Certifying a Tree Automata Completion Checker

Benoît Boyer, Thomas Genet, and Thomas Jensen

IRISA / Université de Rennes 1 / CNRS
Campus de Beaulieu
F-35042 Rennes Cedex
bboyer,genet,jensen@irisa.fr

Abstract. Tree automata completion is a technique for the verification of infinite state systems. It has already been used for the verification of cryptographic protocols and the prototyping of Java static analyzers. However, as for many other verification techniques, the correctness of the associated tool becomes more and more difficult to guarantee. It is due to the size of the implementation that constantly grows and due to optimizations which are necessary to scale up the efficiency of the tool to verify real-size systems. In this paper, we define and develop a checker for tree automata produced by completion. The checker is defined using Coq and its implementation is automatically extracted from its formal specification. Using extraction gives a checker that can be run independently of the Coq environment. A specific algorithm for tree automata inclusion checking has been defined so as to avoid the exponential blow up. The obtained checker is certified in Coq, independent of the implementation of completion, usable with any approximation performed during completion, small and fast. Some benchmarks are given to show how efficient the tool is.

1 Introduction

Static program analysis is one of the cornerstones of software verification and is increasingly used to protect computing devices from malicious or mal-functioning code. However, program verifiers are themselves complex programs and a single error may jeopardize the entire trust chain of which they form part. Efforts have been made to certify static analyzers [KN03,BD04,CJPR05] or to certify the results obtained by static analyzers [LT00,BJP06] in Coq in order to increase confidence in the analyzers. In this paper, we instantiate the general framework used in [BJP06] to the particular case of analyzing term rewriting systems by tree automata completion [Gen98,FGVTT04]. Given a term rewriting system, the tree automata completion is a technique for over-approximating the set of terms reachable by rewriting in order to prove the unreachability of certain "bad" states that violate a given security property. This technique has already been used to prove security properties on cryptographic protocols [GK00], [GTTVTT03,BHK04,ABB+05,ZD06] and, more recently, to prototype static analyzers on Java byte code [BGJL07].

In this paper, we show how to mechanize the proof, within the Coq proof assistant, that the tree automaton produced by completion recognizes an over-approximation of all reachable terms. Coq is based on constructive logic (Calculus of Inductive Constructions) and it is possible to extract an Ocaml or Haskell function implementing exactly the algorithm whose specification has been expressed in Coq. The extracted code is thus a *certified* implementation of the specification given in the Coq formalism. Extracted programs are standalone and do not require the Coq environment to be executed. For details about the extraction mechanisms, readers can refer to [BC04].

A specific challenge in the work reported here has been how to marry constructive logic and efficiency. Previous case studies with tree automata completion, on cryptographic protocols [GTTVTT03] and on Java bytecode [BGJL07] show that we need an efficient completion algorithm to verify properties of real models. For instance, the current implementation of completion (called Timbuk [GVTT00]) is based on imperative data structures like hash tables whereas Coq allows only pure functional structures. A second problem is the termination of completion. Since Coq can only deal with total functions, functions must be proved terminating for any computation. In general, such a property cannot be guaranteed on completion because it mainly depends on term rewriting system and approximation equations given initially.

For these two reasons, there is little hope to specify and certify an efficient and purely functional version of the completion algorithm. Instead, we have adopted a solution based on a result-checking approach. It consists of building a smaller program (called the *checker*) - certified in Coq - that checks if the tree automaton computed by Timbuk is sound. In this paper, we restrict to the case of left-linear term rewriting systems which revealed to be sufficient for verifying Java programs [BGJL07]. However, a checker dealing with general term rewriting systems like completion does in [FGVTT04] is under development.

The closest work to ours is the one done by X. Rival and J. Goubault-Larrecq [RGL01]. They have designed a library to manipulate tree automata in Coq and proposed some optimized formal data structures that we reuse. However, we aim at dealing with larger tree automata than those used in their benchmarks. Moreover, we need some other tools which are not provided by the library as for example a specific algorithm to check inclusion.

This paper is organized as follows. Rewriting and tree automata are reviewed in Section 2 and tree automata completion in Section 3. Section 4 states the main functions to define, inclusion and closure test, and the corresponding theorems to prove. Section 5 and Section 6 give the Coq formalization of rewriting and of tree automata, respectively. The core of the checker consists of two algorithms: an optimized automata inclusion test, defined in Section 7, and a procedure for checking that an automaton is *closed* under rewriting w.r.t. a given term rewriting system, defined in Section 8. Section 9 gives some details about the performances of the checker in practice. Finally, we conclude and list some ongoing research on this subject.

2 Preliminaries

Comprehensive surveys can be found in [BN98] for rewriting, and in [CDG+02,GT95] 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 function σ from $\mathcal{X}$ into $\mathcal{T}(\mathcal{F},\mathcal{X})$, which can be extended uniquely to an endomorphism of $\mathcal{T}(\mathcal{F},\mathcal{X})$. A position p for a term t is a word over $\mathbb{N}$. The empty sequence ϵ denotes the top-most position. The set $\mathcal{P}os(t)$ of positions of a term t is inductively defined by:

- $\mathcal{P}os(t) = \{\epsilon\}$ if $t \in \mathcal{X}$
- $\mathcal{P}os(f(t_1,\ldots,t_n)) = \{\epsilon\} \cup \{i.p \mid 1 \leq i \leq n \text{ and } p \in \mathcal{P}os(t_i)\}$

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. A term rewriting system (TRS) $\mathcal{R}$ 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* if each variable of l (resp. r) occurs only once in l. A TRS $\mathcal{R}$ is left-linear if every rewrite rule $l \to r$ of $\mathcal{R}$ is left-linear). The TRS $\mathcal{R}$ induces a rewriting relation $\to_{\mathcal{R}}$ on terms whose reflexive transitive closure is denoted by $\to^\star_{\mathcal{R}}$. The set of $\mathcal{R}$-descendants of a set of ground terms E is $\mathcal{R}^*(E) = \{t \in \mathcal{T}(\mathcal{F}) \mid \exists s \in E \text{ s.t. } s \to^\star_{\mathcal{R}} t\}$.

The *verification technique* defined in [Gen98,FGVTT04] is based on $\mathcal{R}^*(E)$. 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 [GT95]. However, it is possible to over-approximate it [Gen98,FGVTT04,Tak04] using tree automata, i.e. a finite representation of infinite (regular) sets of terms. In this verification setting, the TRS $\mathcal{R}$ represents the system to verify, sets of terms E and Bad represent respectively the set of initial configurations and the set of "bad" configurations that should not be reached. Then, using tree automata completion, we construct a tree automaton $\mathcal{B}$ whose language $\mathcal{L}(\mathcal{B})$ is such that $\mathcal{L}(\mathcal{B}) \supseteq \mathcal{R}^*(E)$. Then if $\mathcal{L}(\mathcal{B}) \cap Bad = \emptyset$ then this proves that $\mathcal{R}^*(E) \cap Bad = \emptyset$, and thus that none of the "bad" configurations is reachable. We now define tree automata.

