

**Timbuk**

A Tree  
Automata  
Library

**Version 2.0**

The **Timbuk** tree automata library now contains three standalone tools:

Timbuk: Copyright © 2000-2003 Thomas Genet and Valérie Viet Triem Tong

Taml: Copyright © 2003 Thomas Genet

Tabi: Copyright © 2003 Thomas Genet and [Boinet Matthieu, Brouard Robert, Cudennec Loic, Durieux David, Gandia Sebastien, Gillet David, Halna Frederic, Le Gall Gilles, Le Nay Judicael, Le Roux Luka, Mallah Mohamad-Tarek, Marchais Sebastien, Martin Morgane, Minier François, Stute Mathieu] – aavisu project team for french "maitrise" level (4th University year) 2002-2003 at IFSIC/Université de Rennes 1.

# Contents

<b>1</b>	<b>Timbuk library overview</b>	<b>4</b>
1.1	What is Timbuk? . . . . .	4
1.2	Availability, License and Installation . . . . .	4
1.3	Note on the implementation . . . . .	5
1.4	Bug report and information . . . . .	5
1.5	Changes from version 1.1 to version 2.0 . . . . .	5
<b>2</b>	<b>Tutorial</b>	<b>7</b>
2.1	Timbuk . . . . .	7
2.1.1	Exact case . . . . .	10
2.1.2	Interactive approximations and priority transitions . . . . .	12
2.1.3	Normalization rules . . . . .	15
2.1.4	Bigger example: cryptographic protocol . . . . .	16
2.1.5	Verifying left-linearity condition . . . . .	23
2.1.6	Doing more and going faster . . . . .	24
2.1.7	More tricks . . . . .	25
2.2	Taml . . . . .	27
2.3	Tabi . . . . .	32
2.3.1	Basic . . . . .	32
2.3.2	Display modes . . . . .	33
2.3.3	Using Tabi to approximate in Timbuk . . . . .	34
<b>3</b>	<b>Specification language reference manual</b>	<b>35</b>
3.1	Comments . . . . .	36
3.2	Symbols . . . . .	36
3.3	Alphabets . . . . .	36
3.4	Variable sets . . . . .	36
3.5	Term Rewriting Systems . . . . .	36
3.6	Tree Automata . . . . .	36
3.6.1	Implicit definitions . . . . .	36
3.6.2	Explicit definitions . . . . .	36
3.7	Approximations . . . . .	37
<b>4</b>	<b>Timbuk reference manual</b>	<b>38</b>
4.1	Running Timbuk . . . . .	38
4.2	Timbuk normalization and approximation tools . . . . .	39
4.2.1	Priority transitions . . . . .	39
4.2.2	Normalization rules . . . . .	40
4.2.3	Merging rules . . . . .	40
4.2.4	Approximation equations . . . . .	41
4.3	Timbuk commands . . . . .	41
4.4	Timbuk modes and command line options . . . . .	43
4.4.1	Dynamic completion mode . . . . .	43

4.4.2	Static completion mode . . . . .	45
<b>5</b>	<b>Taml reference manual</b>	<b>45</b>
5.1	Running Taml . . . . .	45
5.2	Basic Taml functions . . . . .	46
5.3	Using all Timbuk library functions through Taml . . . . .	49
<b>6</b>	<b>Tabi reference manual</b>	<b>49</b>
6.1	Mouse actions . . . . .	50
6.2	Buttons . . . . .	50
6.3	File menu . . . . .	51
6.4	Options menu . . . . .	51
<b>7</b>	<b>How to use Ocaml functions of the Timbuk library?</b>	<b>51</b>

# 1 Timbuk library overview

## 1.1 What is Timbuk?

Timbuk is a collection of tools for achieving proofs of reachability over Term Rewriting Systems (TRS for short) and for manipulating Tree Automata. The tree automata we use here are bottom-up non-deterministic finite tree automata (NFTA for short).

The Timbuk library provides three standalone tools and a bunch of Objective Caml [12] functions for basic manipulation on Tree Automata, alphabets, terms, Term Rewriting Systems, etc. The three tools are:

- Timbuk: a tree automata completion engine for reachability analysis over Term Rewriting Systems
- TamL: an Ocaml toplevel with basic functions on NFTA:
  - boolean operations: intersection, union, inversion, etc...
  - emptiness decision
  - cleaning, renaming
  - determinisation
  - matching over Tree Automata
  - parsing, pretty printing
  - normalization of transitions
  - and some more...
- Tabi: a Tree Automata Browsing Interface for visual automata inspection

See [1] for a survey on Term Rewriting Systems and [3] for a survey on tree automata. For reachability analysis and tree automata completion details, look at [7].

## 1.2 Availability, License and Installation

Timbuk is freely available, under the terms of the GNU LIBRARY GENERAL PUBLIC LICENSE, here:

<http://www.irisa.fr/lande/genet/timbuk/>

Now, Timbuk is available for download in Ocaml source and windows binary files. To install and run Timbuk, please refer to the README file of the distribution. For documentation on the Timbuk, TamL and Tabi tools, see tutorial in section 2 and respective reference manuals in sections 4, 5 and 6. For documentation on the interface of Ocaml NFTA functions, see section 7.

### 1.3 Note on the implementation

Most of the functions of the library are implemented straightforwardly without objects and in a functional style. However, we tried to develop this library in a modular way such that optimizations could be added afterward. Thus optimizations are welcome and can even be proposed to us for implementation. Tabi as been developed with Labltk (Ocaml with Tk functions) in collaboration with a group of students in 4th year of Computer Science of Rennes University (see README file for credits).

### 1.4 Bug report and information

Please report comments and bugs to [Thomas.Genet@irisa.fr](mailto:Thomas.Genet@irisa.fr).

See <http://www.irisa.fr/lande/genet/timbuk/> for information about Timbuk and related papers.

### 1.5 Changes from version 1.1 to version 2.0

Between Timbuk 1.1 and Timbuk 2.0, the following changes have been done:

- Added 'Taml': a standalone Ocaml toplevel with preloaded Timbuk library functions over tree automata, terms, term rewriting systems, etc. This replaces the old "CALCULATOR MODE" Timbuk '.mml' files.
- Added 'Tabi': the Tree Automata Browsing Interface for automata inspection. Tabi is a standalone application but can also be called from Timbuk and Taml.
- Made a more robust Makefile
- Made a poor man's windows Makefile
- Added `--dynamic`, `--static`, `--fstatic`, `-f`, `-o`, `--strat` options to Timbuk, with the following usage:

Options: <code>--dynamic</code>	for usual completion algorithm (default)
<code>--static</code>	to activate the static compilation of matching and normalisation (needs a complete set of prior and norm rules)
<code>--fstatic</code>	to activate the static compilation of matching and normalisation. If the set of prior and norm rules is not complete, a transition not covered by the rules is normalised using a single new state <code>#qstatic#</code>
<code>-f file</code>	to read Timbuk commands from a file
<code>-o file</code>	output Timbuk execution trace in a file
<code>--strat</code> followed by keywords:	
<code>exact</code>	(exact normalisation with prioritary rules)
<code>prior</code>	(normalisation with prioritary rules)
<code>norm_rules</code>	(normalisation with approximation rules)

<code>auto</code>	(automatic normalisation with new states)
<code>auto_conf</code>	(similar to 'auto' but asks for confirmation first)
<code>auto_prior</code>	(automatic normalisation with new states where new transitions are stored as prioritary rules)
<code>auto_prior_conf</code>	(similar to 'auto_prior' but asks for confirmation first)
<code>manual_norm</code>	(manual addition of normalisation rules)
<code>manual_norm_conf</code>	(similar to 'manual_norm' but asks for confirmation first)
<code>manual</code>	(manual normalisation)
<code>manual_conf</code>	(similar to 'manual' but asks for confirmation first)

