Certificates for Tree Automata Completion *

Roberto Zunino

Dipartimento di Informatica e Telecomunicazioni, Università di Trento, Italy
Dipartimento di Informatica, Università di Pisa, Italy
zunino@disi.unitn.it

Abstract. We consider the problem of certifying the correctness of a protocol or security API through a formal, machine-checkable proof. To this aim, we re-examine the completion algorithm for tree automata and term rewriting systems, that computes an over-approximation of regular sets of terms up to rewriting. We then define a procedure to certify, via a proof, that the result of the completion is indeed correct. Hence, with our procedure, a program verification tool that uses the completion algorithm can certify its results. As a consequence, we do not need to regard the tool as a trusted component of the verification process. We discuss on our experiments in which we automatically generated security proofs for some selected protocols.

1 Introduction

Static analyses based on tree automata [3] and their completion [6,12,16] have been proposed [2,17] and used in tools [1,11,13] for many purposes, notably cryptographic protocol verification [14,7]. These static verification tools are indeed useful, providing a very high level of confidence on the correctness of the system at hand. However, in this approach to system verification, the correctness of the verification tool itself is crucial. When one of these tools claim *"the system is correct"*, one should actually add the premise *"as long as the used tool is correct"*. A more cautious approach would then be regarding these tools as reducing the correctness problem of the system to that of the tool. While it is quite common to include the verification tool in the trusted computing base, this might not be advisable, especially when the tool is quite a complex system.

Using a small trusted base would greatly improve the confidence on the tool results. In this work, we present a way to remove our Rewrite tool [11] from the trusted base, by reducing its correctness to that of the Coq proof assistant [4]. We stress that our technique is general: we can certify any reachability result produced by Rewrite. We also see no major obstacles in adapting our technique to other tools based on the completion algorithm as well.

We considered proving Rewrite correct using Coq. Many obstacles hinder this path. First, Rewrite consists of roughly 5000 lines of Haskell source code, that

* Partially supported by the EU within the FETPI Global Computing, project IST-2005-16004 SENSORIA.

must be proved correct. Second, defining in Coq the semantics of Haskell is a daunting task on its own. Third, one must either assume the correctness of all the software Rewrite depends on, or prove it correct. The latter seems to be practically unfeasible: it involves proving the correctness of the Haskell libraries, the Haskell compiler, the C system libraries, the operating system kernel, etc.

We instead chose a simpler alternative: rather than proving Rewrite correct, we modified Rewrite to provide a Coq correctness proof of its own outputs. That is, when Rewrite statically establishes a property on its input, it states "the system is correct" *and* it provides a Coq proof for it. In other words, Rewrite certifies its own claims. Note that this does not rule out the presence of bugs in Rewrite or the other software mentioned above. However, when the certificate is validated by Coq, the single component we trust, then the property holds, and the way we generated the certificate is then irrelevant.

Our approach is clearly related to *proof carrying code* [10]: our proofs on tree automata can be seen as proofs on approximations of programs. In this scenario the completion tool plays the rôle of the program compiler, which can be instrumented so that it generates proofs for the program being compiled.

Related techniques can be found in the H1 tool [9] of Goubault-Larrecq. The H1 tool is actually a library of transformations over tree automata, seen as sets of Horn clauses in the $\mathcal{H}1$ class. In particular, the h1mc component is able to generate unreachability proofs for tree automata in Coq [8]. Our approach is similar to that one, except that we focus on proving the correctness of the completion of a tree automaton w.r.t. a term rewriting system. This completion algorithm is not included in the H1 suite. Other tools, e.g. Timbuk [13], TA4SP [2] do exploit the completion, but do not generate proofs.

We experimented with some security protocols and APIs. Notably, we re-examined the protocol based on Diffie-Hellman we discussed in [17]. A proof for the forward secrecy property guaranteed by this protocol was generated by Rewrite in 6 seconds, and is about 1600 lines long. On the other hand, Coq took 4 minutes to check the generated file was indeed a correct proof. The full specification and proof can be found at [11].

A nice comparison for our technique is offered by Courant and Monin [5]: they proved in Coq the security of a cryptographic API used in ATMs. Their manually-written proof is about 2100 lines long. Our Rewrite tool has been used to prove the *same* security property [15]. The automatically generated Coq proof is about 3200 lines long. However, while Coq verifies the manually-written proof in a few seconds, ours requires more time to be checked: on our machine this took 24 minutes. While this running time is still acceptable, it is still a couple of orders of magnitude over the time for the manual proof. To compare these results in a more fair way one must also take into account the time spent by humans on writing proofs: surely [5] required much more effort and time to be written.

Summary We introduce the problem and our contributions in Sect. 2, using intuitive arguments. We give the formal definitions in Sect. 3. The completion algorithm in then briefly discussed in Sect. 4. The generation of Coq proofs is described in Sect.5.

2 The Problem

In this section, we introduce tree automata, term rewritings, and our contribution. Rather than starting with the formal definitions for them, we start from a motivating example: a simple program static analysis. We show how our results can be applied to this scenario, providing as little technical details as possible, and rather focusing on intuition.

To keep our presentation simple, we consider a program excerpt involving lists. More complex, security-related examples can be found at [11]. Lists are built through the constructors $\mathsf{cons}(\bullet, \bullet), \mathsf{nil}$ as usual, but we shall also use the more compact notation $[x_1, \ldots, x_n]$ for cons lists. Below, variables a,b,c have list values, and suffix, filter are functions we shall describe shortly.

```
a := Input; b := suffix(a); c := filter(b); Output c
```

We want to compute a static approximation of the behaviour of the code above. For this, we assume to have previously inferred some information about the input a, and the functions suffix, filter. From this information, we plan to deduce some property about the output c. More in detail:

- Input a is known to be a list of bits in $\{0, 1\}$, i.e. a bit string. More precisely, we know the value of a to be in the regular set $(10|0)^*$.
- We only have some partial information about function suffix. while we do not know its exact dynamic semantics, it is known to return a suffix of its argument.
- Function filter returns its list argument, dropping each element that is not immediately followed by a duplicate of itself. For instance, $\mathsf{filter}([1, 1, 0, 1, 0, 0]) = [1, 0]$.
- Finally, our goal is to show that output c is not a list beginning with 01, that is: not of the form $[0, 1, \ldots]$.