Let $\mathcal{Q}$ be a finite 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 $c = f(q_1,\ldots,q_n)$, $f \in \mathcal{F}$ whose arity is n, and $q_1,\ldots,q_n \in \mathcal{Q}$.*

Definition 2 (Bottom-up nondeterministic finite tree automaton). *A bottom-up nondeterministic 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.*

The *rewriting relation* on $\mathcal{T}(\mathcal{F} \cup \mathcal{Q})$ induced by the transitions of $\mathcal{A}$ (the set Δ) is denoted by $\rightarrow_\Delta$. When Δ is clear from the context, $\rightarrow_\Delta$ will also be denoted by $\rightarrow_\mathcal{A}$. Here is the definition of the recognized language, see [BGJ08] for examples.

Definition 3 (Recognized language). *The tree language recognized by $\mathcal{A}$ in a state q is $\mathcal{L}(\mathcal{A}, q) = \{t \in \mathcal{T}(\mathcal{F}) \mid t \rightarrow^\star_\mathcal{A} q\}$. 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.*

3 Tree Automata Completion

Given a tree automaton $\mathcal{A}$ and a TRS $\mathcal{R}$, the tree automata completion algorithm, proposed in [Gen98,FGVTT04], computes a tree automaton $\mathcal{A}^*_\mathcal{R}$ such that $\mathcal{L}(\mathcal{A}^*_\mathcal{R}) = \mathcal{R}^*(\mathcal{L}(\mathcal{A}))$ when it is possible (for some of the classes of TRSs where an exact computation is possible, see [FGVTT04]) and such that $\mathcal{L}(\mathcal{A}^*_\mathcal{R}) \supseteq \mathcal{R}^*(\mathcal{L}(\mathcal{A}))$ otherwise.

The tree automata completion works as follows. From $\mathcal{A} = \mathcal{A}^0_\mathcal{R}$ completion builds a sequence $\mathcal{A}^0_\mathcal{R}.\mathcal{A}^1_\mathcal{R} \dots \mathcal{A}^k_\mathcal{R}$ 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 fixpoint 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 note $\mathcal{A}^*_\mathcal{R} = \mathcal{A}^k_\mathcal{R}$ and we have $\mathcal{L}(\mathcal{A}^*_\mathcal{R}) = \mathcal{R}^*(\mathcal{L}(\mathcal{A}^0_\mathcal{R}))$, or $\mathcal{L}(\mathcal{A}^*_\mathcal{R}) \supseteq \mathcal{R}^*(\mathcal{L}(\mathcal{A}))$ if $\mathcal{R}$ is not in one class of [FGVTT04]. 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 $\mathcal{T}(\mathcal{F} \cup \mathcal{Q})$. For a substitution $\sigma : \mathcal{X} \mapsto \mathcal{Q}$ and a rule $l \rightarrow r \in \mathcal{R}$, a critical pair is an instance $l\sigma$ of l such that there exists $q \in \mathcal{Q}$ satisfying $l\sigma \rightarrow^*_{\mathcal{A}^i_\mathcal{R}} q$ and $l\sigma \rightarrow_\mathcal{R} r\sigma$. Note that since $\mathcal{R}$, $\mathcal{A}^i_\mathcal{R}$ and the set $\mathcal{Q}$ of states of $\mathcal{A}^i_\mathcal{R}$ are finite, there is only a finite number of critical pairs. 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. $r\sigma \rightarrow^*_{\mathcal{A}^{i+1}_\mathcal{R}} q$, see Figure 1.

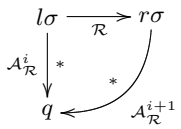

Fig. 1. Critical pair

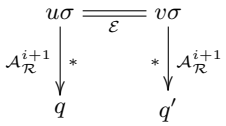

Fig. 2. Detection of merging

However, the transition $r\sigma \rightarrow q$ is not necessarily a normalized transition of the form $f(q_1, \dots, q_n) \rightarrow q$ and so it has to be normalized first. Thus, instead of adding $r\sigma \rightarrow q$ we add $Norm(r\sigma \rightarrow q)$ to transitions of $\mathcal{A}^i_\mathcal{R}$. Here is the $Norm$ function used to normalize transitions. Note that, in this function, transitions are normalized using either new states of $\mathcal{Q}_{new}$ or states of $\mathcal{Q}$, states of the automaton being completed. As we will see in Lemma 1, this has no effect on the safety of the normalization but only on its precision.

Definition 4 ($Norm$). *Let $\mathcal{A} = \langle \mathcal{F}, \mathcal{Q}, \mathcal{Q}_f, \Delta \rangle$ be a tree automaton, $\mathcal{Q}_{new}$ a set of* new *states such that $\mathcal{Q} \cap \mathcal{Q}_{new} = \emptyset$, $t \in \mathcal{T}(\mathcal{F} \cup \mathcal{Q})$ and $q \in \mathcal{Q}$. The function $Norm$ is inductively defined by:*

- $Norm(t \to q) = \emptyset$ *if* $t = q$,
- $Norm(t \to q) = \{c \to q \mid c \to t \in \Delta\}$ *if* $t \in \mathcal{Q}$,
- $Norm(f(t_1, \ldots, t_n) \to q) = \bigcup_{i=1\ldots n} Norm(t_i \to q_i) \cup \{f(q_1, \ldots, q_n) \to q\}$ *where* $\forall i = 1 \ldots n : (t_i \in \mathcal{Q} \Rightarrow q_i = t_i) \wedge (t_i \in \mathcal{T}(\mathcal{F} \cup \mathcal{Q}) \setminus \mathcal{Q} \Rightarrow q_i \in \mathcal{Q} \cup \mathcal{Q}_{new})$.

When using only new states to normalize all the new transitions occurring in all the completion steps, completion is as precise as possible. However, doing so, completion is likely not to terminate (because of general undecidability results [GT95]). Enforcing termination of completion can be easily done by bounding the set of new states to be used with $Norm$ during the whole completion. We then obtain a finite tree automaton over-approximating the set of reachable states. The fact that normalizing with any set of states (new or not) is *safe* is guaranteed by the following simple lemma. For the general safety theorem of completion see [FGVTT04].

Lemma 1. *For all tree automaton $\mathcal{A} = \langle \mathcal{F}, \mathcal{Q}, \mathcal{Q}_f, \Delta \rangle$, $t \in \mathcal{T}(\mathcal{F} \cup \mathcal{Q}) \setminus \mathcal{Q}$ and $q \in \mathcal{Q}$, if $\Pi = Norm(t \to q)$ whatever the states chosen in $Norm(t \to q)$ we have $t \to_\Pi^* q$.*

Proof. This can be done by a simple induction on transitions [FGVTT04].

