INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Reachability Analysis of Term Rewriting Systems with Timbuk

- extended version -

Thomas Genet , Valérie Viet Triem Tong

N°4266

Septembre 2001

THÈME 2

Rapport de recherche

ISSN 0249-6399 ISRN INRIA/RR--4266--FR+ENG

Reachability Analysis of Term Rewriting Systems with Timbuk

- extended version -

Thomas Genet , Valérie Viet Triem Tong

Thème 2 — Génie logiciel
et calcul symbolique
Projet Lande

Rapport de recherche n° 4266 — Septembre 2001 — 16 pages

Abstract: We present Timbuk – a tree automata library – which implements usual operations on tree automata as well as a completion algorithm used to compute an over-approximation of the set of descendants $\mathcal{R}^*(E)$ for a regular set E and a term rewriting system $\mathcal{R}$, possibly non linear and non terminating. On several examples of term rewriting systems representing programs and systems to verify, we show how to use Timbuk to construct their approximations and then prove unreachability properties of these systems.

Key-words: Timbuk, Term Rewriting, Reachability, Tree Automata, Descendants, Program Verification

(Résumé : tsvp)

Unité de recherche INRIA Rennes
IRISA, Campus universitaire de Beaulieu, 35042 RENNES Cedex (France)
Téléphone : 02 99 84 71 00 - International : +33 2 99 84 71 00
Télécopie : 02 99 84 71 71 - International : +33 2 99 84 71 71

Analyse d'atteignabilité dans les systèmes de réécriture avec Timbuk

Résumé : Nous présentons Timbuk– une librairie d'automates d'arbres – qui implémente les opérations classiques sur les automates d'arbres ainsi qu'un algorithme de complétion utilisé pour calculer une sur-approximation de l'ensemble des descendants $\mathcal{R}^*(E)$ pour un ensemble régulier de termes E et un système de réécriture $\mathcal{R}$ (éventuellement non linéaire et non terminant). Sur des exemples de systèmes de réécriture représentant des programmes ou des systèmes à vérifier, nous montrons ensuite comment utiliser Timbuk pour construire leurs approximations et prouver la non-atteignabilité d'ensembles de termes proscrits.

Mots-clé : Timbuk, Réécriture, atteignabilité, automates d'arbres, descendants, vérification de programmes

Contents

Introduction

Term Rewriting Systems (TRSs for short) are a very simple way to describe functions as well as parallel processes or state transition systems where rewriting models respectively evaluation, progression or transitions. Rewriting specifications can generally be executed using a rewriting tool [2, 3] and verified by proving some properties like termination, confluence and some inductive theorems, where confluence and inductive theorems proofs generally require the termination property.

In [7], we proposed a technique for approximating the set of descendants: given a TRS $\mathcal{R}$ and two regular sets of terms E and F both recognized by tree automata, approximation of the set of descendants $\mathcal{R}^*(E)$ permits, in particular, to show the non $\mathcal{R}$-reachability of terms of F from terms of E. One of the main difference with other existing proof techniques on TRSs is that approximations can be computed on TRSs even if they are non terminating (and non confluent). With regards to regular approximations used in abstract interpretation, our method does not focus on automation, but instead, it lets the user adapt its approximation rules to the TRS and the property he wants to verify. Thus, regular approximations can be more precise at the price of requiring user interaction. These aspects turns out to be of interest and have some practical applications like for the verification of cryptographic protocols [8].

The approximation technique, initially prototyped with ELAN [2], is now implemented in the Timbuk library [9], written in Ocaml [14], a programming language of the ML family. This library provides basic primitives on non deterministic tree automata like intersection, union, complement of languages, determinisation of tree automata, as well as a *completion* algorithm for computing approximations. This library can be called from some other Ocaml code, interfaced with C, or used as a simple calculator on tree automata at the top-level of Ocaml. The library also contains a simple executable approximation front end (parsing a specification file and starting an approximation computation) that we used to work on the examples contained in this paper.

In this paper, we briefly recall the basic definitions of TRSs and tree automata in section 1. Then, in section 2, we recall what approximations are. The construction of approximation in practice with Timbuk is detailed in section 3. And finally, we conclude on some comparisons with other works and systems in section 4.

1 Preliminaries

Comprehensive surveys can be found in [6, 1] for term rewriting systems, in [4, 10] for tree automata and tree language theory.

Let $\mathcal{F}$ be a finite set of symbols, each associated with an arity function, 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 mapping σ from $\mathcal{X}$ into $\mathcal{T}(\mathcal{F},\mathcal{X})$, which can uniquely be extended to an endomorphism of $\mathcal{T}(\mathcal{F},\mathcal{X})$. Its domain $\mathcal{D}om(\sigma)$ is $\{x \in X \mid x\sigma \neq x\}$.