First, we should convince ourselves that the goal can really be inferred from the above information. Intuitively, a suffix of $(10|0)^*$ can never contain two consecutive 1 bits, since each 1 is followed by 0. So, each 1 bit will be removed by filter and will not end up in variable c. This implies that c can not be $[0, 1, \ldots]$, which is our goal.

Now, we show how to derive that result in a fully automatic way from a formal specification of our scenario. We start by formalizing our knowledge about the input a. To this purpose, we use a *tree automaton*, with the following transitions:

$$@a \to nil \qquad @a \to cons(1, cons(0, @a)) \qquad @a \to cons(0, @a)$$

Its meaning should be rather intuitive. The language of the automaton state $@a$ is formed by lists, and is inductively defined by the transitions above. The first transitions provides the base case, while the following ones allow one to combine a list in the language with the prefixes $1, 0$ or 0, respectively. Clearly, the language of $@a$ is $(10|0)^*$.

We can now model out knowledge about the function suffix, and the related assignment in the program itself. This can be done by augmenting the tree automaton as follows:

$$@b \to @a \qquad @b \to tail(@b)$$

The language of the new automaton state $@b$ contains the language of $@a$, and is closed under the tail operation, which we shall define now so to return the second component of a cons cell. Therefore, $@b$ includes all the suffixes of $@a$. To properly define tail, we use a *rewriting rule*.

$$tail(cons(X, Xs)) \Rightarrow Xs$$

Finally, we need to tackle the last assignment in the program, involving the filter function. A simple automaton transition suffices:

$$@c \to filter(@b)$$

Of course, now we need rewriting rules for the filter operation.

$$\begin{array}{l} filter(nil) \Rightarrow nil \\ filter(cons(X, nil)) \Rightarrow nil \\ filter(cons(X, cons(X, Xs))) \Rightarrow cons(X, filter(cons(X, Xs))) \\ filter(cons(0, cons(1, Xs))) \Rightarrow filter(cons(1, Xs)) \\ filter(cons(1, cons(0, Xs))) \Rightarrow filter(cons(0, Xs)) \end{array}$$

The above is an inductive definition for dropping all the elements of a list that have no duplicate following them. The first two rules handle the base cases (length ≤ 1). The third rule handles the duplicates. Note that this rule has two occurrences of the variable X in its left hand side (such a rule therefore is non-linear). Finally, the last two rules handle the non-duplicate case, for the bits 0 and 1.

Now, we must state the goal of our analysis: $@c$ must not contain values of the form $[0, 1, \ldots]$. First, to make it more convenient, we explicitly mark all the unwanted values with a failure constructor.

$$cons(0, cons(1, X)) \Rightarrow failure(cons(0, cons(1, X)))$$

Then, we simply require that $@c$ has no failure($\bullet$) value. Technically, this is done in Rewrite with the line

```
| @c : failure(X) => @zzResult : !FAIL!
```

The actual meaning of the goal above can more precisely be stated as follows: we do not have $@c \to^* \Rightarrow^*$ failure(X), for any instantiation of X. So, our goal states the *unreachability* of a class of terms from the state $@c$, through automaton transitions and term rewritings. We shall formalize this in Sect. 3.

Above, we showed both the transitions forming a tree automaton, and the rewriting rules of a term rewriting system. These form a complete specification

that can be passed to Rewrite. Indeed, the actual Rewrite input file contains *only* the lines shown above, and a declaration of each term constructor and the associated arity.

Running Rewrite on this specification quickly produces the result of the static analysis: there are indeed no failure(•) terms in @c. At the same time, a Coq proof is produced, stating the same fact, in a reasonably human-readable form:

```
Lemma Unreachability_1 : ~ exists x1, O_zac (h_failure x1) .
```

In Coq, @ is rendered as `O_za`, so `O_zac` stands for @c, while term constructors ("heads") are prefixed with `h_`. Aside from this minor encoding, the meaning of the lemma is very immediate and clear. Of course, one does need to check that the definition of `O_zac` in the Coq file indeed corresponds to the intended one: the language of @c, closed under all the term rewriting rules. Since this check must be performed by manual inspection or with the help of a small trusted tool, it is very important to make it as simple as possible. In other words, we must keep the definitions in the Coq file as close as possible to the ones in the original specification, even if this makes the proofs more complicated or harder to generate.

We indeed claim our formalization in Coq to be very close to the original specification. We now support this statement by showing some excerpts from the Coq file generated by Rewrite. Here is a part of the definition of the automaton, listing the inductive definition for `O_zac`.

```
Inductive [...] O_zac : St := [...]
   | T_O_90 : forall x1, O_zab x1 -> O_zac (h_filter x1)
   | T_O_91 : forall t1 t2, O_zac t1 -> RCs t1 t2 -> O_zac t2
```

The first line models the @c transition we showed above: when term $x1$ belongs to (the language of) the state `O_zab` (@b), then filter($x1$) must belong to `O_zac` (@c).

The last line explicitly closes `O_zac` under the rewriting relation `RCs`, in order to close `O_zac` under rewriting. The `RCs` relation is defined as the reflexive transitive closure of another relation `RC`.

```
Inductive RCs : term -> term -> Prop :=
| RCs_S : forall t, RCs t t
| RCs_T : forall t1 t2 t3, RCs t1 t2 -> RC t2 t3 -> RCs t1 t3
```

In turn, relation `RC` is defined to be the closure under (ground) contexts of the relation `R`. This is done by closing it under each term constructor, as it is done for tail below.

```
Inductive RC : term -> term -> Prop := [...]
| RC_R : forall t u, R t u -> RC t u
| RC_h_tail : forall t1 u1, RC t1 u1 -> RC (h_tail t1) (h_tail u1)
```

Finally, `R` is defined from the rewriting rules introduced above. Here is the excerpt related to tail:

```
Inductive R : term -> term -> Prop := [...]
| R_6 :  forall V_X V_Xs , R (h_tail (h_cons V_X V_Xs)) V_Xs
```

As one might guess, `R` relates $\mathsf{tail}(\mathsf{cons}(X, Xs))$ and Xs. Once this relation `R` is closed under contexts and by reflexivity and transitivity, one gets the usual term rewriting relation. Therefore, the relation `RCs` is indeed the intended rewriting relation. Hence, the languages of the automaton are indeed closed under rewritings, and the unreachability lemma shown before correctly models our goal.