To let the user of completion guide the approximation, we use two different tools: a set N of *normalization rules* (see [FGVTT04]) and a set $\mathcal{E}$ of *approximation equations*. Rules and equations can be either defined by hand so as to prove a complex property [GTTVTT03], or generated automatically when the property is more standard [BHK04]. Normalization rules can be seen as a specific strategy for normalizing new transitions using the $Norm$ function. We have seen that Lemma 1 is enough to guarantee that the chosen normalization strategy has no impact on the safety of completion. Similarly, for our checker, we will see in Section 8 that the related Coq safety proof can be carried out independently of the normalization strategy (i.e. set N of normalization rules). On the opposite, the effect of approximation equations is more complex and has to be studied more carefully. An approximation equation is of the form $u = v$ where $u, v \in \mathcal{T}(\mathcal{F}, \mathcal{X})$. Let $\sigma : \mathcal{X} \mapsto \mathcal{Q}$ be a substitution such that $u\sigma \to_{\mathcal{A}_\mathcal{R}^{i+1}} q$, $v\sigma \to_{\mathcal{A}_\mathcal{R}^{i+1}} q'$ and $q \neq q'$, see Figure 2. Then, we know that there exists some terms recognized by q and some recognized by q' which are equivalent modulo $\mathcal{E}$. A correct over-approximation of $\mathcal{A}_\mathcal{R}^{i+1}$ consists in applying the $Merge$ function to it, i.e. replace $\mathcal{A}_\mathcal{R}^{i+1}$ by $Merge(\mathcal{A}_\mathcal{R}^{i+1}, q, q')$, as long as an approximation equation of $\mathcal{E}$ applies. The $Merge$ function, defined below, merges states in a tree automaton. See [BGJ08] for examples of completion and approximation.

Definition 5 ($Merge$). *Let $\mathcal{A} = \langle \mathcal{F}, \mathcal{Q}, \mathcal{Q}_F, \Delta \rangle$ be a tree automaton and q_1, q_2 be two states of $\mathcal{A}$. We denote by $Merge(\mathcal{A}, q_1, q_2)$ the tree automaton where every occurrence of q_2 is replaced by q_1 in $\mathcal{Q}$, $\mathcal{Q}_F$ and in every left-hand side and right-hand side of every transition of Δ.*

4 A result checker for tree automata completion

By moving the certification problem from the completion algorithm to the checker, the certification problem consists in proving the following Coq theorem. In Coq specifications, recall that '→' is used to denote both the logical implication and functional types. Similarly, ':' is used to give the type of a function, the type of a data or the statement for a theorem.

```
Theorem sound_checker :
      ∀ A A' R, checker A R A' = true → ApproxReachable A R A'.
```

where `ApproxReachable` is a Coq predicate that describes the Soundness Property: $\mathcal{L}(A')$ *contains all terms reachable by rewriting terms of* $\mathcal{L}(A)$ *with* $\mathcal{R}$, i.e. $\mathcal{L}(\mathcal{A}') \supseteq \mathcal{R}^*(\mathcal{L}(\mathcal{A}))$. To state formally this predicate in Coq, we need to give a Coq axiomatization of Term Rewriting Systems and of Tree Automata. It is given in Section 5. Given two automata $\mathcal{A}$, $\mathcal{A}'$ and a TRS $\mathcal{R}$ the checker verifies that $\mathcal{L}(\mathcal{A}') \supseteq \mathcal{R}^*(\mathcal{L}(\mathcal{A}))$ or (`ApproxReachable A R A'`) in Coq. To perform this, we need to check the two following properties:

- `Included`: inclusion of initial set in the fixpoint: $\mathcal{L}(\mathcal{A}) \subseteq \mathcal{L}(\mathcal{A}')$.
- `IsClosed`: $\mathcal{A}'$ is closed by rewriting with $\mathcal{R}$: For all $l \to r \in \mathcal{R}$ and all $t \in \mathcal{L}(\mathcal{A}')$, if t is rewritten in t' by the rule $l \to r$ then $t' \in \mathcal{L}(\mathcal{A}')$.

For each item, we provide a Coq function and its correctness theorem: function `inclusion` is dedicated to inclusion checking and function `closure` checks if a tree automaton is closed by rewriting. We also give the theorem used to deduce `ApproxReachable A R A'` from `Included A A'` and `IsClosed R A'`:

```
Theorem inclusion_sound:
      ∀ A A', inclusion A A' = true → Included A A'.

Theorem closure_sound:
      ∀ R A', closure R A' = true → IsClosed R A'.

Theorem Included_IsClosed_ApproxReachable:
      ∀ A A' R, Included A A' → IsClosed R A' → ApproxReachable A R A'.
```

Note that, in this paper we focus on the proof of $\mathcal{L}(\mathcal{A}') \supseteq \mathcal{R}^*(\mathcal{L}(\mathcal{A}))$. However, to prove the unreachability property, the emptiness of the intersection between $\mathcal{L}(\mathcal{A}')$ and the bad term set has also to be verified. Since the formalization in Coq of the intersection and emptiness decision are close to their standard definition [CDG+02], and since they have already been covered by [RGL01], they are not be detailed in this paper.

5 Formalization of Term Rewriting Systems

The aim of this part is to formalize in Coq: terms, term rewriting systems, reachable terms and the reachability problem itself. First, we use the positive integers

provided by the Coq's standard library to define symbol sets like variables ($\mathcal{X}$) or function symbols ($\mathcal{F}$). We rename `positive` into `ident` to be more explicit. Then, we define term set $\mathcal{T}(\mathcal{F}, \mathcal{X})$ using inductive types:

```
Inductive term : Set :=
| Fun : ident → list term → term
| Var : ident → term.
```

A rewrite rule $l \to r$ is represented by a pair of terms with a well-definition proof, i.e. a Coq proof that the set of variables of r is a subset of the set of variables of l. The function `Fv : term →list ident` builds the set of variables for a term.

```
Inductive rule : Set :=
| Rule (l r : term)(H : subseteq (Fv r) (Fv l)) : rule.
```

In the following, `list rule` type represents a TRS. In Coq we use `(t @ sigma)` to denote the term resulting of the application of a substitution `sigma` to each variable that occurs in a term `t`. In Coq, the rewriting relation *"u is rewritten in v by $l \to r$"*, commonly defined by $\exists \sigma \; s.t. \; u|_p = l\sigma \; \wedge \; v = u[r\sigma]_p$, is split into two predicates:

- The first one defines the rewriting of a term at the topmost position. In fact, the set of term pairs (t, t') which are rewritten at the top most by the rule can be seen as the set of term pairs $(l\sigma, r\sigma)$ for any substitution σ.
- The second one just defines inductively the rewriting relation at any position of a term t by a rule $l \to r$, by the topmost rewriting of any subterm of t by $l \to r$.

```
(* Topmost rewriting : *)
Inductive TRew (x : rule) : term → term → Prop :=
| R_Rew : ∀ s l r (H : subseteq (Fv r) (Fv l)),
             x = Rule l r H → TRew x (l @ s) (r @ s).
```