A term rewriting system $\mathcal{R}$ is a set of *rewrite rules* $l \to r$, where $l, r \in \mathcal{T}(\mathcal{F}, \mathcal{X})$, $l \notin \mathcal{X}$, and $\mathcal{V}ar(l) \supseteq \mathcal{V}ar(r)$. A rewrite rule $l \to r$ is *left-linear* (resp. *right-linear*) if each variable of l (resp. r) occurs only once. A rule is linear if it is both left and right-linear. A TRS $\mathcal{R}$ is linear (resp. left-linear, right-linear) if every rewrite rule $l \to r$ of $\mathcal{R}$ is linear (resp. left-linear, right-linear). The TRS $\mathcal{R}$ induce a rewriting relation $\to_{\mathcal{R}}$ on terms whose reflexive transitive closure is denoted by $\to_{\mathcal{R}}^*$. The set of $\mathcal{R}$-descendants of a set of ground terms E is $\mathcal{R}^*(E) = \{t \in \mathcal{T}(\mathcal{F}) \mid \exists s \in E \text{ s.t. } s \to_{\mathcal{R}}^* t\}$.

Let $\mathcal{Q}$ be a finite set of symbols, with arity 0, called *states*. $\mathcal{T}(\mathcal{F} \cup \mathcal{Q})$ is called the set of *configurations*. A *transition* is a rewrite rule $c \to q$, where $c \in \mathcal{T}(\mathcal{F} \cup \mathcal{Q})$ and $q \in \mathcal{Q}$. A *normalized transition* is a transition $c \to q$ where $c = q' \in \mathcal{Q}$ or $c = f(q_1, \ldots, q_n)$, $f \in \mathcal{F}$, $ar(f) = n$, and $q_1, \ldots, q_n \in \mathcal{Q}$. 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 $\mathcal{Q}_f \subseteq \mathcal{Q}$ and Δ is a set of normalized transitions. A tree automaton is *deterministic* if there are no two rules with the same left-hand side. The rewriting relation induced by the transitions of $\mathcal{A}$ (the set Δ) is denoted by $\to_{\mathcal{A}}$. The tree language recognized by $\mathcal{A}$ is $\mathcal{L}(\mathcal{A}) = \{t \in \mathcal{T}(\mathcal{F}) \mid \exists q \in \mathcal{Q}_f \text{ s.t. } t \to_{\mathcal{A}}^* q\}$.

2 Approximations

Starting from a tree automaton $\mathcal{A}_0 = \langle \mathcal{F}, \mathcal{Q}_0, \mathcal{Q}_f, \Delta_0 \rangle$ and a left-linear[1] TRS $\mathcal{R}$, the aim of the approximation algorithm is to compute a tree automaton $\mathcal{A}_k$ such that $\mathcal{L}(A_k) \supseteq \mathcal{R}^*(\mathcal{L}(\mathcal{A}_0))$. Approximations are used to show that terms recognized by a tree automaton $\mathcal{A}_{bad}$ are not reachable by rewriting terms of $\mathcal{L}(\mathcal{A}_0)$ with $\mathcal{R}$. For this, it is enough to show that $\mathcal{L}(\mathcal{A}_k) \cap \mathcal{L}(\mathcal{A}_{bad}) = \emptyset$ i.e., compute the automaton recognizing the intersection and show that the recognized language is empty.

The technique consists in successively computing tree automata $\mathcal{A}_1$, $\mathcal{A}_2$, … such that $\forall i \geq 0 : \mathcal{L}(\mathcal{A}_i) \subset \mathcal{L}(\mathcal{A}_{i+1})$ and if $s \in \mathcal{L}(\mathcal{A}_i)$, such that $s \to_{\mathcal{R}} t$ then $t \in \mathcal{L}(A_{i+1})$, until we get an automaton $\mathcal{A}_k$ with $k \in \mathbb{N}$ such that $\mathcal{L}(\mathcal{A}_k) = \mathcal{L}(\mathcal{A}_{k+1})$. Thus, $\mathcal{A}_k$ also verifies $\mathcal{L}(\mathcal{A}_k) \supseteq \mathcal{R}^*(\mathcal{L}(\mathcal{A}_0))$. More precisely, to construct $\mathcal{A}_{i+1}$ from $\mathcal{A}_i$, we achieve a *completion step* which consists in finding *critical pairs* between $\to_{\mathcal{R}}$ and $\to_{\mathcal{A}_i}$. For a substitution σ and a rule $l \to r \in \mathcal{R}$, a critical pair is an instance $l\sigma$ of l such that there exists $q \in \mathcal{Q}$ satisfying $l\sigma \to_{\mathcal{A}_i}^* q$ and $r\sigma \not\to_{\mathcal{A}_i}^* q$. For $r\sigma$ to be recognized as the same state, it is enough to join the critical pair:

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

[1]Approximations can also be computed for non left-linear systems with some restrictions on the automaton we consider, see section 3.3.

and add the new transition $r\sigma \to q$ to $\mathcal{A}_{i+1}$. However, the transition $r\sigma \to q$ is not necessarily of the form $f(q_1, \ldots, q_n) \to q'$ and so has to be normalized first. For example, to normalize a transition of the form $f(g(a), h(q')) \to q$, we need to find some states q_1, q_2, q_3 and replace the previous transition by a set of normalized transitions: $\{a \to q_1, g(q_1) \to q_2, h(q') \to q_3, f(q_2, q_3) \to q\}$.