- Added the ability to define approximation equations (in specifications and at run time)
- Added the 'exact' strategy which is always terminating and exact on some specific syntactic classes (see documentation)
- Changed the (w) command so that it writes to disk the TRS, the current completed automaton, the initial automaton, the current approximation and the list of automaton used for intersection in a same file
- Added the (p) Timbuk command for searching a pattern containing symbols, variables and states, in the completed automaton.
- Added the (o) Timbuk command for computing intersection between the completed automaton and some external automaton read in a file.
- Added the (m) Timbuk command for merging some states in the completed automaton.
- Added the (b) Timbuk command for browsing the current completed automaton with Tabi. Tabi also gives the ability to construct some merging rules graphically.
- Added the (f) Timbuk command to forget some completion steps.
- Added the (d) Timbuk command for displaying the current TRS used for completion.
- Added the (e) Timbuk command for consulting/adding the current approximation equations.
- Added the (g) Timbuk command for consulting/adding normalization rules interactively
- Added the (det) Timbuk command for determinizing the current completed automaton.
- Added the completion step number to Timbuk.
- Extended the `norm_rules` syntax to use normalization rules where it is possible to achieve matching on the state labels. For instance it is now possible to define such a normalization rule:

```
[encr(pubkey(q(x)), m) -> qstore] -> [m -> q(secret(x))]
```

in *dynamic mode* only, where `q` is here a state operator of arity 1 (defined using `States q:1` syntax).

- Added the 'Import' keyword in the specification language to import tree automata state operators in the approximation.
- Added the 'Set' keyword in the specification language to define automata through their finite language, i.e. finite set of terms which are compiled into a deterministic tree automaton.
- Optimised standard completion mode (dynamic) in Timbuk.
- Discarded some bugs.

## 2 Tutorial

In this tutorial, we assume that the reader is familiar with term rewriting systems [1], tree automata [3] as well as the notations and definitions of [7]. However, in the first part of this tutorial, basic notions on term rewriting systems should be enough.

The Timbuk library was initially designed to achieve reachability over Term Rewriting Systems, i.e. given a TRS  $\mathcal{R}$  and two terms  $s$  and  $t$  we try to prove or disprove that  $s \rightarrow_{\mathcal{R}}^* t$ . In Timbuk, we consider a more general problem which is reachability over *sets* of terms rather than on couple of terms, i.e. are terms of a set  $F$   $\mathcal{R}$ -reachable from terms of an initial set  $E$ . In that case, we can define the set of  $\mathcal{R}$ -descendants of a set of terms  $E$  denoted by  $\mathcal{R}^*(E) = \{t \mid s \in E \text{ and } s \rightarrow_{\mathcal{R}}^* t\}$ . Then given a second set of terms  $F$ , it is possible to prove for instance that all terms  $\mathcal{R}$ -reachable from  $E$  are in  $F$  ( $\mathcal{R}^*(E) \subseteq F$ ) or that none of the terms of  $F$  can be  $\mathcal{R}$ -reached from  $E$  ( $\mathcal{R}^*(E) \cap F = \emptyset$ ). This last property has some applications in verification [10, 7] where TRS are used to model programs,  $E$  all their possible initial configurations and  $F$  a set of *dangerous* configurations not to be reached when executing the program from the initial configurations.

When the sets  $E$  and  $F$  are infinite sets of terms it is necessary use a specific representation in order to model them finitely. This is essentially the role of the tree automata. Then computing exactly or approximately  $\mathcal{R}^*(E)$  for an infinite set of terms represented by a tree automaton can be done using a tree automaton completion algorithm. This algorithm is precisely the core of the Timbuk tool we now present. The aim of this tool is to produce a tree automaton recognizing  $\mathcal{R}^*(E)$  exactly when it is possible and approximately otherwise, and then to check if  $\mathcal{R}^*(E) \cap F = \emptyset$  for verification purposes.

### 2.1 Timbuk

The Timbuk tool performs tree automata completion with regards to a term rewriting system to compute exactly or approximately  $\mathcal{R}^*(E)$ . Tree automata and term rewriting systems are stored in a *Timbuk specification file* (see section 3 for details about the syntax). Let us begin by a simple example where there is no need to cope with tree automata syntax. Start Timbuk on the `basic.txt` file containing a first example, i.e. simply type:

```
timbuk basic.txt
```



in a command window. Depending on the way you obtained Timbuk, you may not be able to directly use 'timbuk' as a standalone command and you may need to type `ocamlrun timbuk basic.txt` instead. Please refer to the `README` file of the distribution for details on how to run the Timbuk library tools.

If launching Timbuk succeeds, then Timbuk reads the following `basic.txt` specification file:

```
(* This is a specification file to be used with the Timbuk tutorial *)

Ops f:1 g:1 a:0

Vars x y z

TRS R
  f(x) -> g(f(x))

Set init
  f(a)

Set check1
  f(g(f(a)))
  g(f(g(a)))

Set check2
  g(g(g(g(g(g(f(a))))))))
```

and starts completion on the TRS `R` and the set of terms (here a set containing a single term) denoted by `init`. When given a finite set of terms using the `Set` constructor, Timbuk transforms it into a tree automaton recognizing exactly this set, i.e. the set  $\{f(a)\}$  in our case. The other sets (and thus other tree automata) associated with names `check1` and `check2` will be used later for verification purpose. When launching Timbuk on this specification, you should obtain the following output:

```
Completion step: 0
Do you want to:
(c)omplete one step (use Ctrl-C to interrupt if necessary)
complete (a)ll steps (use Ctrl-C to interrupt if necessary)
(m)erge some states
(s)ee current automaton
(b)rowse current automaton with Tabi
(d)isplay the term rewriting system
(i)ntersection with verif automata
intersection with (o)ther verif automata on disk
search for a (p)attern in the automaton
(v)erify linearity condition on current automaton
(w)rite current automaton, TRS and approximation to disk
(f)orget old completion steps
(e)quation approximation in gamma
(g)amma normalisation rules
(det)erminise current automaton
(u)ndo last step
```

(q)uit completion

(c/a/m/s/b/d/i/o/p/v/w/f/e/g/det/u/q)?

meaning that the current completion step number is 0 and that Timbuk expect you to give one command. For instance, type `s` to display the current tree automaton being completed. You should obtain the following output:

States qterm0:0 qterm1:0

Final States qterm0

Transitions

a -> qterm1

f(qterm1) -> qterm0

which is the tree automaton recognizing the set of terms  $\{f(a)\}$ . Now it is possible to search for reachable terms from  $\{f(a)\}$  by doing some completion steps. Type `c` to achieve one completion step. Timbuk finds a new reachable term which corresponds to a new tree automata transition to add to the current tree automaton:

Adding transition:

g(f(qterm1)) -> qterm0

Adding this transition to the tree automaton will permit to recognize the term  $g(f(a))$  which is reachable from  $f(a)$  when applying rule  $f(x) \rightarrow g(f(x))$ . However the transition `g(f(qterm1)) -> qterm0` has to be normalized first, i.e. be transformed into an equivalent set of normalized transitions. Normalized transitions are of the form  $f(q_1, \dots, q_n) \rightarrow q$  where  $q_1, \dots, q_n$  are states. In our case, the subterm `f(qterm1)` is not a state. Timbuk asks if you want to give specific normalization rules by hand to normalize this transition. Answer no `n` and use automatic normalization with new states instead, by answering `y` to the second question. This causes Timbuk to create a new state `qnew0` to normalize automatically the transition into a set of two normalized transitions equivalent<sup>1</sup> to `g(f(qterm1)) -> qterm0`:

Adding transition:

g(qnew0) -> qterm0

... already normalised!

Adding transition:

f(qterm1) -> qnew0

... already normalised!