3 Background: Term Rewriting and Tree Automata

In this paper we shall use the following denumerable sets: the set $\mathcal{Q}$ of *states* ($@q$, $@a$, $@b$, $@c$, ...) , the set $\mathcal{X}$ of *variables* (X, Y,...) , the set $\mathcal{F}$ of *function symbols* (f, g, 0, 1, ...). Terms are defined as usual.

Definition 1. *The set of* terms $\mathcal{T}$ *is inductively defined by*

$$\begin{array}{lll} T ::= X & & \textit{variable} \\ \quad\; @q & & \textit{state} \\ \quad\; \mathsf{f}(\cdots, T, \cdots) & & \textit{application} \end{array}$$

By definition, we have $\mathcal{Q} \cup \mathcal{X} \subseteq \mathcal{T}$. Moreover, we assume that each function symbol f has a fixed associated arity. Note that the arity can be zero, so that we can use constants. When the arity is zero, we simply write f instead of $\mathsf{f}()$.

The *size* of a term is the number of the applications occurring in it. The *depth* of a term is the maximum number of *nested* applications. For instance, g, $\mathsf{f}(X)$, and $\mathsf{f}(@q)$ have depth and size 1, while $\mathsf{f}(\mathsf{g}, \mathsf{h})$ has depth 2 and size 3, and X and $@q$ have depth and size zero. The variables occurring in a term T are denoted by $\mathsf{vars}(T)$.

Also, we adopt the following conventions: a *ground* term ($\in \mathcal{T}_{\mathsf{gr}}$) is a term with no occurring variables; a *simple* term ($\in \mathcal{T}_{\mathsf{sm}}$) is a term with no occurring states; a *pure* term ($\in \mathcal{T}_{\mathsf{pr}}$) is a term which is both ground and simple; a *plain* term ($\in \mathcal{T}_{\mathsf{pl}}$) is a ground term of depth at most one.

Accordingly, we shall use the sets $\mathcal{C}_{\mathsf{gr}}, \mathcal{C}_{\mathsf{sm}}, \mathcal{C}_{\mathsf{pr}}$ to denote the sets of ground, simple, and pure *contexts* $C[\bullet]$, respectively. That is, $C[\bullet] \in \mathcal{C}_{\mathsf{type}}$ iff $C[T] \in \mathcal{T}_{\mathsf{type}}$, where T is an arbitrary pure term, and type ranges over $\mathsf{gr}, \mathsf{sm}, \mathsf{pr}$. Sometimes, we also use contexts with many placeholders $C[\bullet, \ldots, \bullet]$.

A *substitution* is a function $\sigma : \mathcal{X} \to \mathcal{T}_{\mathsf{gr}}$. We adopt the usual notation σx for the application $\sigma(x)$. Substitutions are homomorphically extended to terms. So, we write σT as the application of the extension of σ to term T.

Below, we give the definitions for rewriting systems and automata.

Definition 2. *A* term rewriting system $\mathcal{R}$ *is a finite set of rewriting rules* $\mathcal{R}$ *having the form* $L \Rightarrow R$*, where* $L, R \in T_{\mathsf{sm}}$ *and* $\mathsf{vars}(R) \subseteq \mathsf{vars}(L)$.

Def. 3 is slightly non standard since we want it to be directly usable in Coq.

Definition 3. *Given a term rewriting system $\mathcal{R}$, we define the relation $\Rightarrow^{\bullet} \subseteq \mathcal{T}_{\mathsf{gr}} \times \mathcal{T}_{\mathsf{gr}}$ as the one obtained from $\Rightarrow$ by instantiating all the variables, i.e. $L' \Rightarrow^{\bullet} R'$ iff*

$$\exists L, R, \sigma \in (vars(L, R) \to \mathcal{T}_{\mathsf{gr}}).\ L' = \sigma L \wedge R' = \sigma R \wedge L \Rightarrow R$$

Similarly, we define the relation $\Rightarrow^{[\bullet]}$ as the reflexive closure under contexts of $\Rightarrow^{\bullet}$. More formally, $\Rightarrow^{[\bullet]}$ is the minimum relation such that

1. $\forall T \in \mathcal{T}_{\mathsf{gr}}.\ T \Rightarrow^{[\bullet]} T$
2. $\forall T_1, T_2 \in \mathcal{T}_{\mathsf{gr}}.\ T_1 \Rightarrow^{\bullet} T_2 \implies T_1 \Rightarrow^{[\bullet]} T_2$
3. $\forall \mathsf{f} \in \mathcal{F}, T_1 \dots T_{\mathsf{arity(f)}} U_1 \dots U_{\mathsf{arity(f)}} \in \mathcal{T}_{\mathsf{gr}}.$
   $(\forall i.\ T_i \Rightarrow^{[\bullet]} U_i) \implies \mathsf{f}(T_1, \dots, T_n) \Rightarrow^{[\bullet]} \mathsf{f}(U_1, \dots, U_n)$

Finally, the relation $\Rightarrow^{[\bullet]*}$ is the transitive closure of $\Rightarrow^{[\bullet]}$.
When we need to make $\mathcal{R}$ explicit, we add it to the above notations, e.g. $\Rightarrow^{[\bullet]*}_{\mathcal{R}}$.

The definition of $\Rightarrow^{\bullet}$ is standard. By comparison, the relation $\Rightarrow^{[\bullet]}$ has a rather complex definition, so we describe it here in more intuitive terms. Basically, the relation $\Rightarrow^{[\bullet]}$ is similar to the closure under contexts of $\Rightarrow^{\bullet}$, which we write as $\Rightarrow_{[\bullet]}$. Using the latter, we have $C[T] \Rightarrow_{[\bullet]} C[U]$ whenever $T \Rightarrow^{\bullet} U$ and $C \in \mathcal{C}_{\mathsf{gr}}$. Instead, $\Rightarrow^{[\bullet]}$ allows *multiple parallel* rewritings to be performed in a context, e.g. relating $C[T_1, T_2]$ with $C[U_1, U_2]$ whenever $T_i \Rightarrow^{\bullet} U_i, \forall i \in \{1, 2\}$. Therefore, while $\Rightarrow_{[\bullet]}$ allows one to rewrite exactly one subterm, the relation $\Rightarrow^{[\bullet]}$ allows to rewrite any number, including zero, of *disjoint* subterms. A simple inductive argument shows that the relations $\Rightarrow_{[\bullet]}$ and $\Rightarrow^{[\bullet]}$ have the same reflexive transitive closure, i.e. the one defined above as $\Rightarrow^{[\bullet]*}$. So, we could have used the simpler relation $\Rightarrow_{[\bullet]}$ instead of the more complex $\Rightarrow^{[\bullet]}$ as the basis for the definition of $\Rightarrow^{[\bullet]*}$. Again, we chose the apparently more complex one to ease the translation into Coq.