Assume that q_1, q_2, q_3 are new states, then adding the transition itself or its normalized form does not make any difference. Now, assume that $q_1 = q_2$, the normalized form becomes $\{a \to q_1, g(q_1) \to q_1, h(q') \to q_3, f(q_1, q_3) \to q\}$. This set of normalized transitions represents the regular set of non normalized transitions of the form $f(g^*(a), h(q')) \to q$ which contains the transition we wanted to add initially but also many others. Hence, this is an approximation. We could have made an even more drastic approximation by identifying q_1, q_2, q_3 with q, for instance.

Timbuk provides several techniques to automatise the normalization process. We detail only two of them:

- One is a set of *priority transitions*: these are specific transitions of the automaton which are systematically used to simplify the new transitions before any user normalization is performed. For example, assume that $a \to q_a$ and $g(q_a) \to q_0$ are priority transitions of the automaton, then the transition of the previous example would be simplified into $f(q_0, h(q')) \to q$ before requiring new states to end the normalization. Note that the set of priority transitions has to be defined by the user because using such 'deterministic' transitions (always normalizing the same configuration by the same state) leads to approximation in general[2].

- A second tool for normalizing automatically new transitions are *approximation rules*. Approximation rules are strictly more expressive[3] than priority transitions since they are not only tree automata transitions but rewrite rules with variables. The approximation rules are applied on the new transitions to normalize. The general form for approximation rules is the following: $[s \to x] \to [l_1 \to x_1, \ldots, l_n \to x_n]$ where $[s \to x]$ with $s \in \mathcal{T}(\mathcal{F} \cup \mathcal{Q}, \mathcal{X})$ and $x \in \mathcal{X} \cup \mathcal{Q}$ is a pattern to be matched over the transition to normalize and $[l_1 \to x_1, \ldots, l_n \to x_n]$ are rules used to normalize the left hand side of the new transition. The syntactical constraint for those rules is the following: $l_i \in \mathcal{T}(\mathcal{F} \cup \mathcal{Q}, \mathcal{X})$ and either $x_i \in \mathcal{Q}$ or $x_i \in \mathcal{V}ar(l_i) \cup \mathcal{V}ar(s) \cup \{x\}$. To normalize a transition of the form $t \to q'$, we match s on t and x on q', obtain a given substitution σ and then we normalize t with the rewrite rules $l_1\sigma \to r_1\sigma, \ldots, l_n\sigma \to r_n\sigma$ where $r_1\sigma, \ldots, r_n\sigma$ should be some states. For example, normalizing a transition $f(h(q_1), g(q_2)) \to q_3$ with approximation rule $[f(x, g(y)) \to z] \to [g(u) \to z]$ will give a substitution $\sigma = \{x \mapsto h(q_1), y \mapsto q_2, z \mapsto q_3\}$, an instanciated set of rewrite rules $[g(u) \to q_3]$. Thus, $f(h(q_1), g(q_2)) \to q_3$ will be normalized into a normalized transition $g(q_2) \to q_3$ and a partially normalized transition $f(h(q_1), q_3) \to q_3$.

[2]For example, using the set of priority transitions $\{a \to q', b \to q'\}$ to normalize the transition $f(a) \to q$ will give the transition $f(q') \to q$ which is an approximation of the initial transition since it represents the set of non normalized transitions $\{f(a) \to q, f(b) \to q\}$.

[3]One may use approximation rules to simulate priority transitions but the interest of priority transitions lies in the fact they can be added during completion, see section 3.1.

Note that, whatever the normalization may be, a safety theorem of [7] ensures that when the completion terminates on a tree automaton $\mathcal{A}_k$, it is such that $\mathcal{L}(\mathcal{A}_k) \supseteq \mathcal{R}^*(\mathcal{L}(\mathcal{A}_0))$.

3 Computing Approximations with Timbuk

A completion step i with Timbuk, constructing automaton $\mathcal{A}_i$ from automaton A_{i-1}, can be divided into 5 phases:

1. automatically find some new critical pairs between the TRS and $\mathcal{A}_{i-1}$
2. automatically construct the corresponding new transitions,
3. automatically normalize the new transitions by priority transitions and approximation rules,
4. ask the user to provide some rules and possibly new states to normalize what remains to be normalized in the new transitions,
5. automatically construct A_i by adding the normalized new transitions to A_{i-1}.