Similarly, using an inductive definition it is possible to define the `Rew` predicate for rewriting at any position. Then we have to define $\to^*_{\mathcal{R}}$. In Coq, we prefer to see it as the predicate `Reachable R u` that characterizes the set of reachable terms from u by $\to^*_{\mathcal{R}}$.

```
Inductive Reachable(R : list rule)(t : term) : term → Prop:=
| R_refl : Reachable R t t
| R_trans : ∀ u v r, Reachable R t u → In r R → Rew r u v →
    Reachable R t v.
```

6 Formalization of Tree Automata

The fact that the checker, to be executed, is directly extracted from the Coq formalization has an important consequence on the tree automata formalization. Since the data structures used in the formalization are those that are really used for the execution, they need to be formal *and* efficient. For tree automata,

instead of a naive representation, it is thus necessary to use optimized formal data structures borrowed from [RGL01]. In Section 5, we have represented variables $\mathcal{X}$ and function symbols $\mathcal{F}$ by the type `ident`. We do the same for $\mathcal{Q}$. We define a tree automaton as a pair $(\mathcal{Q}_F, \Delta)$, where $\mathcal{Q}_F$ is the finite set of final states, and Δ the finite set of normalized transitions like $f(q_1, \ldots, q_n) \to q$. In Coq, the `t_automaton` record stands for this pair where the final state field `qf` is a simple `list ident` and the transition set field `delta` has `Delta.t` type. The `Delta` module contains the implementation, the theorems and proofs for normalized transition sets. This representation is based on the `FMapPositive` Coq library of functional mappings, where data are indexed by `positive` numbers. A set of transitions of type `Delta.t` is a map where each state q indexes a set of configuration like $f(q_1, \ldots, q_n)$. In the same way, each set of configuration is a map where function symbols are used to index a stack of state lists. For all transitions $f(q_1, \ldots, q_n) \to q$, the state list $(q_1, \ldots, q_n)$ is stored in the stack indexed by symbol f and state q. Now we can define a predicate to characterize the recognized language of a tree automaton. In fact, we are defining the set of ground terms that are reduced to a state q by a transition set Δ. This set, which corresponds to $\mathcal{L}(\mathcal{A}, q)$ if Δ is the set of transitions of $\mathcal{A}$, can be constructed inductively in Coq using the single deduction rule:

$$\frac{t_1 \in \mathcal{L}(\Delta, q_1) \qquad \ldots\ldots \qquad t_n \in \mathcal{L}(\Delta, q_n)}{f(t_1, \ldots, t_n) \in \mathcal{L}(\Delta, q)} \text{ If } f(q_1, \ldots, q_n) \to q \in \Delta$$

In Coq, we express this statement using a mutually inductive predicate `IsRec`. A term t is recognized by a tree automaton $(\mathcal{Q}_F, \Delta)$, if the predicate `IsRec` $\Delta\ q\ t$ is valid for $q \in \mathcal{Q}_F$.

7 An optimized inclusion checker

In this part, we give the formal definition of the `Included` property and of the `inclusion` Coq function used to effectively check the tree automata inclusion. From the previous formal definitions on tree automata, we can state the `Included` predicate in the following way:

```
Definition Included (a b : t_automaton) : Prop :=
  ∀ t q, In q a.(qf) → IsRec a.(delta) q t →
    ∃ q', In q' b.(qf) ∧ IsRec b.(delta) q' t.
```

Now let us focus on the function `inclusion` itself. The usual algorithm for proving inclusion of regular languages recognized by nondeterministic bottom-up tree automata, for instance for proving $\mathcal{L}(\mathcal{A}) \subseteq \mathcal{L}(\mathcal{B})$, consists in proving that $\mathcal{L}(\mathcal{A}) \cap \mathcal{L}(\overline{\mathcal{B}}) = \emptyset$, where $\overline{\mathcal{B}}$ is the complement automaton for $\mathcal{B}$. However, the algorithm for building $\overline{\mathcal{B}}$ from $\mathcal{B}$ is EXPTIME-complete [CDG+02]. This is the reason why we here define a criterion with a better practical complexity. It is is based on a simple syntactic comparison of transition sets, i.e. we check the inclusion of transition sets modulo the renamings performed by the *Merge* function. This increases a lot the efficiency of our checker, especially by saving

memory. This is crucial to check inclusion of big tree automata (see Section 9). This algorithm is correct but, of course, it is not complete in general, i.e. not always able to prove that $\mathcal{L}(\mathcal{A}) \not\subseteq \mathcal{L}(\mathcal{B})$. However, we show in the following that, under certain conditions on $\mathcal{A}$ and $\mathcal{B}$ which are satisfied if $\mathcal{B}$ is obtained by completion of $\mathcal{A}$, this algorithm is also complete and thus becomes a decision procedure. First, we introduce the following notation:

Γ	: induction hypothesis set
Δ_i	: transition set of the tree automaton $\mathcal{A}_i$
$\{c \mid c \to q \in \Delta\}$	: configurations of Δ that are rewritten in q
$\{c_i\}_n^m$	: configuration set from c_n to c_m

We formulate our inclusion problem by formulas of the form: $\Gamma \vdash_{A,B} q \Subset q'$. Such a statement stands for: under the assumption Γ, it is possible to prove that $\mathcal{L}(A, q) \subseteq \mathcal{L}(B, q')$. The algorithm consists in building proof trees for those statements using the following set of deduction rules.

$$\text{(Induction)}\ \frac{\Gamma \cup \{q \Subset q'\} \vdash_{A,B} \{c \mid c \to_{\Delta_A} q\} \Subset \{c \mid c \to_{\Delta_B} q'\}}{\Gamma \vdash_{A,B} q \Subset q'} \text{ if } (q \Subset q') \notin \Gamma$$

$$\text{(Axiom)}\ \frac{}{\Gamma \cup \{q \Subset q'\} \vdash_{A,B} q \Subset q'} \qquad \text{(Empty)}\ \frac{}{\Gamma \vdash_{A,B} \emptyset \Subset \{c'_j\}_1^m}$$

$$\text{(Split-l)}\ \frac{\Gamma \vdash_{A,B} c_1 \Subset \{c'_j\}_1^m \quad \ldots\ldots \quad \Gamma \vdash_{A,B} c_n \Subset \{c'_j\}_1^m}{\Gamma \vdash_{A,B} \{c_i\}_1^n \Subset \{c'_j\}_1^m}$$

$$\text{(Weak-r)}\ \frac{\Gamma \vdash_{A,B} c \Subset c'_k}{\Gamma \vdash_{A,B} c \Subset \{c'_i\}_1^n} \text{ if } (1 \leq k \leq n) \qquad \text{(Const.)}\ \frac{}{\Gamma \vdash_{A,B} a() \Subset a()}$$

$$\text{(Config)}\ \frac{\Gamma \vdash_{A,B} q_1 \Subset q'_1 \quad \ldots\ldots \quad \Gamma \vdash_{A,B} q_n \Subset q'_n}{\Gamma \vdash_{A,B} f(q_1, \ldots, q_n) \Subset f(q'_1, \ldots, q'_n)}$$