---

<sup>1</sup>Note that with these two new transitions it is possible to rewrite term `g(f(qterm1))` into `g(qnew0)` and then rewrite `g(qnew0)` into `qterm0`. Hence adding those two transitions permits to rewrite `g(f(qterm1))` into `qterm0` which corresponds to the transition we initially wanted to add.

This ends the first completion step. Using the same normalization methodology (i.e. always normalize with new states) it is possible to complete step 2, step 3 and so on, but completion does not terminate with this strategy. This is not really surprising since rule  $f(x) \rightarrow g(f(x))$  is not terminating on term  $f(a)$  and we are incrementally adding an infinite set of descendants of  $f(a)$ . However, since this example belongs to a specific decidable class<sup>2</sup>, we know that  $R^*(\{f(a)\})$  can be exactly computed using a tree automaton (it is regular). In the next section, we achieve the completion automatically on the same example using the **exact** strategy dedicated to the specifications of the decidable class.

### 2.1.1 Exact case

First, quit Timbuk if it is still running by typing `q` and launch it again with the exact strategy by typing

```
timbuk --strat exact basic.txt
```

Then either type repeatedly `c` or type once `a` for achieving completion until Timbuk succeeds at step 3:

```
Automaton is complete!!
```

```
-----
```

You can see the final completed automaton by typing `s`, and write this result into a file by typing `w`. Then it is possible to check if terms of the sets **check1** and **check2** are R-reachable from **f(a)**. This can be done by computing an intersection between the completed automaton recognizing the set of all R-reachable terms from **f(a)** ( $R^*(\text{init}) = R^*(\{f(a)\})$ ) with **check1** and **check2**. Intersections with finite sets or other automata contained in the same specification file can be done by typing `i`, this results in:

```
Intersection with check1 gives (the empty automaton):
```

```
States
```

```
Final States
```

```
Transitions
```

for **check1**, meaning that terms of **check1** are not reachable and for **check2** this results in:

```
Intersection with check2 gives (not empty):
```

```
States q9:0 q8:0 q7:0 q6:0 q5:0 q4:0 q3:0 q2:0 q1:0 q0:0
```

```
Final States q9
```

```
Transitions
```

```
a -> q0
```

---

<sup>2</sup>See exact strategy in section 4.4.1 for details on the decidable classes.

```

f(q0) -> q1
g(q1) -> q2
g(q2) -> q3
g(q3) -> q4
g(q4) -> q5
g(q5) -> q6
g(q6) -> q7
g(q7) -> q8
g(q8) -> q9

```

meaning that the term of `check2` is reachable from `f(a)` by rewriting with `R`. Now, quit `Timbuk` and try a new sample file `example.txt`

```
timbuk --strat exact example.txt
```

where `R` describes the classical *append* function on lists (the `app` symbol in the specification file) and `A0` recognizes an infinite set of terms of the form `app(t1, t2)` where `t1` is any flat list of `a` and `t2` is any flat list of `b`. Automaton `Problem1` recognizes only the two terms given in the definition, i.e. terms `cons(b, cons(a, nil))` and `cons(b, cons(b, cons(a, cons(a, nil))))`. Finally, the automaton `Problem2` recognizes the language of lists where there is at least one `b` followed by an `a`. This example is also in one of the decidable classes and can be automatically completed using `a` (or `c`) after 3 completion steps. Like in the previous example, we can verify that the intersection between the completed automaton and `Problem1` is empty meaning that the two recognized terms are not `R`-reachable from terms recognized by `A0`. However, the language corresponding to `Problem1` is finite and is a particular case. Thus, to really prove in the general case that the `append` function applied to *any* list of `a` and any list of `b` cannot result in any list where there is at least one `b` before an `a` it is necessary to compute the intersection between the completed automaton and `Problem2`, which is hopefully empty and thus guarantees the property.

Conjointly to intersections with additional tree automata, `Timbuk` provide another tool for proving or disproving reachability: pattern matching over the completed tree automaton. To do pattern matching, type `p`. `Timbuk` first recalls the symbols on which the pattern can be built: the alphabet, the set of states operators and the set of variables. On our example this results in the following output:

```

Alphabet=
cons:2 a:0 b:0 nil:0 app:2 rev:1

States=
qnew0:0 qa:0 qb:0 qla:0 qlb:0 qf:0

Variables=
x y z

```

Then `Timbuk` asks for a given pattern. For instance by typing `nil` for the pattern to be searched, we obtain:

```
Type a term and hit Return: nil
```

Solutions:

Occurrence in state qla!

solution 1: Empty substitution

Occurrence in state qlb!

solution 1: Empty substitution

Occurrence in state qf!

solution 1: Empty substitution

Occurrence in state qnew0!

solution 1: Empty substitution

which means that the term `nil` is recognized by four different states in the completed automaton, namely `qla`, `qlb`, `qf` and `qnew0`. Note that pattern matching is achieved on every terms recognized by the automaton as well as on all their subterms, this is why we here have several occurrences of this pattern. Now let us look for the following pattern:

```
cons(x, qla)
```

which produces the following list of solutions:

Solutions:

Occurrence in state qla!

solution 1: `x <- qa`

where this solution means that `cons(qa, qla)` is uniquely recognized by `qla`, and there is no other state `q` such that `cons(q, qla)` is recognized by the automaton. Now, if we get back to our verification problem, we can check that with append on lists of `a` and lists of `b`, no `b` can occur before an `a` by looking for this simple pattern: `cons(b, cons(a, y))` which results in the following output:

Pattern not found!

### 2.1.2 Interactive approximations and priority transitions

When the specification used is outside of decidable (regular) classes, completion with the exact strategy generally does not terminate. It is however possible to build an under-approximation of the reachable terms by computing  $n$  steps of completion for a given natural  $n$ . On the other hand, Timbuk permits to build an over-approximations of the set of reachable terms. In the next specification example `example2.txt`, we compute an approximation of the reverse function (symbol `rev` defined by TRS `R`) on the regular language of terms recognized by automaton `A0` i.e., `rev` applied to any flat lists of `a` and `b` where all `a`'s are before `b`'s in the list. The second automaton called `Problem1` recognizes a regular language of terms that should be unreachable from `A0` by rewriting with `R`: flat lists where there is at least one `'a'` before a `'b'` in the list. Here is the complete specification file `example2.txt`:

```

(* This is a specification file to be used with the Timbuk tutorial *)

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

Vars x y z

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

Automaton AO
States qrev qlab qlb qa qb
Description  qrev: "rev applied to lists where a are before b"
             qlab: "lists where a are before b (possibly empty)"
             qlb : "lists of b (poss. empty)"
Final States qrev
Transitions
  rev(qlab) -> qrev      nil -> qlab      cons(qa, qlab) -> qlab
  cons(qa, qlb) -> qlab  nil -> qlb      cons(qb, qlb) -> qlb
  a -> qa              b -> qb

Automaton Problem1
States qa qb qany qlb qlab qnil
Description
  qany: "Any flat list made of a and b"
  qlb : "Any flat list made of a and b, beginning with a b"
  qlab: "Any flat list with at least an a followed by a b"
Final States qlab
Transitions
  a -> qa
  b -> qb
  cons(qa, qany) -> qany
  cons(qb, qany) -> qany
  nil -> qany
  cons(qb, qany) -> qlb
  cons(qa, qlb) -> qlab
  cons(qb, qlab) -> qlab
  cons(qa, qlab) -> qlab

```

Let us achieve an interactive manual completion on this example (we will see how to automate this process in the following): type the command `timbuk --strat prior manual example2.txt` to use Timbuk with a normalization strategy using priority transitions first and then manual introduction of priority transition at a second time. The first completion step gives some new transitions and the following output:

Adding transition:

```
app(rev(qlb),cons(qa,nil)) -> qrev
```

Use key word 'States' followed by the names of the new states ended by a dot '.' (optional) then give a sequence of transitions ended by a dot '.'