After each completion step i, the user can choose between several actions (in a menu) like displaying the current automaton A_i, checking the intersection between A_i and some other automata recognizing the terms which should be proven unreachable, and an undo action to come back to the automaton A_{i-1} corresponding to the previous completion step.

3.1 Interactive approximation

In the following introductory example, we compute an approximation of the reverse function (symbol `rev` defined by TRS `R1`) on the regular language of terms recognized by automaton `A0` i.e., `rev` applied to any flat lists of `a`'s and `b`'s where all `a`'s are before `b`'s in the list. The second automaton called `Problems` recognize a regular language of terms that should be unreachable from `A0` by rewriting with `R1`: flat lists where there is at least one '`a`' before a '`b`' in the list. Here is the complete specification file:

```
Ops nil:0 cons:2 app:2 rev:1 a:0 b:0

Vars x y z

TRS R1
   app(x, nil) -> x
   app(cons(x, y), z) -> cons(x, app(y, z))
   rev(nil) -> nil
   rev(cons(x, y)) -> app(rev(y), cons(x, nil))

Automaton A0
States qrev qlab qlb qa qb
Description  qrev: "rev applied to lists where a are before b"
```

```
            qlab: "lists where a are before b (possibly empty)"
            qlb : "lists of b (possibly empty)"
            qa  : "symbol a"
            qb  : "symbol b"
```
Final States `qrev`
Transitions
```
      rev(qlab) -> qrev          nil -> qlab
      cons(qa, qlab) -> qlab     cons(qa, qlb) -> qlab
      nil -> qlb                 cons(qb, qlb) -> qlb
      a -> qa                    b -> qb
```

Automaton `Problems`
States `qlabl qlbl ql qa qb`
Final States `qlabl`
Transitions
```
      cons(qa, qlabl) -> qlabl   cons(qb, qlabl) -> qlabl
      cons(qa, qlbl) -> qlabl    cons(qb, ql) -> qlbl
      cons(qa, ql) -> ql         cons(qb, ql) -> ql
      a -> qa                    b -> qb
      nil -> ql
```

The first completion step gives some new transitions and the following output:

```
Adding transition:
nil -> qrev
... already normalized!

Adding transition:
app(rev(qlab),cons(qa,nil)) -> qrev
Do you want to give by hand some rules to normalize the transition? (y/n)?
```

The first transition is already normalized and automatically added, however the second one has to be normalized. First, we have to find states to place configurations `rev(qlab)` and `nil`. Since the state `qlab` recognizes a list of `a`'s followed by some `b`'s, we intend `rev(qlab)` to be a list of `b`'s followed by some `a`'s, so let us normalize it by a new state called `qlba`. In fact, we define two new states, say `qlba` to normalize `rev(qlab)` and `qnil` to normalize `nil`, by typing the following commands:

```
New States qlba qnil.
* rev(qlab) -> qlba
* nil -> qnil.
```

where the `*` symbol preceding the transitions means that we want to install the following transition in the set of priority transitions. Hence, in the next completion steps, if a new configuration of the form `rev(qlab)` appears, it will be automatically normalized into the state `qlba`. After giving these normalization rules, the transition is still not normalized. Timbuk shows the result of the normalization process so far:

```
Normalization simplifies the transition into: app(qlba,cons(qa,qnil)) -> qrev
Adding transition:
app(qlba,cons(qa,qnil)) -> qrev
```

Once more, we are asked to give some rules for normalizing this transition. Since `cons(qa, qnil)` represents a list with one `a`, we can create a new state `qla` to normalize it, and this terminates the normalization of the first transition. There remains a transition to normalize:

```
Adding transition:
app(rev(qlb),qla) -> qrev
```

Since we intend that the `rev` function applied to any list of `b`'s should give a list of `b`'s we do not need to introduce any new state but simply normalize it by proposing the following priority transition: `* rev(qlb) -> qlb`.

This terminates the first completion step. In the following six completion steps it is enough to successively introduce the following priority transitions to normalize the new transitions we are proposed and thus terminate the completion:

```
* app(qlb, qla) -> qlba        * cons(qb, qnil) -> qlb        * app(qnil, qlb) -> qlb
* app(qnil, qla) -> qla        * rev(qnil) -> qnil            * app(qla, qla) -> qla.
```

Finally, from the menu it is possible to see the completed automaton which now contains 30 transitions and to compute the intersection with the automaton `Problems`:

```
Intersection with Problems gives (the empty automaton):
States
Final States
Transitions
```

meaning that applying `rev` to a list of `a`'s followed by some `b`'s cannot result into any list where there is an '`a`' before a '`b`'.