Given $\mathcal{Q}_{F_\mathcal{A}}$ and $\mathcal{Q}_{F_\mathcal{B}}$ the sets of final states of $\mathcal{A}$ and $\mathcal{B}$, $\#()$ a symbol of arity 1 not occurring in $\mathcal{F}$, to prove $\mathcal{L}(\mathcal{A}) \subseteq \mathcal{L}(\mathcal{B})$, we start our deduction by the statement: $\emptyset \vdash_{\mathcal{A},\mathcal{B}} \{\#(q) \mid q \in \mathcal{Q}_{F_\mathcal{A}}\} \Subset \{\#(q) \mid q \in \mathcal{Q}_{F_\mathcal{B}}\}$

Example 1. Let $\mathcal{A}$ and $\mathcal{B}$ be two automata s.t.:

$$\mathcal{A} = \left\{ \begin{array}{r} a \to q_1 \\ b \to q_2 \\ f(q_1, q_2) \to \mathbf{q} \end{array} \right\} \text{ with } \mathcal{Q}_{F_\mathcal{A}} = \{\mathbf{q}\} \text{ and } \mathcal{B} = \left\{ \begin{array}{r} a \to \mathbf{q}' \\ b \to \mathbf{q}' \\ f(q', q') \to \mathbf{q}' \end{array} \right\} \text{ with } \mathcal{Q}_{F_\mathcal{B}} = \{\mathbf{q}'\}$$

Here we have $\mathcal{L}(\mathcal{A}) \subseteq \mathcal{L}(\mathcal{B})$ and we can derive $\emptyset \vdash_{\mathcal{A},\mathcal{B}} \#(q) \Subset \#(q')$ with the deduction rules:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\{q \Subset q', q_1 \Subset q'\} \vdash_{\mathcal{A},\mathcal{B}} a() \Subset a()}\text{(Const.)}}{\{q \Subset q',\ q_1 \Subset q'\} \vdash_{\mathcal{A},\mathcal{B}} a() \Subset \{a(), b(), f(q', q')\}}\text{(Weak-r)}}{\{q \Subset q'\} \vdash_{\mathcal{A},\mathcal{B}} q_1 \Subset q'}\text{(Induction)} \qquad \dfrac{\dfrac{\dfrac{}{\{q \Subset q', q_2 \Subset q'\} \vdash_{\mathcal{A},\mathcal{B}} b() \Subset b()}\text{(Const.)}}{\{q \Subset q',\ q_2 \Subset q'\} \vdash_{\mathcal{A},\mathcal{B}} b() \Subset \{a(), b(), f(q', q')\}}\text{(Weak-r)}}{\{q \Subset q'\} \vdash_{\mathcal{A},\mathcal{B}} q_2 \Subset q'}\text{(Induction)}}{\{q \Subset q'\} \vdash_{\mathcal{A},\mathcal{B}} f(q_1, q_2) \Subset f(q', q')}\text{(Config)}}{\{q \Subset q'\} \vdash_{\mathcal{A},\mathcal{B}} f(q_1, q_2) \Subset \{a(), b(), f(q', q')\}}\text{(Weak-r)}}{\emptyset \vdash_{\mathcal{A},\mathcal{B}} q \Subset q'}\text{(Induction)}}{\emptyset \vdash_{\mathcal{A},\mathcal{B}} \#(q) \Subset \#(q')}\text{(Config)}$$

The main property we want to demonstrate in Coq is that this syntactic criterion implies the semantic inclusion for the considered languages in 6.

```
Theorem inclusion_sound :
  ∀ A B, inclusion A B = true → Included A B.
```

Before proving this in Coq, we need to define more formally the function `inclusion`. This function cannot be defined as a simple structural recursion. Thus Coq needs a termination proof for this algorithm. Thanks to the Coq feature `Function`, it is possible to define the algorithm using a measure function and provide a proof that its value decreases at each recursive call to ensure the termination.

Termination of deduction rules can be proved by defining a measure function μ on statements of the form $\Gamma \vdash_{\mathcal{A},\mathcal{B}} \alpha \Subset \beta$. The Γ relation can be seen as a subset of $\mathcal{Q}_A \times \mathcal{Q}_B$ which is a finite set. All tree automata have a finite number of states. Then the statement measure $\mu(\Gamma \vdash_{A,B} \alpha \trianglelefteq \beta)$ is defined as tuple $(\mu_1(\Gamma), \mu_2(\alpha) + \mu_2(\beta))$ where:

$$\begin{cases} \mu_1(\Gamma) = |\mathcal{Q}_A \times \mathcal{Q}_B| - |\Gamma| \\ \mu_2(x) = \begin{cases} (m+1-n) \text{ if } x = \{c_i\}_n^m \\ 1 \text{ if } x = f(q_1, \ldots, q_n), \\ 0 \text{ otherwise} \end{cases} \end{cases}$$

Then we define the ordering $\ll$ by the lexicographic combination of the usual order $<$ on natural numbers for μ_1 and μ_2. Since $<$ is well founded, the lexicographic combination $\ll$ is also well founded.

Theorem 1. *(Termination) At each deduction step, the measure decreases strictly:*

$$\frac{\Gamma_1 \vdash_{A,B} \alpha_1 \Subset \beta_1 \ldots \Gamma_n \vdash_{A,B} \alpha_n \Subset \beta_n}{\Gamma' \vdash_{A,B} \alpha' \Subset \beta'} \Longrightarrow \bigwedge_{i=1}^{n} \mu(\Gamma_i \vdash_{A,B} \alpha_i \Subset \beta_i) \ll \mu(\Gamma' \vdash_{A,B} \alpha' \Subset \beta')$$

Proof. See [BGJ08], for details.

Theorem 2. *(Soundness) For all tree automata $\mathcal{A}$ and $\mathcal{B}$, if there exists $\prod$ a proof tree of $\emptyset \vdash_{\mathcal{A},\mathcal{B}} q \Subset q'$ then we have $\mathcal{L}(\Delta_{\mathcal{A}}, q) \subseteq \mathcal{L}(\Delta_{\mathcal{B}}, q')$*

Proof. This can be done by an induction on the size of the term of $\mathcal{L}(\Delta_{\mathcal{A}}, q)$. See [BGJ08] for details.

As said above, the described algorithm is not complete in general. However, we show that it is complete for tree automata produced by completion. In particular if $\mathcal{A}^k_{\mathcal{R}}$ is obtained after k completion step from $\mathcal{A}^0$ then we can build a proof $\prod$ for the statement $\emptyset \vdash_{\mathcal{A}^0,\mathcal{A}^k_{\mathcal{R}}} \{\#(q) \mid q \in \mathcal{Q}_{F_0}\} \Subset \{\#(q') \mid q' \in \mathcal{Q}_{F_k}\}$. Recall that the tree automaton produced by the k^{th} step of completion is noted $\mathcal{A}_k = \langle \mathcal{F}, \mathcal{Q}_k, \mathcal{Q}_{F_k}, \Delta_k \rangle$. The tree automata completion performs two main operations at each step of calculus: *normalization* and *state merging*. In the case of normalization, the language inclusion can simply be proved using transition set inclusion. With the state merging operation, set inclusion is not enough because it implies transition merging too. This is the reason why we have to define a new order relation preserved by each operation.