Add a star '\*' before transitions you want to add to the prior set. The prior transitions should be normalized!!

We are proposing a transition which has to be normalized. First, we have to find states to recognize subterms `rev(qlb)` and `nil`. Since `qlb` recognizes lists of `b`, `rev(qlb)` represents the reverse function applied to lists of `b` and this should be a list of `b`. Thus we can recognize `rev(qlb)` by `qlb`. We define a new state `qnil` to normalize `nil`, and give the priority transitions to apply using the following syntax:

```

States qnil.
* rev(qlb) -> qlb
* nil -> qnil.

```

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

```

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

```

```

Adding transition:
app(qlb,cons(qa,qnil)) -> qrev

```

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

```

States qla.
* cons(qa, qnil) -> qla.

```

and this terminates the normalization of the first transition. There remains a transition to normalize:

```

Adding transition:
app(rev(qlab),cons(qa,qnil)) -> qrev

```

Since the state `qlab` recognizes a list of `a`'s followed by some `b`'s, we intend `rev(qlab)` to be a list of `b`'s followed by some `a`'s, so let us normalize it by a new state called `qlba` and introduce the corresponding priority transition.

```

States qlba.
* rev(qlab) -> qlba.

```

Then, some other transitions are automatically normalized and added, and this terminates the first completion step. In the following completion steps no other new states are necessary and it is enough to successively introduce the following priority transitions to normalize the new transitions we are proposed and thus terminate the completion:

```

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

```

Finally, from the menu it is possible to see the completed automaton which now contains 37 transitions and to compute the intersection with the automaton `Problems`, which gives an empty automaton meaning that applying `rev` to a list of `a`'s followed by some `b`'s cannot result into any list where there is an `'a'` before a `'b'`.

Now, save the produced completed automaton in a file named `comp.txt` by typing `w` and then the file name `comp.txt`. Now you can edit this file and check that the whole specification (TRS, completed automaton, initial automaton, additional automata used for verification as well as the constructed approximation) are stored in this file in Timbuk syntax. Note that since the approximation has been entirely built with priority rules and priority rules are usually stored in the `completed_A0` automaton, the approximation stored in the file is empty.

### 2.1.3 Normalization rules

Normalization rules (or norm rules) are rules of the form:

$$[s \rightarrow x] \rightarrow [l_1 \rightarrow r_1 \dots l_n \rightarrow r_n]$$

where  $s, l_1, \dots, l_n$  are terms that may contain symbols, variables and states, and  $x, r_1, \dots, r_n$  are either states or variables such that if  $r_i$  is a variable then it is equal to  $x$ . To normalize a transition of the form  $t \rightarrow q'$ , we match the pattern  $s$  on  $t$  and  $x$  on  $q'$ , obtain a given substitution  $\sigma$  and then we normalize  $t$  with the rewrite system  $\{l_1\sigma \rightarrow r_1\sigma, \dots, l_n\sigma \rightarrow r_n\sigma\}$  where  $r_1\sigma, \dots, r_n\sigma$  are necessarily states (see section 4.2.2 for details about norm rules).

Let us come back to the previous example and achieve completion with normalization rules. Start again Timbuk on the `example2.txt` file with the default Timbuk normalization strategy:

```
timbuk example2.txt
```

The default normalization strategy corresponds to the strategy operator sequence: `prior norm_rules manual_norm_conf auto_conf`, meaning that any transition is first normalized using priority transitions, then using normalization rules and if it is still not normalized, the user is asked for normalization rules, finally he can leave the automatic normalization finish the normalization if necessary. Doing a first step of completion, we are proposed a first transition to normalize and since there is still no priority transitions nor normalization rules, the strategy now consider the `manual_norm` operator:

Adding transition:

```
app(rev(qlb),cons(qa,nil)) -> qrev
```

Do you want to give by hand some NORMALIZATION rules? (y/n)?

Answer y to this question. First, Timbuk recalls the current normalization rules (here no one is already defined), alphabet, variables and state operators on which new rules can be built:

Do you want to give by hand some NORMALIZATION rules? (y/n)? y  
Current normalisation rules are:

```
Alphabet=cons:2 a:0 b:0 nil:0 app:2 rev:1
and Variables= x y z
and States= qrev:0 qlab:0 qlb:0 qa:0 qb:0
```

Type additional normalization rules using the 'States' and 'Rules' keyword and end by a dot '.':

(use keyword 'Top' to place a rule at the beginning of the rule list)

For this example, let us use a naive approximation strategy: for every term of the form `app(t1, t2)` let us normalize the two parameters of `app` by two distinct states, i.e. normalize term `t1` by a common state `qapp1` and `t2` by `qapp2` for every possible terms `t1` and `t2`. This can be done by typing interactively the following text:



```

States qapp1 qapp2
Rules
[app(x, y) -> z] -> [x -> qapp1 y -> qapp2].

```

where **States** (optional) is used to define a sequence of new states (if necessary) and **Rules** (mandatory) defines a sequence of norm rules ended by a dot symbol. Completion continues and proposes a new transition to normalize: `cons(qa,nil) -> qapp2`. Let us give some new normalization rules using the same naive strategy: we define two dedicated states `qcons1` and `qcons2` recognizing respectively the first and second subterm of every term of the form `cons(t1, t2)`.

```

States qcons1 qcons2
Rules
[cons(x, y) -> z] -> [x -> qcons1 y -> qcons2].

```

This is enough to terminate this completion step. Remaining steps are automatic and does not need any new approximation rule construction. Finally, we obtain a tree automaton with only 24 transitions but that does not fulfill the property we wanted to prove with automaton **Problem1** (type `i` to check that intersection is not empty) because approximation has been too drastic. However, some weaker properties can be verified on this automaton, for instance that the term `cons(a, rev(cons(a, nil)))` is not reachable from `A0` (by pattern matching). With regards to the property we wanted to prove initially with automaton **Problem1**, the approximation we gave in section 2.1.2 is one of the simplest we could build. All we can do with normalization rules here is to give the set of priority rules of section 2.1.2 as a normalization rule:

```

States qnil qla qlba
Rules [x -> y] -> [ rev(qlb) -> qlb
                    nil -> qnil
                    rev(qlab) -> qlba
                    app(qlb, qla) -> qlba
                    cons(qb, qnil) -> qlb
                    app(qnil, qlb) -> qlb
                    app(qnil, qla) -> qla
                    rev(qnil) -> qnil
                    app(qla, qla) -> qla]

```

where the pattern `[x -> y]` of the left-hand side of the normalization rule matches every transition, hence the right hand side will be applied on every transitions (like priority transitions). In the next section, we give an example where normalization rules shows their efficiency when approximation has to be precise on some parts and can be more drastic on the remaining ones.