3.2 Debugging Term Rewriting Systems with Timbuk

The second example describes the behavior of a system of two processes that is supposed to count '+'and '−' symbols in a list. Initially, the list is divided into two parts and each part is given to a single process. One process, let us call it P_+ is supposed to count the '+' symbols and the other one, P_- is supposed to count the '−' symbols. Each process have an incoming message queue. The process P_+ counts '+' symbol in its list, it sends a message to process P_- each time that it finds a '−' symbol, and reads messages in its message queue to take into account the '+' symbols found by process P_-. The behavior of process 'P_- is symmetrical. This behavior can be described by the following TRS, where a term of the form `S(p1, p2, s1, s2)` represent a state of the system where process P_+ is in a configuration described by term `p1`, P_- is in a configuration described by term `p2`, and the message queues for processes P_+ and P_- are respectively `s1` and `s2`. A term of the form `Proc(l, c)` is a process configuration where its current list of symbols is `l` and its local counter is `c`. The terms `o`, `s(o)`, ... represent the naturals $0, 1, \ldots$. Queues are represented by lists where the `add` symbol adds a message in the queue i.e., at the end of the list.

TRS `R1`

```
add(x, nil) -> cons(x, nil)                                      (* add in queue *)
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)         (* P+ counts + *)
S(Proc(cons(minus, y), c), u, m, n) -> S(Proc(y, c), u, m, add(minus, n))
```

```
S(Proc(x, c), z, cons(plus,m), n) -> S(Proc(x, s(c)), z, m, n) (* P+ reads a mesg *)

S(x, Proc(cons(minus, y), c), m, n) -> S(x, Proc(y, s(c)), m, n)          (* P- counts - *)
S(x, Proc(cons(plus, y), c), m, n) -> S(x, Proc(y, c), add(plus, m), n)
S(x, Proc(z, c), m, cons(minus,n)) -> S(x, Proc(z, s(c)), m, n) (* P- reads a mesg *)
```

The initial configuration of the system is described by the following tree automaton recognizing every configuration where the two processes have a counter initialized to zero, any non empty list of symbols to count and an empty message queue.

Automaton `A1`
States `q0 qinit qzero qlist qsymb`
Description

```
          q0        : "the initial configuration"
          qinit     : "a process in an initial state"
          qzero     : "zero"
          qnil      : "the empty list"
          qlist     : "any non empty list of plus and minus symbols"
          qsymb     : "any symbol"
```

Final States `q0`
Transitions

```
      o -> qzero                          nil -> qnil
      plus -> qsymb                       minus -> qsymb
      cons(qsymb, qnil) ->qlist           cons(qsymb, qlist) -> qlist
      Proc(qlist, qzero) -> qinit         S(qinit, qinit, qnil, qnil) -> q0
```

Assume that we want to find the proper conditions for process P_+ (resp. P_-) to terminate without leaving any uncounted '$+$' symbol (resp. '$-$' symbol). When the process ends, it returns the value of its counter: the term `Stop(c)` represents a terminated process returning the value `c`. An automaton `Bad_state`, representing the incorrect states of the system, recognizes all the terms of the form `S(Stop(i), p, full, m)` and `S(p, Stop(i), m, full)` where `i` is any natural, `p` is a process in any configuration (terminated or not), `full` is a non empty message queue and `m` is any message queue (empty or not).

Assume that we naively choose to stop a process as soon as its list is empty. This can be done by adding the following rewrite rule to the previous TRS: `Proc(nil, c) -> Stop(c)`. Since the whole TRS is linear, we can achieve some exact completion steps by normalizing every new transition with new states. After the third completion step, if we compute an intersection with the `Bad_state` automaton, we obtain a non empty intersection meaning that such a bad configuration occurs in the reachable terms. To avoid this bad behavior, it is necessary that each process additionally check that its message queue is empty before terminating. This can be encoded by the following rules, replacing `Proc(nil, c) -> Stop(c)`:

```
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)
```

Similarly we can achieve some exact completion steps and check that the intersection is empty. It is the case for the three first steps but the intersection is no longer empty after the fourth exact completion step. This is due to the fact that a process P_+, for example, may have an empty list, an empty queue and then stops while process P_- has some + in its list, and thus may send later some messages to P_+ that is already stopped. A solution consists in using a synchronizing message `end` sent by P_- to P_+ when P_- has reached the end of its list. Then, P_+ stops only if its list is empty and the head of its queue contains the `end` message sent by P_-. The behavior of P_- is symmetrical. The set of rules controlling the process termination becomes:

```
S(Proc(nil, c), z, m, n) -> S(Proc(nil, c), z, m, add(end,n))   (* P+ ends its list *)
S(x, Proc(nil, c), m, n) -> S(x, Proc(nil, c), add(end,m), n)   (* P- ends its list *)

S(Proc(nil, c), z, cons(end,m), n)  -> S(Stop(c), z, m, n)      (* P+ stops *)
S(x, Proc(nil, c), m, cons(end, n)) -> S(x, Stop(c), m, n)      (* P- stops *)
```