Definition 6. *Given $\mathcal{A}$, $\mathcal{B}$ two tree automata, $\sqsubseteq$ is the reflexive and transitive relation defined as follows: $\mathcal{A} \sqsubseteq \mathcal{B}$ if there exists a function ϱ that renames states of $\mathcal{A}$ into states of $\mathcal{B}$ and such that all renamed rules $\Delta_{\mathcal{A}}$ are contained in $\Delta_{\mathcal{B}}$:*

$$\mathcal{A} \sqsubseteq \mathcal{B} \Longleftrightarrow \exists \varrho : \mathcal{Q}_{\mathcal{A}} \to \mathcal{Q}_{\mathcal{B}},\ \varrho(\Delta_{\mathcal{A}}) \subseteq \Delta_{\mathcal{B}}\ \wedge\ \varrho(\mathcal{Q}_{F_{\mathcal{A}}}) \subseteq \mathcal{Q}_{F_{\mathcal{B}}} \tag{1}$$

Lemma 2. *Given a tree automaton $\mathcal{A}$,*

1. *if $\mathcal{A}' = \mathcal{A} \cup Norm(r\sigma \to q)$ then $\mathcal{A} \sqsubseteq \mathcal{A}'$*
2. *if $\mathcal{A}' = Merge(\mathcal{A}, q_1, q_2)$ then $\mathcal{A} \sqsubseteq \mathcal{A}'$*

Proof. For details, see [BGJ08].

Theorem 3. *Given a tree automaton $\mathcal{A}^0$, a TRS $\mathcal{R}$ and an equation set $\mathcal{E}$, after k completion steps we obtain $\mathcal{A}^k_{\mathcal{R}}$ such that $\mathcal{A}^0 \sqsubseteq \mathcal{A}^k_{\mathcal{R}}$.*

Proof. Since we have proved that $\sqsubseteq$ is preserved by $Norm$ and $Merge$ functions, it is also the case for every completion step between $\mathcal{A}^k_{\mathcal{R}}$ and $\mathcal{A}^{k+1}_{\mathcal{R}}$, i.e $\mathcal{A}^k_{\mathcal{R}} \sqsubseteq \mathcal{A}^{k+1}_{\mathcal{R}}$. Then, the theorem can be deduced using the reflexivity and transitivity of $\sqsubseteq$. See [BGJ08].

Now, we define the completeness property as the following:

Theorem 4. *(Completeness) Given two tree automata $\mathcal{A}$ and $\mathcal{B}$ if $\mathcal{A} \sqsubseteq \mathcal{B}$ then there exists $\prod$ a proof of statement $\emptyset \vdash_{\mathcal{A},\mathcal{B}} \{\#(q_f) \mid q_f \in \mathcal{Q}_{F_A}\} \Subset \{\#(q'_f) \mid q'_f \in \mathcal{Q}_{F_B}\}$.*

Proof. For details, see [BGJ08].

Thus, we can ensure that for an automaton $\mathcal{A}^k_{\mathcal{R}}$ obtained by k completion steps from $\mathcal{A}^0$, there exists a proof $\prod$ of the statement $\emptyset \vdash_{\mathcal{A}^0,\mathcal{A}^k_{\mathcal{R}}} \{\#(q) \mid q \in \mathcal{Q}_{F_0} \Subset \{\#(q') \mid q' \in \mathcal{Q}_{F_k}\}$. This can be obtained by a simple combination of the two previous theorems.

Finally, as shown in [BGJ08], this algorithm has a polynomial complexity w.r.t. space. Using tabling, it can also be implemented so as to be polynomial in time. However, since proofs are more difficult to carry out in Coq on a tabled version, we chose to stick to a simpler implementation that appears to be sufficient for our test cases (see Section 9).

8 Formalization of closure by rewriting

In this part we aim at defining formally the `IsClosed` predicate, the function `closure` and prove the soundness of this function w.r.t. `IsClosed`. Recall that to check if a tree automaton $\mathcal{A} = \langle \mathcal{Q}_F,\ \Delta \rangle$ is closed w.r.t. a TRS $\mathcal{R}$, it is enough to prove that for all $t \in \mathcal{L}(\mathcal{A})$, if t' is reachable from t by $\to^*_{\mathcal{R}}$ then $t' \in \mathcal{L}(\mathcal{A})$. Now that we have defined in Coq rewriting and tree automata, we can define more formally the `IsClosed` predicate and recall the `closure_sound` theorem to prove:

```
Definition IsClosed (R : list rule) (A : t_aut) : Prop :=
  ∀ q t t', IsRec A.delta q t → Reachable R t t' → IsRec A.delta q t'.

Theorem closure_sound:
    ∀ R A', LeftLinear R → closure R A' = true → IsClosed R A'.
```

The algorithm to check closure of $\mathcal{A}$ by $\mathcal{R}$ computes for each left-linear rule $l \to r \in \mathcal{R}$ the full set of the substitutions σ s.t. $l\sigma \to_{\Delta}^{*} q$ and then, checks that $r\sigma \to_{\Delta}^{*} q$. Then, the correctness proof consists in showing that if `closure` answers true, then $\mathcal{L}(\mathcal{A})$ is closed by $\to_{\mathcal{R}}$. We now give some hints to define the `closure` function. First, for all left-linear rule $l \to r$ of $\mathcal{R}$, this function has to find all the substitutions $\sigma : \mathcal{X} \mapsto \mathcal{Q}$ and all the states $q \in \mathcal{Q}$ such that $l\sigma \to_{\Delta}^{*} q$. This is what we call the *matching-problem*. Second, this function has to check that for all the q and σ found, we have $r\sigma \to_{\Delta}^{*} q$. Third, in the correctness theorem, we have to show that all the substitutions $\sigma : \mathcal{X} \mapsto \mathcal{Q}$ cover the set of substitutions on terms, i.e. of the form $\sigma' : \mathcal{X} \mapsto \mathcal{T}(\mathcal{F})$, and hence cover all reachable terms.