Definition 4. *A* non deterministic finite tree automaton *(NTFA) $\mathcal{A}$ is a pair $(\mathcal{Q}_{\mathcal{A}}, \mathcal{T}_{\mathcal{A}})$, where $\mathcal{Q}_{\mathcal{A}} = \{@a, @b, \dots\}$ is a finite set of* states *and $\mathcal{T}_{\mathcal{A}}$ is the finite set of* transitions*. Transitions have the form $@q \to T$, where $T \in \mathcal{T}_{\mathsf{pl}}$. To simplify notation, we identify $\mathcal{T}_{\mathcal{A}}$ with $\mathcal{A}$, and simply write $(@q \to T) \in \mathcal{A}$ instead of $(@q \to T) \in \mathcal{T}_{\mathcal{A}}$. We sometimes write several transitions in the compact form $@q \to T_1, \dots, T_n$, meaning $@q \to T_1, \dots, @q \to T_n$.*

Note the transitions of a NTFA are either of the form $@a \to @b$ (ϵ-transition) or of the form $@a \to f(@b_1, \dots, @b_k)$. We allow ϵ-transitions in the definition above since they can be used in the input to Rewrite and allow for a nice formulation of data flow abstraction [17]. All automata we shall use here are NTFA, so we will often refer to them as "automata", simply.

Definition 5. *A set $\mathcal{I}$ of* intersection constraints *is a set of triples of states, i.e. $\mathcal{I} \subseteq \mathcal{Q} \times \mathcal{Q} \times \mathcal{Q}$. We write each triple in $\mathcal{I}$ as a constraint $@a \supseteq @b \cap @c$.*

Intersection constraints are useful to precisely approximate equalities in guards. For instance, consider `P1 ; if x=y then P2(x)`. We can compute approximations $@x$ and $@y$ for the values of variables `x` and `y` after `P1`. We could safely use $@x$ when approximating `P2(x)`, neglecting the guard `x=y`, and pretend that every value of $@x$ can reach `P2(x)`. A better approximation can be computed by refining $@x$ to a new state $@x'$ and adding an intersection constraint $@x' \subseteq @x \cap @y$. Intersection constraints were used for modelling a version of Kerberos [17].

Each state of an automaton $\mathcal{A}$ has an associated *language*, which intuitively is the set of terms reachable through transitions. When a set of intersection constraints $\mathcal{I}$ is also taken into account, the languages of the states of $\mathcal{A}$ are closed so that the constraints are satisfied. Similarly, when working with a rewriting system $\mathcal{R}$, languages are also closed under rewritings. We can now formalize this fact.

Definition 6. *Given $\mathcal{A}, \mathcal{R}, \mathcal{I}$ we define the languages of its states up to rewritings in $\mathcal{R}$ through the following inference rules. We write $@q \rightsquigarrow_{\mathcal{A},\mathcal{I},\mathcal{R}} T$ when term $T \in \mathcal{T}_{\mathsf{pr}}$ is in the language of $@q$ according to $\mathcal{A}$, $\mathcal{I}$, and taken up to rewritings in $\mathcal{R}$. When unambiguous, we shall omit the indexes and simply write $@q \rightsquigarrow T$.*

$$\mathsf{Direct}\frac{@b \rightsquigarrow T}{@a \rightsquigarrow T}(@a \to @b) \in \mathcal{A} \qquad \mathsf{Rew}\frac{@a \rightsquigarrow T_1}{@a \rightsquigarrow T_2} T_1 \Rightarrow_{\mathcal{R}}^{[\bullet]*} T_2$$

$$\mathsf{Cons}\frac{\forall i \in [1,n].\ @b_i \rightsquigarrow T_i}{@a \rightsquigarrow \mathsf{f}(T_1, \ldots, T_n)}(@a \to \mathsf{f}(@b_1, \ldots, @b_n)) \in \mathcal{A}$$

$$\mathsf{Inters}\frac{@b \rightsquigarrow T \qquad @c \rightsquigarrow T}{@a \rightsquigarrow T}(@a \supseteq @b \cap @c) \in \mathcal{I}$$

It is sometimes convenient to refer to the $\rightsquigarrow$ relation as defined using only a few selected inference rules from those above. In these cases we shall write them explicitly, e.g. $\rightsquigarrow^{\mathsf{Cons}}$

4 Background: Completion

Intuitively, the *completion* algorithm takes $\mathcal{A}, \mathcal{I}, \mathcal{R}$ and outputs another automaton $\mathcal{F}$, such that its languages include those of $\mathcal{A}$. Further, the main property of $\mathcal{F}$ is that its languages are already closed under $\mathcal{R}$ and $\mathcal{I}$, i.e. inference rules Rew and Inters are redundant for $\mathcal{F}$. This means that the triple $\mathcal{A}, \mathcal{I}, \mathcal{R}$ can be over-approximated with $\mathcal{F}$ alone, thus effectively discarding the $\mathcal{I}$ and $\mathcal{R}$ components. Therefore, one can infer facts about the language of $\mathcal{A}$ up to rewriting by inspecting the automaton $\mathcal{F}$, only. This is convenient because the inspection of an automaton is simple to perform, while reasoning about general term rewriting systems is difficult, often leading to undecidability – indeed, manually writing a Coq proof on $\mathcal{A}$ up-to-rewriting can be very hard.

To keep our presentation simple, we here assume that the automaton $\mathcal{A}$ has a single *goal* state, the language of which is the only one we care about. This intuitive statement will be correctly formalized below, when we shall introduce a

preorder for the automata. Note, however, that the techniques we present in this paper can easily be extended to handle multiple goal states, and that Rewrite indeed does not rely on this assumption in any way.