Note that that the two first rewrite rules are non terminating. We also have to modify the tree automaton `Bad_states` such that it recognizes the states where a process is stopped but its message queue contains *at least one* uncounted symbol '+' or '−' and possibly some `end` messages. Then, starting from `A0`, we can perform some exact completion steps and check that the intersection is empty. After the fourth exact completion step, the completed automaton has more than 6000 states and more than 8000 transitions, and intersection with `Bad_state` still results in an empty automaton. Thus, it is worth trying to approximate in order to *prove* that this solution is finally a correct one. Here is a possible approximation to add to the specification:

```
Approximation Procapp
States qlist qnil qsymb qzero qnat qrunproc qemptyproc qterminated
           qend qnil_end qlist_end
Rules [x -> y] -> [
    o -> qzero                           Proc(qlist, z) -> qrunproc
    s(qzero) -> qnat                     Proc(qnil, z) -> qemptyproc
    s(qnat) -> qnat                      Stop(z) -> qterminated
    plus -> qsymb                        cons(qend, qnil) -> qnil_end
    minus -> qsymb                       cons(qend, qnil_end) -> qnil_end
    end -> qend                          add(qend, qnil) -> qnil_end
    nil -> qnil                          add(qend, qnil_end) -> qnil_end
    cons(qsymb, qnil) ->qlist            add(qend, qlist) -> qlist_end
    cons(qsymb, qlist) -> qlist          add(qend, qlist_end) -> qlist_end
                                         add(qsymb, qnil) -> qlist
                                         add(qsymb, qlist) -> qlist ]
```

In this approximation named `Procapp` we define a set of states (new or in common with `A0`) and a set of approximation rules. Note that the left-hand side `[x -> y]` of the rule of `Procapp` matches every new transition to normalize. In the right-hand side of the rule, we simply give all configurations or all configuration patterns that should be normalized into distinct states in order to be able to prove the property. For instance, it is important to distinguish between empty lists (`qnil`), lists with at least a symbol (`qlist`), empty lists with at least an `end` message (`qnil_end`) and lists with at least a symbol and at least an `end` message (`qlist_end`). Similarly, it is important to distinguish between running processes (`qrunproc`), processes with an empty list (`qemptyproc`) and terminated processes (`qterminated`). Thanks to this approximation, the completion process does not require any other normalization from the user and terminate on a tree automaton with 13 states and 95 transitions. The intersection between this automaton and the automaton `Bad_state` is empty and thus we have proved that all bad states are unreachable from the initial configurations.

3.3 More expressiveness and more detailed approximations

In this section, we show how to design more precise approximations to verify more complex rewrite specifications with Associative and Commutative (AC) symbols and non left-linear TRSs. Such an extended specification formalism is useful, for instance, to model and verify cryptographic protocols [8]. Those protocols are supposed to be secure in an hostile environment where an intruder stores every message and every key he sees, decrypt some parts, forges new messages with the parts he has and send every possible message in its store in order to attack some agents. We can model the intruder store using a term built with an Associative Commutative (AC) symbol `store`, where for example the term `store(a, store(store(b, a), c))` represents the multiset $\{\!\{a, a, b, c\}\!\}$. The terms `pubkey(x)`, `privkey(x)`, `encr(k, c)`, and `cons(x, y)` represent respectively the public and private key of an agent `x`, the encryption of `c` using the key `k` and a message composed of two parts `x` and `y`. We can model some of the message constructions that an intruder can do on its store, like it is done for example in [16]:

```
(* The intruder can encrypt any stored component with any stored key *)
store(z, pubkey(x)) -> store(encr(pubkey(x), z), store(z, pubkey(x)))
store(z, privkey(x)) -> store(encr(privkey(x), z), store(z, privkey(x)))

(* The intruder can decompose or compose any component he has *)
store(cons(x,y), m) -> store(store(cons(x,y), m),store(x, y))
store(x, y) -> store(cons(x, y), store(x,y))
```

The rules encoding the AC behavior of the `store` symbol are also necessary:

```
store(x, y) -> store(y, x)
store(store(x, y), z) -> store(x, store(y, z))
store(x, store(y, z)) -> store(store(x, y), z)
```

Note that those rules are highly non terminating, we thus have to define strong approximation rules. When using AC symbols are simply used for representing sets of objects, a quite natural approximation rule for the `store` symbol is the following:

```
[store(x, y) -> z] -> [x -> z      y -> z]
```

This rule normalizes every new configuration of the form `store(s, t) -> q` (where `s` and `t` are not states) into configurations `s -> q`, `t -> q` and `store(q, q) -> q`. The intuition behind this rule is that every 'subset' `x` and `y` of the store `store(x, y)` should be recognized by the same state as `store(x, y)`.