We note $l \trianglelefteq q$ the matching problem consisting in finding all the substitutions $\sigma : \mathcal{X} \mapsto \mathcal{Q}$ and all the states $q \in \mathcal{Q}$ such that $l\sigma \to_{\Delta}^{*} q$. An algorithm solving this kind of problems was defined in [Gen97]. Note that it is complete only if l is linear. The algorithm consists in normalizing the formula $l \trianglelefteq q$ with the following deduction rules:

$$\text{(Unfold)}\ \frac{f(s_1,\ldots,s_n) \trianglelefteq f(q_1,\ldots,q_n)}{s_1 \trianglelefteq q_1 \wedge \cdots \wedge s_n \trianglelefteq q_n} \qquad \text{(Clash)}\ \frac{f(s_1,\ldots,s_n) \trianglelefteq g(q'_1,\ldots q'_m)}{\bot}$$

$$\text{(Config)}\ \frac{s \trianglelefteq q}{s \trianglelefteq c_1 \vee \cdots \vee s \trianglelefteq c_k \vee \bot} \text{if } s \notin \mathcal{X}, \text{ and } \forall c_i,\ s.t.\ c_i \to q \in \Delta.$$

Moreover, after each application of one of these rules, the result is also rewritten into disjunctive normal form. When normalization of the initial problem is terminated, we obtain a formula like $\bigvee_{i=1}^{n} \phi_i$ where $\phi_i = \bigwedge_{j=1}^{m} x_j^i \trianglelefteq q_j^i$ such that $x_j^i \in \mathcal{X}$ and $q_j^i \in \mathcal{Q}$. Each ϕ_i can be seen as a substitution $\sigma_i = \{x_j^i \mapsto q_j^i\}$. The implementation of the `matching` function in Coq is very close to this algorithm. Moreover, the soundness and completeness properties of this algorithm can be defined in Coq as follows:

```
Theorem matching_sound :
  ∀ D q l s, In s (matching D q l) → IsRed D q (l @ s).

Theorem matching_complete :
  ∀ D q l s, linear l → IsRed D q (l @ s) → In s (matching D q l).
```

As mentioned before, the `matching` algorithm is only complete for linear terms. Thus, this assumption occurs in the `matching_complete` theorem as well as in all theorems using the left-side of a rule. The second part of the `closure` function consists in verifying that for each substitution σ s.t. $l\sigma \to_{\Delta}^{*} q$, we also have $r\sigma \to_{\Delta}^{*} q$. This job is performed using the `all_red` function, we

define, whose purpose is to check that this property is true for all the found substitutions. Then, we only need to prove the soundness of this function using the following Coq theorem:

```
Theorem all_red_sound :
  ∀ D q r sigmas,
    all_red D q r sigmas = true → ∀ s, In s sigmas → IsRed D q (r@s).
```

By combining the `matching` and the `all_red` functions, we obtain `closure_at_state` the function for checking up all critical pairs found at state q and for the rule $l \to r$. We define the combination as:

```
Definition closure_at_state D q l r := all_red D q r (matching D q l).

Theorem closure_at_state_sound :
   ∀ D q l r, linear l → closure_at_state D q l r = true →
      (∀ s, IsRed D q (l @ s) → IsRed D q (r @ s)).
```

Given a left-linear rule $l \to r$ and a state q, this algorithm answers `true` if for all substitution $\sigma : \mathcal{X} \mapsto \mathcal{Q}$ s.t. $l\sigma \to_\Delta^* q$ then $r\sigma \to_\Delta^* q$. Now that we have proved this result for substitutions $\sigma : \mathcal{X} \mapsto \mathcal{Q}$, we have to prove that it implies the same property for substitutions $\sigma' : \mathcal{X} \mapsto \mathcal{T}(\mathcal{F})$, this is Lemma 3. On the opposite, to prove that every reachable term of $\mathcal{T}(\mathcal{F})$ will be covered by a configuration of $\mathcal{T}(\mathcal{F} \cup \mathcal{Q})$ in Δ, we have to prove that if there exists a substitution $\sigma' : \mathcal{X} \mapsto \mathcal{T}(\mathcal{F})$, then we can construct a corresponding substitution $\sigma : \mathcal{X} \mapsto \mathcal{Q}$, this is Lemma 4.

Lemma 3. *Given a term $u \in \mathcal{T}(\mathcal{F}, \mathcal{X})$, $\sigma : \mathcal{X} \mapsto \mathcal{Q}$ a substitution s.t. $u\sigma \to_\Delta^* q$, if we have a substitution $\sigma' : \mathcal{X} \mapsto \mathcal{T}(\mathcal{F})$ s.t. $\forall x \in \mathcal{D}om(\sigma) : \sigma' x \in \mathcal{L}(\Delta, \sigma x)$, then we have $u\sigma' \to_\Delta^* q$ and thus $u\sigma' \in \mathcal{L}(\Delta, q)$.*

Roughly, if the left or right-hand side u of a rewriting rule matches a configuration $u\sigma \in \mathcal{T}(\mathcal{F} \cup \mathcal{Q})$ and $u\sigma \to_\Delta^* q$ then, all terms $u\sigma' \in \mathcal{T}(\mathcal{F})$, matched by u, are also reducible into q, i.e. $u\sigma' \to_\Delta^* q$ and $u\sigma' \in \mathcal{L}(\Delta, q)$.

Lemma 4. *Given a term $u \in \mathcal{T}(\mathcal{F}, \mathcal{X})$, if there exists a substitution $\sigma' : \mathcal{X} \mapsto \mathcal{T}(\mathcal{F})$ such that $u\sigma' \to_\Delta^* q$, then there exists a substitution $\sigma : \mathcal{X} \mapsto \mathcal{Q}$ s.t. $\sigma' x \in \mathcal{L}(\Delta, \sigma x)$ and $u\sigma \to_\Delta^* q$.*

Using those two lemmas, we can conclude that for all term $t \in \mathcal{L}(\Delta, q)$ rewritten in t' at the topmost position by $l \to r$, then $t' \in \mathcal{L}(\Delta, q)$. This property is easily lifted as a property of the `closure` function for all states of $\mathcal{Q}$ and using all rules of $\mathcal{R}$ at topmost position. Then, it is enough to lift this property to general rewriting at any position. Finally, the `closure_sound` general theorem is shown by using a reflexive and transitive application of the last property.

9 Benchmarks

From the Coq formal specification (about 2000 lines for definitions and 5500 lines for proofs), we have extracted an Ocaml checker implementation which is

connected to the Timbuk parser. Since Coq extraction ignore all Ocaml data types (integers, lists, maps...) and redefine all them (including primitive types). Thus, we defined a set of functions to convert Coq types into Ocaml types and conversely.

In the following table, we have collected several benchmarks. For each test, we give the size of the two tree automata (initial $\mathcal{A}^0$ and completed $\mathcal{A}^*_{\mathcal{R}}$) as number of transitions/number of states. For each TRS $\mathcal{R}$ we give the number of rules. The 'CS' column gives the number of completion steps necessary to complete $\mathcal{A}^0$ into $\mathcal{A}^*_{\mathcal{R}}$ and 'CT' gives the completion time. The 'CKT' column gives the time for the checker to certify the $\mathcal{A}^*_{\mathcal{R}}$ and the 'CKM' gives the memory usage. The important thing to observe here is that, the completion time is very long (sometimes more than 24 hours), the *checking* of the corresponding automaton is always fast (a matter of seconds).