#### 2.1.4 Bigger example: cryptographic protocol

Now let us introduce a bigger example coming from the cryptographic protocol verification domain. This example is the corrected version of the Needham-Schroder Public Key (NSPK for short) cryptographic protocol [13]. The NSPK protocol aims at mutual authentication of two agents, an initiator *A* and a responder *B*, separated by an insecure network. Mutual authentication means that, when a protocol session is completed between two agents, they should be assured of each other's identity. This protocol is based on an exchange of *nonces* (usually fresh random numbers

or time stamps) and on *asymmetric* encryption of messages: every agent has a *public key* (for encryption) and a *private key* (for decryption). Every public key is supposed to be known by any agent whereas, the private key of agent  $X$  is supposed to be only known by  $X$ . Thus, in this setting, we suppose that messages encrypted with the public key of  $X$  can only be decrypted and read by  $X$ . This is in fact a common hypothesis of the Dolev-Yao model [6]. Here is a description of the three steps of the fixed version of protocol, borrowed from [13]:

1.  $A \hookrightarrow B : \{N_A, A\}_{K_B}$
2.  $B \hookrightarrow A : \{N_A, N_B, B\}_{K_A}$
3.  $A \hookrightarrow B : \{N_B\}_{K_B}$

In the first step,  $A$  tries to initiate a communication with  $B$ :  $A$  creates a nonce  $N_A$  and sends to  $B$  a message, containing  $N_A$  as well as his identity, encrypted with the public key of  $B$ :  $K_B$ . Then, in the second step,  $B$  sends back to  $A$  a message encrypted with the public key of  $A$ , containing the nonce  $N_A$  that  $B$  received, a new nonce  $N_B$ , and  $B$ 's identity. Finally, in the last step,  $A$  returns the nonce  $N_B$  he received from  $B$ . If the protocol is completed, mutual authentication of the two agents is ensured:

- as soon as  $A$  receives the message containing the nonce  $N_A$ , sent back by  $B$  at step 2.,  $A$  *believes* that this message was really built and sent by  $B$ . Indeed,  $N_A$  was encrypted with the public key of  $B$  and, thus,  $B$  is the only agent that is able to send back  $N_A$ ,
- similarly, when  $B$  receives the message containing the nonce  $N_B$ , sent back by  $A$  at step 3.,  $B$  *believes* that this message was really built and sent by  $A$ .

Another property that may be expected for this kind of protocol is *confidentiality* of nonces. In particular, if nonces remain confidential, they can be used later as keys for symmetric encryption of communications between  $A$  and  $B$ . Thus, confidentiality of nonces is also of interest. In this part we are going to focus on this last aspect: for agents respecting the protocol and whatever the intruder may do, we expect that nonces remain confidential. The corrected version of the Needham-Schroder public key protocol is encoded in the `example_nspk.txt` file of the distribution.

In this specification file, each agent is labeled by a unique identifier. Let  $L_{agt} = \{A, B, o, s(o), s(s(o)), \dots\}$  be the set of agent labels, where  $A$  and  $B$  are some agents we observe which are supposed to be honest and  $\{o, s(o), \dots\}$  is an infinite set of dishonest agents. For any agent label  $l \in L_{agt}$ , the term  $ident(l)$  will denote the agent whose label is  $l$ . The term  $pubkey(a)$  denotes the public key of agent  $a$  and  $encr(k, a, c)$  denotes the result of encryption of content  $c$  by key  $k$ . In this last term,  $a$  is a flag recording who has performed the encryption. This field is not used by the protocol rules but is used for verification. The term  $N(x, y)$  represents a nonce generated by agent  $x$  for identifying a communication with  $y$ . We also use an AC binary symbol *store* in order to represent sets. For example the term  $store(x, store(y, z))$  (equivalent modulo AC to  $store(store(x, y), z)$  and to  $store(y, store(z, x))$ , etc.) will represent the set  $\{x, y, z\}$ . With regards to this set interpretation of terms, the *store* represent a set union. Like in many other approaches based on the Dolev-Yao, the intruder is considered as being the network itself, i.e. every message can be read, erased, replayed, etc. In our setting the intruder/network is thus a set of messages represented using the *store* symbol.

Starting from a set of initial requests, our aim is to compute a tree automaton recognizing an over-approximation of all possible sent messages with any number of running protocol sessions and

an active intruder. The approximation also contains some terms signaling either communication requests or established communications. For example, a term of the form  $goal(x, y)$  means that  $x$  expect to open a communication with  $y$ . A term of the form  $connect(x, y, z)$  means that  $x$  believes to have initiated a communication with  $y$ , but, in reality  $x$  communicates with  $z$ . The encoding into the TRS is straightforward: each step of the protocol is described thanks to a rewrite rule whose left-hand side is a precondition on the current state (set of received messages and communication requests), and the right-hand side represents the message to be sent (and sometimes established communication) if the precondition is met. This encoding is very similar to the one detailed in [10].

The tree automaton **A0** recognizes the initial configurations (state  $qnet$ ), i.e. any term of the set  $E$  defined inductively as follows

$$E = \{null, ident(A), ident(B), ident(o), ident(s(o)), \dots, pubkey(A), pubkey(B), pubkey(o), pubkey(s(o)), \dots, privkey(o), privkey(s(o)), \dots, goal(A, A), goal(A, B), goal(B, A), goal(B, B), goal(A, o), goal(o, A), goal(o, B), goal(B, o), goal(A, s(o)), \dots, store(t_1, t_2) \mid t_1, t_2 \in E\}$$

Hence, initially the intruder/network knows identity of all the agents, all the public keys, the private keys of the dishonest agents. Terms of the form  $goal(\dots)$  cannot be exploited by the intruder but are needed to initialize the protocol between each pair of agents. Note that connection requests of **A** (resp. **B**) with himself are taken into account but can easily be discarded of initial configurations of the protocol analysis if they are not relevant. For this case study, we assumed that such a behavior may occur.

In the first part of the automaton some priority transitions are defined in order to force some of the terms to be recognized deterministically by a unique (priority) state. This is used for verification purpose or for ensuring left-linearity condition (see section 2.1.5). For left-linearity condition, for instance, since terms matched by non left-linear variables of the rewrite rules of the protocol are agent labels, it is important that agent labels are recognized deterministically. This is why the set of priority transitions contains transitions to force terms  $o, s(o), s(s(o)), \dots$  to be deterministically recognized by state  $Ilabel$ ,  $A$  to be deterministically recognized by  $Alabel$  and  $B$  label by state  $Blabel$ . It is similar for nonces which all have some dedicated (priority) deterministic states.

First, let us try to complete the automaton **A0** without the approximation contained in the file `example_nspk.txt`. This can be done by typing:

```
timbuk --noapprox example_nspk.txt
```

The first step of completion produces some transitions which are already covered by the current automaton and partially normalize another one, which is finally proposed to the user to finish the normalization.

Adding transition:

```
store(store(qnet,qnet),qnet) -> qnet
```

```
... covered by current automaton.
```

Adding transition:

```
store(qnet,store(qnet,qnet)) -> qnet
```

... covered by current automaton.

Adding transition:

```
store(qnet,qnet) -> qnet
```

... already normalised!

Adding transition:

```
store(incr(privkey(Ilabel),o,qnet),privkey(Ilabel)) -> qnet
```

Prior normalisation simplifies the transition into:

```
store(incr(privkey(Ilabel),Ilabel,qnet),privkey(Ilabel)) -> qnet
```

Adding transition:

```
store(incr(privkey(Ilabel),Ilabel,qnet),privkey(Ilabel)) -> qnet
```

To normalize this transition, we can give some new normalization rules. The transition we here have to normalize is of the form `store(t1, t2) -> qnet` where `qnet` is the state recognizing the set of every message of the intruder/network. To normalize this transition, it is enough to remark that if the intruder has the *union* of stores (or message elements) `t1` and `t2` in its knowledge then he reasonably has also `t1` and `t2` independently. Hence we can normalize `t1` by `qnet` and `t2` by `qnet` for every possible `t1` and `t2`. This can be done by adding the following normalization rule: `[store(x, y) -> qnet] -> [x -> qnet y -> qnet]` meaning that for normalizing every transition of the form `store(x, y) -> qnet`, subterm `x` and subterm `y` will be normalized by the state `qnet`. This rule can be added during the completion using the following syntax (first, Timbuk recalls the alphabets and variables on which rules can be built):

Do you want to give by hand some NORMALIZATION rules? (y/n)? y

Current normalisation rules are:

```
Alphabet=goal:2 store:2 null:0 incr:3 pubkey:1 privkey:1 N:2 cons:2 ident:1 o:0  
s:1 A:0 B:0 connect:3
```

```
and Variables= x y z u v w m
```

```
and States= Ilabel:0 qnet:0 Alabel:0 Blabel:0 Aident:0 Bident:0 Iident:0 NAB:0  
NAA:0 NBB:0 NBA:0 NI:0
```