We have represented some of the intruder manipulations but our specification still lacks decryption of encrypted components for which the intruder has the decryption key. To describe this behavior, we need non left-linear rules in order to check the correspondence between stored keys and the key that was used to encrypt the message:

```
store(encr(privkey(x),z), pubkey(x)) -> store(encr(privkey(x),z), store(pubkey(x),z))
store(encr(pubkey(x),z), privkey(x)) -> store(encr(pubkey(x),z), store(privkey(x),z))
```

This encodes the fact that to decrypt a message encoded with the *private* key of an agent x, the intruder should have in its store the *public* key of x, and vice versa. In [8], we have defined a sufficient constraint on the automaton such that completion with non left-linear rules still gives an over-approximation of reachable terms. Roughly, the constraint on the automaton $\mathcal{A}_i$ is the following: for every rule $l \to r$ whose left-hand side has a non linear variable x, for every substitution σ and every state q such that $l\sigma \to_{\mathcal{A}}^* q$ there exists a unique q' such that $x\sigma \to_{\mathcal{A}}^* q'$. The state q' is called a *deterministic state*. Going back to the two rewrite rules of our example, this means that we have to ensure that terms matched by the variable `x` are recognized by deterministic states. Since terms matched by `x` should be agents, all we have to ensure is that during completion all terms representing agents are recognized by deterministic states: this can be ensured by the choice of appropriate approximation rules or priority transitions. For example assume, that we represent agents by terms of the form `agent(i)` where `i` is a natural. Then the following approximation rules should make the state `qagt` deterministic[4]:

```
[x -> y] -> [o -> qnat      s(qnat) -> qnat      agent(qnat) -> qagt]
```

The state `qagt` is deterministic but it contains all agents. However, using the same technique it is possible to have any variety of regular categorization of agents, for example: two distinct agents A and B, a server S, an unbounded number of other honest agents and an unbounded number of dishonest agents:

```
[x -> y] -> [o -> zero               s(zero) -> one               s(one) -> two
             s(two) -> qodd          s(qodd) -> qeven             s(qeven) -> qodd
             agent(zero) -> qA       agent(one)  -> qB            agent(two)  -> qS
             agent(qodd) -> qHonnest   agent(qeven) -> qDishonnest ]
```

In the context of cryptographic protocols, deterministic states are also useful for a precise normalization of all the new transitions generated by the protocol specification. For instance, assume that the intruder obtains a new piece of information: let `encr(pubkey(qA), cons(m1, cons(m2, m3))) -> qstore` be the new transition to be added, where `qA` is the deterministic state recognizing only the agent A and `qstore` the state recognizing all the store of the intruder. If we use a too drastic normalization rule of the form:

```
[x -> y] -> [z -> qstore]
```

then messages components `m1`, `m2`, `m3` are likely to be normalized into state `qstore` and thus available to the intruder in spite of their encryption by `pubkey(qA)`. A normalization rule of the form:

```
[encr(x, y) -> z] -> [y -> qprotected]
```

[4]For safety, this can also be checked during completion by computing intersections between the states matched by non linear variables. This is another action proposed by the user menu.

avoids this problem but normalizes the content of every encrypted component by the same state `qprotected`, and thus makes no difference between a secret known by A, B or any other agent. A correct solution is for example:

```
[encr(pubkey(qA), y) -> z] -> [y -> qAsecret]
```

which produces the normalized transitions `encr(pubkey(qA), qAsecret) -> qstore` and `cons(m1, cons(m2, m3)) -> qAsecret`. If order and number of messages `m1`, `m2` and `m3` are not important for the verification of the property, it is either possible to collapse the structure of the message by adding the following approximation rule:

```
[x -> qAsecret] -> [y -> qAsecret]
```

which normalizes every subterm of the configuration matched by `x` with the unique state `qAsecret`, for every new transition matching `[x -> qAsecret]`. Thus, every message component remains secret but the message structure has been lost. More details about proving cryptographic protocols properties with our approximation technique are given in [8].

4 Conclusion

In this article we presented the Timbuk tool and its completion algorithm to construct approximations of the sets of descendants. Whereas many tools prove some properties on TRS under strong restrictions on the TRSs, approximations permit to prove some unreachability properties on non linear TRSs that are non terminating and may contain some AC symbols. Properties that can be proved are only 'regular' properties but are of practical interest for instance in the case of cryptographic protocols. Furthermore, we have presented here some new tools – priority transitions and approximation rules – in order to make approximation construction more intuitive and more automatic.

Although we focus on approximation, for some classes of TRSs and regular languages E, $\mathcal{R}^*(E)$ is regular and can be exactly recognized [10, 11]. But those classes are very restrictive if we intend to use TRS for specification in general. In spite of this, D. Monniaux [15] has shown that even some simple decidable classes can be used to model some intruder knowledge for cryptographic protocol verification. A more recent regular class was found by P. Réty [17] where restrictions are weaker on the TRS and stronger on the regular language E which is restricted to data terms. Nevertheless, the restriction on the TRS is still strong w.r.t. a specification language since it forbids, in particular, nested function symbols. As a result, all the examples of this paper, are still out of the scope of an exact computation of $\mathcal{R}^*(E)$. However, we think that it is worth integrating results of [17] in Timbuk for proving reachability in addition to unreachability.