The purpose of completion can be understood using the inference rules of Def. 6. Given $\mathcal{A}, \mathcal{I}, \mathcal{R}$, and the goal state $@q$, we consider its language $W_{\mathcal{A}} = \{T | @q \rightsquigarrow_{\mathcal{A}} T\}$. The result of the completion is another automaton $\mathcal{F}$, the states of which include all the states occurring in $\mathcal{I}$, as well as the goal state $@q$. Let the new language of $@q$ be $W_{\mathcal{F}} = \{T | @q \rightsquigarrow_{\mathcal{F}} T\}$. Then, the automaton $\mathcal{F}$ satisfies

$$W_{\mathcal{F}} = \{T | @q \rightsquigarrow_{\mathcal{F}}^{\mathsf{Direct},\mathsf{Cons}} T\} \tag{1}$$

$$W_{\mathcal{A}} \subseteq W_{\mathcal{F}} \tag{2}$$

Property (1) states that terms in the language $W_{\mathcal{F}}$ can be derived without using rules Rew and Inters. In other words, we can neglect the rewritings in $\mathcal{R}$ and the intersection constraints in $\mathcal{I}$ when we study $W_{\mathcal{F}}$. This leads to a decision procedure for determining whether a pure term T belongs to $W_{\mathcal{F}}$, which we shall describe below.

Property (2) states that $W_{\mathcal{F}}$ is an over-approximation of $W_{\mathcal{A}}$. So, when the decision procedure results in T not belonging to $W_{\mathcal{F}}$, we can automatically infer $T \notin W_{\mathcal{A}}$ (the converse is not true, because the approximation $W_{\mathcal{F}}$ can be larger than $W_{\mathcal{A}}$). Hence, we can effectively use the decision procedure to prove facts about $W_{\mathcal{A}}$, and therefore about $\mathcal{A}, \mathcal{I}, \mathcal{R}$.

Deciding Membership We give now a very short description of the decision procedure: the actual details can be found in [16]. Basically, we try to build a derivation for $@q \rightsquigarrow_{\mathcal{F}} T$ using only rules Direct and Cons. We proceed bottom-up from the goal $@q \rightsquigarrow_{\mathcal{F}}^{\mathsf{Direct},\mathsf{Cons}} T$ and try the rules for all the transitions of $\mathcal{F}$ in a non-deterministic way (the non-determinism branching is bounded by $|\mathcal{T}_{\mathcal{F}}|$). In a finite time, we can then decide whether $@q \rightsquigarrow_{\mathcal{F}}^{\mathsf{Direct},\mathsf{Cons}} T$. While we use this naïve membership test in our proofs, more efficient algorithms exist [3].

Comparing Automata We now introduce a preorder to compare automata. Given the goal state $@q$, we write $\mathcal{A} \sqsubseteq \mathcal{A}'$ when $@q \rightsquigarrow_{\mathcal{A}} T$ implies $@q \rightsquigarrow_{\mathcal{A}'} T$. We write $\mathcal{A} \equiv \mathcal{A}'$ when both $\mathcal{A} \sqsubseteq \mathcal{A}'$ and $\mathcal{A}' \sqsubseteq \mathcal{A}$. Note that this involves all the rules of Def. 6, and not only Direct and Cons.

The completion algorithm used in Rewrite exploits the following automaton transformations, changing $\mathcal{A}$ into $\mathcal{A}'$ with $\mathcal{A} \sqsubseteq \mathcal{A}'$.

$$\mathsf{Augment} \quad \frac{}{\mathcal{A} \sqsubseteq \mathcal{A} \cup (@l \to T)} \quad T \in \mathcal{T}_{\mathsf{pl}}$$

$$\mathsf{Transitivity} \quad \frac{(@l_1 \to @l_2) \in \mathcal{A} \quad (@l_2 \to T) \in \mathcal{A}}{\mathcal{A} \equiv \mathcal{A} \cup (@l_1 \to T)}$$

$$\mathsf{Join} \quad \frac{\forall T \,.\, (@l_1 \to T) \in \mathcal{A} \iff (@l_2 \to T) \in \mathcal{A}}{\mathcal{A} \equiv \mathcal{A}\{@l_2/@l_1\}} \quad \begin{array}{l} @l_1 \neq @q \\ @l_1 \notin \mathsf{states}(\mathcal{I}) \end{array}$$

Rule Augment simply adds a transition to $\mathcal{A}$, clearly preserving our preorder. Rule Transitivity short-cuts two transitions, yielding a completely equivalent automaton. Rule Join is more peculiar. It replaces one state $@l_1$ with another state

Inputs: $\mathcal{A}, \mathcal{I}, \mathcal{R}$
Output: $\mathcal{F}$

1. $\mathcal{F} := \mathcal{A}$
2. **repeat** close $\mathcal{F}$ under Transitivity
3. close $\mathcal{F}$ under intersection constraints $\mathcal{I}$
4. apply rule Join in $\mathcal{F}$ as much as possible
5. **for each** critical pair w.r.t. $\mathcal{R}$
6. resolve the pair by applying Augment to $\mathcal{F}$
7. **until** $\mathcal{F}$ reaches a fixed point

Fig. 1. The Rewrite completion algorithm

$@l_2$, under the assumption they have the same transitions (and therefore recognize the same language). The languages recognized by the states are unaffected, except for $@l_1$ which disappears from the automaton. The side conditions of rule Join ensures that $@q$ as well as the states mentioned in $\mathcal{I}$ are not removed from the automaton, so that equivalence is indeed preserved.

The Completion Algorithm The exact way of applying the transformations above to form the completion algorithm used in Rewrite is detailed in [16]. Here, we only describe the portions of it that are relevant to the generation of the Coq proof. Notably, we omit how we ensure termination. The (simplified) main loop of Rewrite is shown in Fig. 1. We start from $\mathcal{F} = \mathcal{A}$, and apply the transformations until we reach a fixed point.