Type additionnal normalization rules using the 'States' and 'Rules' keyword and end by a dot '.':

(use keyword 'Top' to place a rule at the beginning of the rule list)

Rules

```
[store(x, y) -> qnet] -> [x -> qnet    y -> qnet].
```

This lead to the automatic normalization of many new transitions produced by the completion. The next new transition the user is proposed is the following:

Adding transition:

```
encr(pubkey(Alabel), Ilabel, cons(NI, Iident)) -> qnet
```

This means that the intruder has received in its knowledge (**qnet**) a new term which is of the form **encr(pubkey(Alabel), x, m)** i.e. a message **m** encrypted with the public key of **A**. In this case, it is a bad idea to normalize **m** with the state **qnet** since it would directly give the secret message **m** to the intruder though it is encrypted with the public key of **A** (and should remain secret, if the protocol is correct). Normalizing **m** with **qnet** would thus build a too big over-approximation where this secret is given to the intruder. On the opposite, it is possible to define a particular state (say **qAcontent**) for recognizing *every* secret belonging to **A**. It is also necessary to define a new specific state **qAkey** for recognizing **pubkey(Alabel)**. Defining those new states and the new normalisation rules can be done interactively using the following syntax:

States **qAcontent** **qAkey**

Rules

```
[encr(pubkey(Alabel), x, y) -> z] ->
  [ y -> qAcontent
    pubkey(Alabel) -> qAkey ].
```

where every subterm **y** under an encryption with the public key of **A** will be normalized using the **qAcontent** state. The following transition to normalize is similar to the previous one but for **B**: **encr(pubkey(Blabel), Ilabel, cons(NI, Iident)) -> qnet**. The normalization rule to add is thus of the same form:

States **qBcontent** **qBkey**

Rules

```
[encr(pubkey(Blabel), x, y) -> z] ->
  [ y -> qBcontent
    pubkey(Blabel) -> qBkey ].
```

Next transition is also concerned with the public encryption of a message but this time with the public key of dishonest agents all recognized by state **Ilabel**. Like in the previous cases, we could add a specific state for recognizing the encrypted message, however, since the intruder knows the private key of those agents it is likely to obtain the content of the encrypted message anyway. Hence, it is not erroneous to normalize the encrypted message with **qnet** (and put the content of the message directly in the intruder's knowledge). Here, using state **qnet** instead of a new dedicated state permits to produce a more compact approximation that is still correct with regards to secrecy properties for **A** and **B**. It is possible to do the same with the subterm **pubkey(Ilabel)**. Here is the corresponding normalization rule to add interactively:

## Rules

```
[encr(pubkey(Ilabel), x, y) -> z] ->
  [ y -> qnet
    pubkey(Ilabel) -> qnet].
```

Note that in previous transitions, normalizing `pubkey(Alabel)` and `pubkey(Blabel)` would have built a too big approximation losing secrecy properties associated to  $A$  and  $B$ . Indeed normalizing `pubkey(Alabel)` by `qnet` in a transition of the form `encr(pubkey(Alabel),x, m) -> qnet` would produce two new transitions, namely: `pubkey(Alabel) -> qnet` and `encr(qnet,x, m) -> qnet`. The problem does not come from the first one (since the intruder already has the public key of  $A$ ) but from the second since with this last transition and the transition `pubkey(Ilabel) -> qnet` that is already in the automaton, the intruder can build the term `encr(pubkey(Ilabel),x, m) -> qnet`. Then, since `privkey(Ilabel)` is also in `qnet`, the intruder can apply decryption on the last term and obtain  $m$  in clear.

Adding the last normalization rule permits to end the first completion step. In the next completion step, we are successively proposed the following new transitions to normalize:

```
cons(NI,cons(NI,Bident)) -> qnet
cons(NAA,cons(NAA,Aident)) -> qAcontent
cons(NBA,cons(NAB,Aident)) -> qBcontent
```

All those transitions represent structured messages respectively stored in the intruders knowledge,  $A$  secret message content, and  $B$  secret message content. One could now define some new secret states for recognizing the (secret) subterms of those messages. However, we can also do a more drastic approximation by using the three same states to normalize the subterms, i.e. collapse the message structure:

## Rules

```
[cons(x,y) -> qnet]          -> [y -> qnet]
[cons(x, y) -> qAcontent] -> [y -> qAcontent]
[cons(x, y) -> qBcontent] -> [y -> qBcontent] .
```

This approximation does not invalidate the secrecy property of the protocol and make the approximation more compact. Note that those three rules can be equivalently replaced by the following normalization rule: `[cons(x, y) -> z] -> [y -> z]`. This is the last approximation rules to give and the remaining completion steps are performed automatically within some minutes. Finally the automaton is complete. Now to prove the secrecy properties, two steps are necessary. First, since the TRS used for completion is non left-linear, to guarantee that this automaton is really an over-approximation of  $\mathcal{R}^*(E)$ , it is necessary to verify the left-linearity condition. This condition can be automatically verified on the completed automaton (see section 2.1.5 for details). The second step, necessary to prove that secrecy of honest nonces is guaranteed consists in computing the intersection between the completed automaton and an automaton describing all the possible cases where an honest nonce has been captured by the intruder. This last automaton is the automaton Problems of the `example_nspk.txt` file. This automaton recognizes any term of the form `store(N, t)` where  $t$  is any term built on the alphabet and  $N$  is any term in the set  $N(A,B)$ ,  $N(A,A)$ ,  $N(B,B)$ ,  $N(B,A)$ , i.e. every possible nonces produced by an honest agent for an other honest agent. Typing `i` in the menu make Timbuk compute an intersection between the

completed automaton and the automaton problems and results into an empty intersection, meaning that those nonces cannot be grabbed by the intruder.

Note that this can also be checked using the pattern matching. Type `p` and then the pattern `store(N(A,B), x)` for instance. This pattern has no solution meaning that this term is not reachable. For a more general verification, now type `p` and pattern `store(N(x, y), z)`. This results in the following output:

Solutions:

Occurrence in state `qnet`!

```
solution 1: x <- Alabel, y <- Ilabel, z <- NI
solution 2: x <- Ilabel, y <- Ilabel, z <- NI
solution 3: x <- Ilabel, y <- Blabel, z <- NI
solution 4: x <- Ilabel, y <- Alabel, z <- NI
solution 5: x <- Blabel, y <- Ilabel, z <- NI
solution 6: x <- Alabel, y <- Ilabel, z <- Iident
solution 7: x <- Ilabel, y <- Ilabel, z <- Iident
solution 8: x <- Ilabel, y <- Blabel, z <- Iident
solution 9: x <- Ilabel, y <- Alabel, z <- Iident
solution 10: x <- Blabel, y <- Ilabel, z <- Iident
solution 11: x <- Alabel, y <- Ilabel, z <- Aident
solution 12: x <- Ilabel, y <- Ilabel, z <- Aident
solution 13: x <- Ilabel, y <- Blabel, z <- Aident
solution 14: x <- Ilabel, y <- Alabel, z <- Aident
solution 15: x <- Blabel, y <- Ilabel, z <- Aident
solution 16: x <- Alabel, y <- Ilabel, z <- Bident
solution 17: x <- Ilabel, y <- Ilabel, z <- Bident
solution 18: x <- Ilabel, y <- Blabel, z <- Bident
solution 19: x <- Ilabel, y <- Alabel, z <- Bident
solution 20: x <- Blabel, y <- Ilabel, z <- Bident
solution 21: x <- Alabel, y <- Ilabel, z <- qnet
solution 22: x <- Ilabel, y <- Ilabel, z <- qnet
solution 23: x <- Ilabel, y <- Blabel, z <- qnet
solution 24: x <- Ilabel, y <- Alabel, z <- qnet
solution 25: x <- Blabel, y <- Ilabel, z <- qnet
```

meaning that nonces produced *by* or produced *for* a dishonest agent (`x` or `y` is associated to `Ilabel`) have been captured but none of the fully honest ones (where `x` and `y` have been associated to `A` or `B`).

