed by `expandForRule3` and match with $\mathtt{h}(\mathtt{y})$. Thus, $\mathtt{transition}(\mathtt{C_3}(\mathtt{q_6}), \mathtt{q_2})$ and $\mathtt{transition}(\mathtt{C_3}(\mathtt{q_7}), \mathtt{q_3})$ are added to the set `newTransitions`. The rules 2, 4 and 5 cannot be executed since the symbols $\mathtt{C_1}$, $\mathtt{C_2}$, $\mathtt{C_3}$ and $\mathtt{C_4}$ do not occur in any transitions of $\mathcal{A}$. The set `newTransitions` is not empty, so a new iteration is started. The instruction `currentA.update()` is executed and consequently, the expanded form of $\mathtt{q_2}$, $\mathtt{q_3}$ and $\mathtt{q_4}$ are updated and equal to $\mathtt{sons}(\mathtt{h}(\mathtt{q_6}), \mathtt{C_3}(\mathtt{q_6}))$, $\mathtt{sons}(\mathtt{h}(\mathtt{q_7}), \mathtt{C_3}(\mathtt{q_7}))$ and $\mathtt{sons}(\mathtt{C_1}(), \mathtt{a}())$ respectively.

Step 2. The rules 1 and 3 can be applied a new, but no new transitions are added since they have been added at the previous completion step already. Now, the rule 2 can be applied. Indeed, `expandForRule2`, for $\mathtt{transition}(\mathtt{f}(\mathtt{q_4}, \mathtt{q_5}), \mathtt{q_1})$, transforms $\mathtt{f}(\mathtt{q_4}, \mathtt{q_5})$ into $\mathtt{f}(\mathtt{sons}(\mathtt{C_1}(), \mathtt{a}()), \mathtt{q_5})$ Thus, $\mathtt{transition}(\mathtt{C_2}(\mathtt{q_5}), \mathtt{q_1})$ is added into `newTransitions`. The rules 4 and 5 cannot be applied yet. At the end of iteration, the set `newTransitions` is not empty, so a new iteration is started. The instruction `currentA.update()` is executed and consequently, the expanded form of $\mathtt{q_1}$ is updated and equal to $\mathtt{sons}(\mathtt{f}(\mathtt{q_4}, \mathtt{q_5}), \mathtt{C_2}(\mathtt{q_5}))$.

Step 3. We can now apply the rule 4 i.e. $g(C_2(x), C_3(y)) \rightarrow C_4(x, y)$. By applying the function `expandForRule4` the configurations $\mathtt{g}(\mathtt{q_1}, \mathtt{q_2})$ and $\mathtt{g}(\mathtt{q_1}, \mathtt{q_3})$ are transformed into $\mathtt{g}(\mathtt{sons}(\mathtt{f}(\mathtt{q_4}, \mathtt{q_5}), \mathtt{C_2}(\mathtt{q_5})), \mathtt{sons}(\mathtt{h}(\mathtt{q_6}), \mathtt{C_3}(\mathtt{q_6})))$ (resp. $\mathtt{g}(\mathtt{sons}(\mathtt{f}(\mathtt{q_4}, \mathtt{q_5}), \mathtt{C_2}(\mathtt{q_5})), \mathtt{sons}(\mathtt{h}(\mathtt{q_7}), \mathtt{C_3}(\mathtt{q_7}))))$. Therefore $\mathtt{transition}(\mathtt{C_4}(\mathtt{q_5}, \mathtt{q_6}), \mathtt{q_0})$ and $\mathtt{transition}(\mathtt{C_4}(\mathtt{q_5}, \mathtt{q_7}), \mathtt{q_0})$ are added to the set `newTransitions`. Since `newTransitions` is not empty, a new iteration starts. The expanded form of $\mathtt{q_0}$ is updated and equal to $\mathtt{sons}(\mathtt{g}(\mathtt{q_1}, \mathtt{q_2}), \mathtt{g}(\mathtt{q_1}, \mathtt{q_3}), \mathtt{C_4}(\mathtt{q_5}, \mathtt{q_6}), \mathtt{C_4}(\mathtt{q_5}, \mathtt{q_7})$.

Step 4. So at this new iteration, the last rule can thus be applied with these two last transitions. By applying the function `completeStepWithRule5` the configurations $\mathtt{C_4}(\mathtt{q_5}, \mathtt{q_6})$ and $\mathtt{C_4}(\mathtt{q_5}, \mathtt{q_7})$ match with $\mathtt{C_4}(\mathtt{x}, \mathtt{y})$ and after normalization with the approximations rules (with the function `Norm` the following transitions are added to the set `newTransitions`: $\mathtt{transition}(\mathtt{a}, \mathtt{q_4})$, $\mathtt{transition}(\mathtt{f}(\mathtt{q_4}, \mathtt{q_5}), \mathtt{q_5})$, $\mathtt{transition}(\mathtt{h}(\mathtt{q_6}), \mathtt{q_6})$, $\mathtt{transition}(\mathtt{h}(\mathtt{q_7}), \mathtt{q_6})$ and $\mathtt{transition}(\mathtt{g}(\mathtt{q_5}, \mathtt{q_6}), \mathtt{q_0})$.

Step 5. The current automaton is not a fix-point yet. The fix-point is obtained one completion step later after having added the transitions `transition`($\mathtt{C}_3$($\mathtt{q}_6$), $\mathtt{q}_6$) and `transition`($\mathtt{C}_3$($\mathtt{q}_7$), $\mathtt{q}_6$) obtained by executing `completeStepWithRule3` and the transition `transition`($\mathtt{C}_2$($\mathtt{q}_5$), $\mathtt{q}_5$) resulting of the execution of the method `completeStepWithRule2`.

Note that the tree automaton $\mathcal{A}^2_{N,\mathcal{R}}$ computed in Example 2 is included in this fix-point automaton. So, it is actually an over-approximation of $\mathcal{R}^*(\mathcal{L}(\mathcal{A})) = \{g(f^n(a,b), h^n(\{c,d\})) \mid n \geq 0\}$.