In step 2, we look for pair of transitions of the form $@a \to @b$ and $@b \to T$; when that happens we add $@a \to T$ to $\mathcal{F}$. In step 3, for each intersection constraint $@a \subseteq @b \cap @c$, we look for transitions of the form $@b \to T_b$ and $@c \to T_c$. Then we adapt the language of $@a$ to include (an over-approximation of) the terms reachable from both T_b and T_c. In step 4, we merge the states having the same transitions, applying the Join rule. This is not strictly necessary for the soundness of the algorithm, but it proved to be an useful optimization in our applications [17]. Of course, when generating the Coq proof, we shall cope with the merging of states.

In step 5, we deal with rewritings. We look for critical pairs, i.e. terms generable through the automaton that are also subject to some rewriting. For instance, if we find $@a \to \mathsf{fst}(@b)$ and $@b \to \mathsf{cons}(@c, @d)$, we have a critical pair together with the rewriting $\mathsf{fst}(\mathsf{cons}(X, Y)) \Rightarrow X$. When this happens, we resolve the pair in step 6, by applying Augment to add $@a \to @c$ to $\mathcal{F}$. In the general case, this might be more complex (see [16]), but we can always resolve pairs through Augment. The termination of the algorithm can be enforced as shown in [16]. Upon termination, it should be clear that we have $\mathcal{A} \sqsubseteq \mathcal{F}$, since we modified $\mathcal{F}$ through our preorder-preserving transformations, only. This implies the wanted relation between the languages of $@q$ we introduced before, i.e. $W_{\mathcal{A}} \subseteq W_{\mathcal{F}}$. Furthermore, we also have the other wanted property, $W_{\mathcal{F}} = \{T | @q \rightsquigarrow_{\mathcal{F}}^{\mathsf{Direct},\mathsf{Cons}} T\}$, since $\mathcal{F}$ is closed under Inters in step 3, and closed under Rew in step 5.

The automaton $\mathcal{F}$ actually satisfies the following *stronger* property.

$$W_{\mathcal{F}} = \{T | @q \rightsquigarrow_{\mathcal{F}}^{\mathsf{Cons}} T\} \tag{3}$$

So, each term in $W_{\mathcal{F}}$ can be built through Cons rules, only. This property holds because in step 2 we closed $\mathcal{F}$ under Direct. When generating the Coq proofs, this will turn out quite useful.

5 Certifying Unreachability in Coq

We now discuss how we generate the correctness proofs for $\mathcal{F}$. In Sect. 2, we already showed parts of the generated proof for an example. There, we presented the statement of an unreachability lemma and some inductive definitions, involving only $\mathcal{A}$ and $\mathcal{R}$ as provided by the user ($\mathcal{I}$ was empty in the example). Here, we complete the missing parts, discussing the role of $\mathcal{F}$ in the actual proof. Providing a detailed semantics for Coq is clearly outside the scope of this paper, so we shall often appeal to intuition in this section, and refer the reader to [4].

First, we need to provide Coq definitions for terms, the inputs $\mathcal{A}, \mathcal{I}, \mathcal{R}$, and the output $\mathcal{F}$. Terms are defined through Coq `Inductive` declarations.

```
Inductive term : Set := [...]
  | h_cons : term -> term -> term
```

The above also fixes the arity of term constructors (e.g. two for `h_cons`).

We now deal with $\mathcal{R}$, defining the relations introduced in Def. 3. Actually, we already formalized these in Coq in Sect. 2, so we briefly make the connections clear. The relation $\Rightarrow^{\bullet}$ (in Coq: `R`) poses no problem: instantiating all the variables with terms in a rewriting rule is simply done by universally quantifying these variables over all the terms. The relation $\Rightarrow^{[\bullet]}$ (in Coq: `RC`) is defined exactly following points 1,2,3 in Def. 3: point 3 is expanded for all term constructors `f`. Finally, the relation $\Rightarrow^{[\bullet]*}$ (in Coq: `RCs`) is a simple transitive closure.

In our Coq proofs, automata states are defined through their languages, i.e. as sets of terms. Rather than using a Coq library for set theory, we found the standard encoding of sets into the corresponding predicates to be enough for our purposes. So, below we define `St` to be the type representing the sets of terms.

```
Definition St := term -> Prop .
```

We can now define the automaton $\mathcal{A}$ by specifying the languages for all its states. This is done by closely following Def. 6, forming a single mutually recursive definition for the whole automaton. Rules Direct and Cons are expanded for all the transitions of $\mathcal{A}$. Similarly, rule Inters is expanded for all the intersection constraints in $\mathcal{I}$. Rule Rew instead is not expanded, but is stated using the relation `RCs` introduced before. We stress that this way of defining $\mathcal{A}$ makes it closed under all the rules of Def. 6.

In Sect. 2 we showed the result for the state `O_zac`: there, the line `T_O_90` derives from the Cons rule, while the line `T_O_91` derives from the Rew rule. No

Direct rule is applicable there, since there is no transition $@q \to @w$ in $\mathcal{A}$ for any $@w$. Furthermore, in the example $\mathcal{I}$ is empty, so Inters never applies.

We define the result of the completion, i.e. the automaton $\mathcal{F}$, in a slightly different way. Since a main objective of the proof is to show that rules Inters and Rew are unnecessary, we neglect them when defining the languages of $\mathcal{F}$. Except for this, we proceed as for $\mathcal{A}$, i.e. we expand rules Direct and Cons over all the transitions of $\mathcal{F}$.

Finally, we can state and prove the correctness results. We introduce an auxiliary relation `incl_T` to link the states of $\mathcal{A}$ to those of $\mathcal{F}$, intending that when `incl_T` holds between $@a$ and $@b$, then we expect the language of $@a$ to be included in that of $@b$. Basically, `incl_T` relates the states with the same names, except for the first belonging to $\mathcal{A}$ and the second to $\mathcal{F}$. Also, here we must take into account the Join operations performed during completion. If we merged two states in $\mathcal{F}$, making one of them disappear, then `incl_T` must point at the state that is still present in $\mathcal{F}$. In the general case, multiple merges are possible, so we need to follow the Join chain and to make `incl_T` use the state still present in the result of the completion $\mathcal{F}$. Of course, the goal state $@q$ is never removed from $\mathcal{F}$ because of the Join side condition, so `incl_T` just relates the $@q$ in $\mathcal{A}$ to the $@q$ in $\mathcal{F}$. The actual definition of `incl_T` is just a table.