Now, let us try to check the authentication property. Recall that a term of the form `connect(x,y,z)` means that `x` believes to have initiated a communication with `y` but in reality `x` is communicating with `z`. Type `p` and search for the pattern `connect(x, y, z)` in the completed automaton. This produces the following output:

Solutions:

Occurrence in state `qnet`!

```
solution 1: x <- Blabel, y <- Ilabel, z <- Ilabel
solution 2: x <- Alabel, y <- Alabel, z <- Ilabel
solution 3: x <- Alabel, y <- Blabel, z <- Ilabel
```

```

solution 4: x <- Blabel, y <- Ilabel, z <- Blabel
solution 5: x <- Ilabel, y <- Ilabel, z <- Blabel
solution 6: x <- Ilabel, y <- Blabel, z <- Blabel
solution 7: x <- Ilabel, y <- Ilabel, z <- Ilabel
solution 8: x <- Ilabel, y <- Alabel, z <- Ilabel
solution 9: x <- Ilabel, y <- Ilabel, z <- Alabel
solution 10: x <- Ilabel, y <- Blabel, z <- Alabel
solution 11: x <- Alabel, y <- Alabel, z <- Alabel
solution 12: x <- Alabel, y <- Alabel, z <- Blabel
solution 13: x <- Blabel, y <- Blabel, z <- Alabel
solution 14: x <- Blabel, y <- Alabel, z <- Alabel
solution 15: x <- Alabel, y <- Blabel, z <- Blabel
solution 16: x <- Alabel, y <- Blabel, z <- Alabel
solution 17: x <- Ilabel, y <- Alabel, z <- Alabel
solution 18: x <- Alabel, y <- Ilabel, z <- Alabel
solution 19: x <- Blabel, y <- Ilabel, z <- Alabel
solution 20: x <- Ilabel, y <- Blabel, z <- Ilabel
solution 21: x <- Ilabel, y <- Alabel, z <- Blabel
solution 22: x <- Alabel, y <- Ilabel, z <- Blabel
solution 23: x <- Blabel, y <- Alabel, z <- Blabel
solution 24: x <- Blabel, y <- Blabel, z <- Blabel
solution 25: x <- Alabel, y <- Ilabel, z <- Ilabel
solution 26: x <- Blabel, y <- Alabel, z <- Ilabel
solution 27: x <- Blabel, y <- Blabel, z <- Ilabel

```

where some solutions are not satisfactory with regards to authentication. For instance, solution 3 says that **A** thinks that he is talking to **B** whereas it is talking to **I** (any dishonest agent). In fact this is not an error of the protocol but it is due to an approximation function which is too drastic to prove the authentication (see section 2.1.6 for a more precise approximation function and the proof of the authentication property).

### 2.1.5 Verifying left-linearity condition

At the end of the previous successful completion, by typing **v** in the Timbuk menu, one can verify the left-linearity condition (see [7] for details) on the non left-linear TRSs used for modeling the protocol to guarantee that the completed automaton recognizes an over-approximation of  $\mathcal{R}^*(E)$ . On this example, after the full completion, by typing **v** we obtain within a few seconds:

```

Checking intersection: Ilabel ^ Alabel ... done.
Checking intersection: Alabel ^ Blabel ... done.
Checking intersection: Ilabel ^ Blabel ... done.
No linearity problem!

```

meaning that left-linearity condition is fulfilled. What Timbuk does is that it searches for every possible state matched by non left-linear variables and proves that if the states matched by non linear variables are different then the languages recognized by those states are disjoint. This is here the case for states **Ilabel**, **Alabel** and **Blabel**. When it is not the case, it is necessary to modify the normalization rules or the prioritary rules so that those states recognize disjoint languages.



### 2.1.6 Doing more and going faster

Once your approximation are established, it is possible to store it directly in the specification file, see approximation `Secret` in file `example_nspk.txt` for instance. Then it is possible to directly start a completion process with the first approximation by typing:

```
timbuk example_nspk.txt
```

In this file, there is a second approximation called `SecAndAuth` that permits to prove both the secrecy and the authentication property which can be used instead of the first one thanks to the Timbuk option `--approx SecAndAuth`. However, since this completion takes some time, and since this set of approximation rules is known to be complete w.r.t. the completion to perform (i.e. no manual interaction is needed) it is also possible to use the experimental `static` completion algorithm (see section 4.4.2) with the `--static` Timbuk option:

```
timbuk --approx SecAndAuth --static example_nspk.txt
```

Type `a` to achieve the full completion at once. Type `v` to verify the left-linearity condition (note that it is also faster in static mode), then type `i` and check that honest nonces are still not captured by the intruder. Then type `p` and search for pattern `connect(x, y, z)`. This results in the following output:

Solutions:

Occurence in state qnet!

```
solution 1: x <- Alabel, y <- Ilabel, z <- Ilabel
solution 2: x <- Blabel, y <- Ilabel, z <- Blabel
solution 3: x <- Blabel, y <- Ilabel, z <- Alabel
solution 4: x <- Ilabel, y <- Ilabel, z <- Ilabel
solution 5: x <- Ilabel, y <- Blabel, z <- Alabel
solution 6: x <- Ilabel, y <- Blabel, z <- Blabel
solution 7: x <- Ilabel, y <- Alabel, z <- Ilabel
solution 8: x <- Alabel, y <- Blabel, z <- Blabel
solution 9: x <- Blabel, y <- Alabel, z <- Alabel
solution 10: x <- Alabel, y <- Alabel, z <- Alabel
solution 11: x <- Blabel, y <- Blabel, z <- Blabel
solution 12: x <- Ilabel, y <- Alabel, z <- Blabel
solution 13: x <- Ilabel, y <- Alabel, z <- Alabel
solution 14: x <- Ilabel, y <- Blabel, z <- Ilabel
solution 15: x <- Ilabel, y <- Ilabel, z <- Blabel
solution 16: x <- Ilabel, y <- Ilabel, z <- Alabel
solution 17: x <- Blabel, y <- Ilabel, z <- Ilabel
solution 18: x <- Alabel, y <- Ilabel, z <- Alabel
solution 19: x <- Alabel, y <- Ilabel, z <- Blabel
```

This results shows that whenever a dishonest agent is concerned by a communication, authentication is not guaranteed: lines 2, 3, 5, 7, 12, 14, 15, 16, 18, 19 shows each time that `x` is connect to someone else that he expects. On the opposite, each time that `x` and `y` range over honest agents, values for `y` and `z` coincide (lines 8, 9, 10, 11). Hence, for honest agents, this protocol guarantees the authentication.

**Remark on approximation definition in static mode:** When defining approximation rules to be used in the static mode, note that Timbuk may consider that your set of approximation rule

is not complete though you know it is. This is the case for the file `example_nspk.txt`: if you have a careful look to the approximation `Secret` it contains the rules established in section 2.1.4 as well as an additional at the end of the rule set: `[x -> y] -> [z -> qnet]` ensuring that every subterm that has not already been normalized by the previous rule is to be normalized by state `qnet`. This is a trick to help Timbuk static completion algorithm to admit that this approximation is *complete*. Note that instead of completing by hand the approximation rule set it is also possible to use the `--fstatic` option that automatically adds a default rule of the same kind and thus never complains about incomplete normalization rule sets.