In [12, 5], some approximations based on tree languages are proposed: for higher order functional programs transformed into term rewriting systems in [12] and for imperative and functional first order programs in [5]. In both, the aim is to achieve static analysis and thus the priority is given to automation: for ensuring termination, the approximation methodology is fixed and there is no user control over the approximation rules. Moreover, the approximation of [12] and the widening of [5, section 6.1] could not be used in our context since they loose relational information. In particular, if many new transitions with left-hand side of the form `encr(pubkey(A), m1)`, `encr(pubkey(B), m2)`, ... are produced by the same rewrite rule, they are all normalized using the same states. Then `m1` will share a common state with `m2` and thus `m1` will be no longer secret for `B` and vice versa.

As far as we know, two other distributed tools also implement tree automata: Mona & Fido [13] and RX [18]. In Mona & Fido, tree automata are essentially an internal data structure (deterministic binary tree automata optimized with BDDs) used to decide WS1S and WS2S logics. In RX[5] like in Timbuk, the user can describe regular languages in the usual way using deterministic or non-deterministic grammars, and then compute some intersections, unions, differences between those languages. However, like Mona & Fido, RX was designed for a purpose very different from Timbuk and thus does not implement approximations.

References

[1] F. Baader and T. Nipkow. *Term Rewriting and All That.* Cambridge University Press, 1998.

[2] P. Borovanský, C. Kirchner, H. Kirchner, P.-E. Moreau, and M. Vittek. ELAN: A logical framework based on computational systems. In *Proc. 1st WRLA*, volume 4 of *ENTCS*, Asilomar (California), 1996.

[3] M. Clavel, F. Durán, S. Eker, P. Lincoln, and J. Meseguer. An Introduction to Maude (Beta Version). Technical report, SRI, March 1998.

[4] 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/`, 1997.

[5] P. Cousot and R. Cousot. Formal Language, Grammar and Set-Constraint-Based Program Analysis by Abstract Interpretation. In *Conference Record of FPCA'95 SIGPLAN/SIGARCH/WG2.8*, pages 170–181. ACM Press, 1995.

[6] 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.

[7] 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.

[5] which was designed to prove some termination theorems in combinatory logic.

[8] 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.

[9] T. Genet and V. Viet Triem Tong. Timbuk Documentation. IRISA / Université de Rennes 1, 2001. `http://www.irisa.fr/lande/genet/timbuk/`.

[10] R. Gilleron and S. Tison. Regular tree languages and rewrite systems. *Fundamenta Informaticae*, 24:157–175, 1995.

[11] F. Jacquemard. Decidable approximations of term rewriting systems. In H. Ganzinger, editor, *Proc. 7th RTA Conf., New Brunswick (New Jersey, USA)*, pages 362–376. Springer-Verlag, 1996.

[12] N.D. Jones. Flow analysis of lazy higher-order functional programs. In S. Abramsky and C. Hankin, editors, *Abstract Interpretation of Declarative Languages*, pages 103–122. Ellis Horwood, Chichester, England, 1987.

[13] N. Klarlund and A. Møller. MONA Version 1.4 User Manual, January 2001.

[14] X. Leroy, D. Doligez, J. Garrigue, D. Rémy, and J. Vouillon. The Objective Caml system release 3.00 – Documentation and user's manual. INRIA, 2000. `http://caml.inria.fr/ocaml/htmlman/`.

[15] D. Monniaux. Abstracting Cryptographic Protocols with Tree Automata. In *Proc. 6th SAS, Venezia (Italy)*, 1999.

[16] L. Paulson. Proving Properties of Security Protocols by Induction. In *10th Computer Security Foundations Workshop*. IEEE Computer Society Press, 1997.

[17] P. Réty. Regular Sets of Descendants for Constructor-based Rewrite Systems. In *Proc. 6th LPAR Conf., Tbilisi (Georgia)*, volume 1705 of *LNAI*. Springer-Verlag, 1999.

[18] J. Waldmann. RX: an interpreter for Rational Tree Languages, 1998. `http://www.informatik.uni-leipzig.de/~joe/rx/`.

Unité de recherche INRIA Lorraine, Technopôle de Nancy-Brabois, Campus scientifique,
615 rue du Jardin Botanique, BP 101, 54600 VILLERS LÈS NANCY
Unité de recherche INRIA Rennes, Irisa, Campus universitaire de Beaulieu, 35042 RENNES Cedex
Unité de recherche INRIA Rhône-Alpes, 655, avenue de l'Europe, 38330 MONTBONNOT ST MARTIN
Unité de recherche INRIA Rocquencourt, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex
Unité de recherche INRIA Sophia-Antipolis, 2004 route des Lucioles, BP 93, 06902 SOPHIA-ANTIPOLIS Cedex

Éditeur
INRIA, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex (France)
http://www.inria.fr
ISSN 0249-6399