The four tests are Java programs translated into term rewriting systems using the technique detailed in [BGJL07]. All of them are completed using Timbuk except the example `List2.java` which has been completed using a new optimized completion tool detailed in [BBGM08]. In this last paper, the completion times are 10 to 100 times better than using Timbuk. Even if the input and output of this tool are tree automata, the internal computation mechanism is exclusively based on term rewriting and uses no tree automata algorithms. This shows that the completed automaton produced by a totally different algorithm and fully optimized tool is also accepted by our checker. The `List1.java` and `List2.java` corresponds to the same Java program but with slightly different encoding into TRS and approximations. The `Ex_poly.java` is the example given in [BGJL07] and the `Bad_Fixp` is the same problem as `Ex_poly.java` except that the completed automaton $\mathcal{A}^*_{\mathcal{R}}$ has been intentionally corrupted. Thus, this is thus not a valid fixpoint and rejected by the checker.

Name	$\mathcal{A}^0$	$\mathcal{A}^*_{\mathcal{R}}$	$\mathcal{R}$	CS	CT	CKT	CKM
`List1.java`	118/82	422/219	228	180	$\approx$ 3 days	0,9s	2,3 Mo
`List2.java`	1/1	954/364	308	473	1h30	2,2s	3,1 Mo
`Ex_poly.java`	88/45	951/352	264	161	$\approx$ 1 day	2,5s	3,3 Mo
`Bad_Fixp`	88/45	949/352	264	161	$\approx$ 1 day	1,6s	3,2 Mo

10 Conclusion and further research

In this paper we have defined a Coq checker for tree automata completion. The first characteristic of the work presented here is that the checker does not validate a specific implementation of completion but, instead, the result. As a consequence, the checker remains valid even if the implementation of the completion algorithm changes or is optimized. For example, this checker could be used to certify tree automata produced by [Jac96,Ret99] and [Tak04] for left-linear TRS, provided that ϵ-transitions are normalized first. This is quite natural since the behavior of those algorithms is close to tree automata completion. We gave an even more significant example of the independence of the checker w.r.t. the used

completion algorithm by certifying results produced by [BBGM08] whose algorithm is not based on tree automata.

A second salient feature of the checker is that its code is directly generated from the correctness proof of its Coq specification through the extraction mechanism. Third, we have payed particular attention to the formalization of the checker in order not to lose efficiency to obtain the certification. We have defined a specific inclusion algorithm in order to avoid the usual exponential blow-up obtained with the standard inclusion algorithm. We have defined the Coq formal specification so that it was possible to extract an independent OCaml checker. Finally, we made an extensive use of efficient formal data structures leading to more complex proof but also to faster extracted checker. An extension for non left-linear TRS, which are necessary for specifying cryptographic protocols, is under development. Since many different kinds of analyzes can be expressed as reachability problems over tree automata, and since verification of completed automata revealed to be very efficient, we aim at using this technique in a PCC architecture where tree automata are viewed as program certificates. At last, note that even if this checker is external to Coq, we can use the correction proof of the checker and the Coq reflexivity mechanism to lift-up the external verification into a proof in the Coq system. This can be necessary to perform efficient unreachability proofs on rewriting systems in Coq using an external completion tool.

References

[ABB+05] A. Armando, D. Basin, Y. Boichut, Y. Chevalier, L. Compagna, J. Cuellar, P. Hankes Drielsma, P.-C. Héam, O. Kouchnarenko, J. Mantovani, S. Mödersheim, D. von Oheimb, M. Rusinowitch, J. Santos Santiago, M. Turuani, L. Viganò, and L. Vigneron. The AVISPA Tool for the automated validation of internet security protocols and applications. In *CAV'2005*, volume 3576 of *LNCS*, pages 281–285. Springer, 2005.

[BBGM08] E. Ballad, Y. Boichut, T. Genet, and P.-E. Moreau. Towards an Efficient Implementation of Tree Automata Completion. In *AMAST'08*, 2008. To be published.

[BC04] Y. Bertot and P. Castéran. *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. Springer Verlag, 2004.

[BD04] G. Barthe and G. Dufay. A tool-assisted framework for certified bytecode verification. In *FASE'04*, volume 2984 of *LNCS*, pages 99–113. Springer, 2004.

[BGJ08] B. Boyer, T. Genet, and T. Jensen. Certifying a Tree Automata Completion Checker. Technical Report RR 6462, INRIA, 2008. `http://hal.inria.fr/inria-00258275/fr/`.

[BGJL07] Y. Boichut, T. Genet, T. Jensen, and L. Leroux. Rewriting Approximations for Fast Prototyping of Static Analyzers. In *RTA*, volume 4533 of *LNCS*, pages 48–62. Springer Verlag, 2007.

[BHK04] Y. Boichut, P.-C. Héam, and O. Kouchnarenko. Automatic Approximation for the Verification of Cryptographic Protocols. In *Proc. AVIS'2004, joint to ETAPS'04, Barcelona (Spain)*, 2004.

[BJP06] F. Besson, T. Jensen, and D. Pichardie. Proof-carrying code from certified abstract interpretation and fixpoint compression. *Theor. Comput. Sci*, 364(3):273–291, 2006.

[BN98] F. Baader and T. Nipkow. *Term Rewriting and All That.* Cambridge University Press, 1998.

[CDG+02] 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.

[CJPR05] D. Cachera, T. Jensen, P. Pichardie, and V. Rusu. Extracting a data flow analyser in constructive logic. *Theor. Comput. Sci.*, 342(1):56–78, 2005.

[FGVTT04] G. Feuillade, T. Genet, and V. Viet Triem Tong. Reachability Analysis over Term Rewriting Systems. *JAR*, 33 (3-4):341–383, 2004.

[Gen97] T. Genet. Decidable approximations of sets of descendants and sets of normal forms (extended version). Technical Report RR-3325, INRIA, 1997.

[Gen98] 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.

[GK00] 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.

[GT95] R. Gilleron and S. Tison. Regular tree languages and rewrite systems. *Fundamenta Informaticae*, 24:157–175, 1995.

[GTTVTT03] T. Genet, Y.-M. Tang-Talpin, and V. Viet Triem Tong. Verification of Copy Protection Cryptographic Protocol using Approximations of Term Rewriting Systems. In *WITS'2003*, 2003.

[GVTT00] 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/`.

[Jac96] 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.

[KN03] G. Klein and T. Nipkow. Verified bytecode verifiers. *TCS*, 298, 2003.

[LT00] P. Letouzey and L. Théry. Formalizing stalmarck's algorithm in coq. In *Proc. of TPHOL'00*, volume 1869 of *LNCS*. Springer, 2000.

[Ret99] P. Rety. Regular Sets of Descendants for Constructor-based Rewrite Systems. In *Proc. 6th LPAR Conf., Tbilisi (Georgia)*, volume 1705 of *LNAI*. Springer-Verlag, 1999.

[RGL01] X. Rival and Jean Goubault-Larrecq. Experiments with finite tree automata in coq. In *Proc. of TPHOL'01*, LNCS. Springer, 2001.

[Tak04] 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.

[ZD06] R. Zunino and P. Degano. Handling exp, × (and timestamps) in protocol analysis. In *Proc. of FOSSACS'06*, volume 3921 of *LNCS*, pages 413–427. Springer, 2006.