The main correctness result is then the following one, which implies properties (1) and (2). Intuitively, it states that the languages of $\mathcal{A}$ are contained in the corresponding ones in $\mathcal{F}$.

```
Lemma Automaton_T_is_a_safe_overapproximation_of_automaton_0 :
  forall s_o s_t t, incl_T s_o s_t -> s_o t -> s_t t .
```

To prove the above lemma, we use a number of other results. Among those, we show that $\mathcal{F}$ is closed under rewritings in $\mathcal{R}$, as well as under intersection constraints in $\mathcal{I}$. Below, `State_T` and `intCon_T` are predicates, enumerating all the states of $\mathcal{F}$ and the constraints in $\mathcal{I}$, respectively.

```
Lemma Automaton_T_closed_under_RCs :
   forall s_t t u, State_T s_t -> s_t t -> RCs t u -> s_t u.
Lemma Automaton_T_satisfies_intersection_constraints :
   forall s1 s2 s3 t, State_T s1 -> intCon_T s1 s2 s3 ->
      s2 t -> s3 t -> s1 t .
```

Proving in Coq the above three lemmata turned out to be difficult. Of course, these proofs are inductive arguments. We tried several Coq proof tactics for that.

First, we considered the `induction` tactic to perform some kind of *structural induction*. Unfortunately, structural induction on term `t` is not enough to prove the lemmata above. This is because, in the general case, we are not able to reduce the inductive properties to a proper subterm of `t`. This reduction is possible for the Cons case, that removes a constructor from `t`. For instance, when the only transition from $@a$ is $@a \to \mathsf{f}(@b)$, and we must show `O_zaa (f t2)`, then we can reduce that to showing `O_zab t2`, and apply the induction hypothesis on the smaller term `t2`. However, this is not the case for the Direct case which simply changes the state $@a$ to another state, without affecting `t`.

Structural induction failing, we tried *rule induction* on the definition of the automaton languages, i.e. on the rules of Def. 6. For mutually recursive definitions, as those we use, this is best done in Coq with the `Minimality` induction principle scheme. This was still not enough. The problem lies in searching for critical pairs, e.g. in the lemma stating that $\mathcal{F}$ is closed under $\mathcal{R}$. Assume we have $\mathcal{R} = \{\mathsf{f}(\mathsf{g}(X)) \Rightarrow X\}$, and a transition $@a \to \mathsf{f}(@b)$. In order to show that $@a$ is closed under this rewriting rule, we need to search in the language of $@b$ for terms matching $\mathsf{g}(X)$. For this, the induction hypothesis offers no help, since it deals only with terms matching $\mathsf{f}(\mathsf{g}(X))$, saying nothing about $\mathsf{g}(X)$. Unfortunately, finding matches by a simple case analysis of the transitions of $@b$ is not feasible (i.e. using the Coq tactic `inversion`, which enumerates the immediate inductive subcases). This is because we might find a transition loop, e.g. formed by $@b \to @c$ and $@c \to @b$, and get stuck in the Direct case. We thought of several ways to escape from this impasse:

- We could change the induction predicate to consider partial matches of a left hand side of a rewriting rule. This seems to be hard.
- We could detect loops, and in that case start a new subproof by rule induction. We deemed this to be quite complex as well.
- We could reduce the problem to something else involving a simpler inductive definition, hopefully not causing the issues above.

Equation (3) indeed suggests that the third way is doable, since we can neglect Direct rules for $\mathcal{F}$ without affecting its languages. Hence, we define another automaton $\mathcal{N}$, identical to $\mathcal{F}$ except for the transitions of the form $@a \to @b$, which we drop. We prove in Coq that $\mathcal{N}$ and $\mathcal{F}$ are equivalent.

This equivalence proof is done by rule induction. The only non-trivial case is to show that, when $@a \rightsquigarrow_{\mathcal{F}} T$ is derived from $@b \rightsquigarrow_{\mathcal{F}} T$ by Direct and $(@a \to @b) \in \mathcal{F}$, then we also have $@a \rightsquigarrow_{\mathcal{N}} T$. Here, by induction hypothesis we get $@b \rightsquigarrow_{\mathcal{N}} T$. We now conclude showing that, in $\mathcal{N}$ the language of $@b$ is included in that of $@a$. This is done by case analysis (`inversion`) on the transitions of $\mathcal{N}$, since for each transition $@b \to U$ we can find a transition $@a \to U$. The latter was inserted by the completion algorithm when closing under the Transitivity transformation.

So, we prove the lemmata stated above on $\mathcal{N}$ first, and transfer then the results to $\mathcal{F}$, leveraging on the equivalence. Proving that $\mathcal{N}$ is closed under rewritings is now almost trivial: we do not even need full induction for that. With a finite number of Coq `inversion`s, we can find all the critical pairs, i.e. all the matches with the left hand sides of the rewriting rules. This case analysis always terminates, since each `inversion` removes one constructor from the matched term. For each critical pair we find, we show that the rewritten term is indeed included in $\mathcal{N}$ by pointing at the transitions we added to the automaton through Augment in step 6 of the completion algorithm. When the added transition was of the form $@a \to @b$, and so it is in $\mathcal{F}$ but not in $\mathcal{N}$ (recall that $\mathcal{N}$ does not contain ϵ-transitions), we redo a part of the $\mathcal{F}$–$\mathcal{N}$ automata equivalence proof to prove the language of $@b$ included in that of $@a$.

Showing that $\mathcal{N}$ is indeed closed by $\mathcal{I}$, i.e. by Inters, is done in the same way: by case analysis, and then by pointing to the transitions added during completion. This is our third lemma above.

Finally, we can show the main lemma, stating $\mathcal{A}$ is over-approximated by $\mathcal{N}$. This is done by rule induction on the rules in Def. 6 for the transitions of $\mathcal{A}$. The Rew and Inters cases are shown by the auxiliary lemmata. The Cons case follows from the induction hypothesis, since the transition of $\mathcal{A}$ used for Cons is still present in $\mathcal{N}$ (possibly using different states if Join was applied, but `incl_T` took that into account). For the Direct case, once again we redo the part of the $\mathcal{F}$–$\mathcal{N}$ automata equivalence proof to show the inclusion of languages. This completes the proof.

We conclude the proofs by showing the unreachability goals included in the input to Rewrite, as that shown in Sect. 2. This is done first by case analysis on $\mathcal{N}$, and second by applying the main lemma to transfer the result to $\mathcal{A}$. One might note that we do not need to pass through $\mathcal{F}$ to show this goal, since $\mathcal{N}$ suffices. Accordingly, Rewrite can be told to omit the proofs for $\mathcal{F}$ completely, to provide a more compact proof of the unreachability goal on $\mathcal{A}$, only.

The generation of the whole Coq proof, as described in this section, only requires (parameterized) polynomial time in the size of $\mathcal{F}$. This is not surprising, since the proof line is roughly related to running the last iteration of the completion algorithm to check $\mathcal{F}$ is indeed a fixed point. Since the completion is polynomial [14], so is generating the proof.

6 Conclusions

We briefly comment on the size of the generated proofs. Consider the example of Sect. 2. Running Rewrite on that yields a proof file 753 lines long, and includes the unreachability lemma of Sect. 2 and the correctness of the automata $\mathcal{N}$ and $\mathcal{F}$. If we only care about the unreachability lemma, the proof can be shortened to 473 lines. On our desktop machine, Coq checked the proof in about three seconds. The actual proof can be found at [11].

In our experiments, we also generated proofs for automatically generated tree automata. Rewrite can also extract tree automata from cryptographic protocol specifications in a process calculus [17], so we checked these automata as well. A proof for the automata extracted from a Diffie-Hellman-based protocol is about 4500 lines long, and was checked in 23 minutes. The generated proof for a version of Kerberos is about 9000 lines long, and was checked in 161 minutes. In all cases, computing the completion and generating the proof was much faster than checking the proof.

Further, we studied Rewrite running times to understand the relative weights of all the phases: completion, proof generation, proof checking (via Coq). We were pleased to note in our experiments that the proof generation time was practically negligible with respect to the time needed to compute the completion $\mathcal{F}$. We concluded that this was because of two distinct reasons:

– Our proof generation technique only needs the fixed point $\mathcal{F}$ to produce a proof. In particular, all the intermediate approximations for $\mathcal{F}$ computed by the completion algorithm are not used. First, this implies that there is no overhead during all the intermediate loops of the algorithm. Second, this also means that we do not need to certify anything about these intermediate results, so we can save both proof generation time and proof checking time.
– In some "hard" parts of the proof generation, we simply instruct Coq to try several tactics non-deterministically, rather than pointing at the actual one that succeeds. In other words, sometimes we know that the proof can be reached at the end of road A or road B; rather than computing which is the right one first, we let Rewrite tell Coq to try both. We do use branch pruning to avoid EXPTIME complexity. This means that much of the actual work is deferred from proof generation to proof checking. The time penalty for this is, roughly, only the one due to using the Coq tactic language instead of Haskell. While we could use Haskell to prune proof branches, we found the use of the Coq tactic language to be more convenient for this task, since in Coq we can easily query the current proof subgoals, and react accordingly.

Finally, we add a note about Coq tactics. What is generated by the technique presented above is more precisely called a Coq tactic, i.e. a program that generates the actual proof (which is roughly a λ-term). So, when we referred to "proof-checking" we actually referred to running this tactic, and checking the resulting proof. The generated tactic never fails and produces a correct proof. Note however that this fact does not rule out bugs in the implementation: generated tactics still have to be checked.

Concluding, in this paper we discussed how to certify, via a machine-checkable proof, that the result of the completion algorithm on tree automata is indeed correct, i.e. closed under rewriting. Our technique is general, and can be used wherever the completion algorithm is used. We studied its implementation in our tool Rewrite and experimented with certifying cryptographic protocol security.

References

1. AVISPA project home page. http://www.avispa-project.org.
2. Y. Boichut. Tree automata for security protocols (TA4SP) tool. http://lifc.univ-fcomte.fr/∼boichut/TA4SP/TA4SP.html.
3. H. Comon, M. Dauchet, R. Gilleron, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi. Tree automata techniques and applications. Available on: http://www.grappa.univ-lille3.fr/tata, 1997. release October, 1rst 2002.
4. The Coq proof assistant. http://coq.inria.fr.
5. J. Courant and J. Monin. Defending the bank with a proof assistant. In *Proceedings of WITS 2006*, pages 87–98, 2006.
6. G. Feuillade, T. Genet, and V. V. T. Tong. Reachability analysis over term rewriting systems. *Journal of Automated Reasoning*, 33:341–383, 2004.
7. T. Genet, Y. T. Tang-Talpin, and V. V. T. Tong. Verification of copy-protection cryptographic protocol using approximations of term rewriting systems. In *Proc. of Workshop on Issues in the Theory of Security*, 2003.

8. J. Goubault-Larrecq. Une fois qu'on n'a pas trouvé de preuve, comment le faire comprendre à un assistant de preuve? In *Actes 15èmes journées francophones sur les langages applicatifs (JFLA)*, pages 1 – 40, 2004.
9. H1: an automated deduction system (Web site). http://www.lsv.ens-cachan.fr/∼goubault/H1.dist.
10. G. C. Necula. Proof-carrying code. In *POPL*, pages 106–119, 1997.
11. The Rewrite protocol analysis tool. http://www.di.unipi.it/∼zunino/software.
12. T. Takai. A verification technique using term rewriting systems and abstract interpretation. In *Proc. of Rewriting Techniques and Applications*, volume 3091 of *Lecture Notes in Computer Science*, pages 119 – 133, 2004.
13. Timbuk tree automata tool. http://www.irisa.fr/lande/genet/timbuk.
14. R. Zunino. *Models for Cryptographic Protocol Analysis*. PhD thesis, Italy, University of Pisa, 2006.
15. R. Zunino. Defending the bank with a static analysis. In *Proc. of NordSec*, 2006, http://www.di.unipi.it/∼zunino/papers.
16. R. Zunino and P. Degano. Finite approximations of terms up to rewriting. http://www.di.unipi.it/∼zunino/papers/completion.html.
17. R. Zunino and P. Degano. Handling exp, × (and timestamps) in protocol analysis. In *Proceedings of FoSSaCS 2006*, volume 3921 of *LNCS*, pages 413–427, 2006.