### 2.1.7 More tricks

Syntax of normalization rules is in fact a bit less restrictive than what is said in the previous section. Let us retry to complete the `basic.txt` file:

```
timbuk basic.txt
```

During the first completion step we are proposed to give some normalization rules. Let us define a state operator (see section 3.6.2 for details about state operators) and write interactively some normalization rules in extended syntax:

```
States q:1
Rules [g(x) -> y] -> [x -> q(x)].
```

The effect of this rule is to normalize every subterm  $t$  of a transition  $g(t) \rightarrow q'$  by a state labeled by  $q(t)$ . This single normalization rule permits to achieve the completion automatically till the end. Here is a more practical example. Using this extended syntax, the normalization rules given in section 2.1.4 for proving the secrecy on the NSPK cryptographic protocol, can be abbreviated as follows (approximation called `Secret2` in `example_nspk.txt` file):

```
Approximation Secret2
States q:1 secret:1 qnet key:1 Alabel Ilabel Blabel
Rules
```

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

[encr(pubkey(Ilabel), x, y) -> z] ->
[ y -> qnet
 pubkey(Ilabel) -> qnet]
```

```
(* Every message component encrypted by someone else than the intruder goes in a
dedicated state *)
```

```
[encr(pubkey(u), x, y) -> z] ->
[ y -> q(secret(u))
 pubkey(u) -> q(key(u))]
```

```
(* In the storage states, everything is collapsed (structure of the message is
not important) *)
```

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

Recall that approximation rules are used in the order. Hence, every message encrypted by a dishonest agent will be normalized using the second rule and every message encrypted by an (honest) agent  $X$  matched by variable  $u$  will be normalized using the third rule and states  $q(\text{secret}(X))$  and  $q(\text{key}(X))$ . It is possible to reach completion using this new approximation. However, since this extended syntax cannot be used in static mode, we need to achieve completion in dynamic (default) mode:

```
timbuk --approx Secret2 example_nspk.txt
```

Some other tricks for building approximation are still under development but are already integrated in Timbuk for testing: merging rules, approximation equations and interactive merging with Tabi. Merging rules (see section 4.2.3) are rules of the form  $q1 \rightarrow q2$  for merging two states in an automaton. Such rules can be given to Timbuk explicitly using the `m` command, or they can be built interactively using Tabi (see section 2.3.3). Approximation equations are a third way to merge some states of the automaton by giving some equivalence between some terms (patterns in fact).

Here is a simple example done on the `processes.txt` file. This example consists of a TRS modeling the behavior of two parallel processes counting elements on a shared counter that should not be accessed by the two processes at the same time (see [7] for details on this example). If we start a completion with an exact normalization strategy:

```
timbuk --strat exact processes.txt
```

Then completion diverges. This comes from the fact in the initial language the number of elements to be counted by processes is not bounded. Hence, the counter (built on the usual Peano operators for naturals: `o` and `s()`) counts an infinite number of elements. Divergence of completion, can easily be pruned adding interactively an approximation equation. In our case, we achieved completion until the 6th completion step then add the following approximation equation merging together all the naturals greater to 0:

Current equations are:

```
Alphabet=S:4 Proc:2 cons:2 null:0 busy:0 free:0 s:1 o:0
and Variables= x y z u
```

```
Type additionnal equations and end by a dot '.':
s(s(x))=s(x).
```

This equation permits to merge some of the states of the automaton:

State merging using approximation equations!

```
qnew8 -> qnew9
qnew6 -> qnew9
```

Then, doing another completion step permits to end the completion process. It is possible to check that both processes have never accessed the counter at the same time by verifying that the pattern  $S(\text{Proc}(\text{busy}, x), \text{Proc}(\text{busy}, y), z, u)$  has no solution in the automaton:

```

Alphabet=
S:4 Proc:2 cons:2 null:0 busy:0 free:0 s:1 o:0

States=
qnew9:0 qnew8:0 qnew7:0 qnew6:0 qnew5:0 qnew4:0 qnew3:0 qnew2:0 qnew1:0 qnew0:0 q0:0
q1:0 q2:0 q3:0 q4:0

Variables=
x y z u

Type a term and hit Return: S(Proc(busy, x), Proc(busy, y), z, u)

Pattern not found!

```

## 2.2 Tam1

Start Tam1 by typing: `tam1` in a command line window. Tam1 is an Ocaml interpreter extended with Timbuk library functionalities (see section 5 for reference manual of Tam1 and see [12] for details about Ocaml syntax). The following tutorial is a step by step construction of TRS and automata. However, if necessary, the whole tutorial file can be executed at once by loading the file in Tam1, using the following Ocaml directive `#use "tutorial.ml"`.

First, let us define an alphabet `f` by typing the following Tam1 commands (commands are prefixed by the `#` symbol which represents the usual Ocaml prompt, this of course has not to be typed by the user):

```
# let f= alphabet "app:2 cons:2 nil:0 a:0 b:0";;
```

Tam1 gives the following output, meaning that `f` has been accepted as a valid alphabet.

```
val f : Tam1.Alphabet.t = app:2 cons:2 nil:0 a:0 b:0
```

Similarly one can define a variable set `v`:

```
# let v= varset "x y z u";;
val v : Tam1.Variable_set.t = x y z u
```

Now, let us define a term `t` over the alphabet `f` and the variable set `v` as follows:

```
# let t= term f v "cons(a, cons(b, nil))";;
val t : Tam1.Term.t = cons(a,cons(b,nil))
```

Since Tam1 embeds a complete Ocaml interpreter, it is thus possible to use usual Ocaml syntax facilities and also to combine Tam1 functions with usual Ocaml functions. For instance, it is possible to define a specific `term` function specialized for alphabet `f` and variable set `v` in the following way:

```
# let fvterm= term f v;;
val fvterm : string -> Tam1.Term.t = <fun>
```

Now it is possible to construct a list of terms built on alphabet **f** and variable set **v** using the specialized function **fvterm** as well as Ocaml **List.map** function (mapping a function to every element of a list) in the following way:

```
# let l= List.map fvterm ["app(cons(a, nil),cons(b, cons(b, nil)))"; "a"; "cons(a,nil)"];;
val l : Tam1.Term.t list = [app(cons(a,nil),cons(b,cons(b,nil)))
;
a
;
cons(a,nil)
]
```

Similarly we can construct term rewriting systems and tree automata directly in the interpreter:

```
# let tt= trs f v "app(nil, x) -> x    app(cons(x, y), z) -> cons(x, app(y, z))";;
val tt : Tam1.Rewrite.t =
  app(nil,x) -> x
app(cons(x,y),z) -> cons(x,app(y,z))

# let aut= automaton f "
States qa qb q1a q1b qf
Final States qf
Transitions
  a -> qa
  b -> qb
  cons(qa, q1a) -> q1a
  nil -> q1a
  cons(qb, q1b) -> q1b
  nil -> q1b
  app(q1a,q1b) -> qf";;

val aut : Tam1.Automaton.t =
  States qa:0 qb:0 q1a:0 q1b:0 qf:0

Final States qf

Transitions
a -> qa
b -> qb
cons(qa,q1a) -> q1a
nil -> q1a
cons(qb,q1b) -> q1b
nil -> q1b
app(q1a,q1b) -> qf
```

Now let us show that a given term is recognized by a given state in a tree automaton

```

# let t1= List.hd l;;
val t1 : Taml.Term.t = app(cons(a,nil),cons(b,cons(b,nil)))

# let s= state "qf";;
val s : Taml.Automaton.state = qf

# run t1 s aut;;
- : bool = true

```

One can also rewrite terms using the term rewriting system `tt` and the `Rewrite.left_inner_norm` function of the Timbuk library (see section 5.3 for details on use of Timbuk functions outside of Taml interface):

```

# let t2= Rewrite.left_inner_norm tt t1;;
val t2 : Taml.Term.t = cons(a,cons(b,cons(b,nil)))

```

It is also possible to read automaton and TRS from a Timbuk specification file. For instance, let us read the automata `completed_A0` and the TRS `current_TRS` in the file `comp.txt` which corresponds to the completion done in section 2.1.2.

```

# let tt= read_trs "current_TRS" "comp.txt";;
val tt : Taml.Specification.trs =
  app(nil,x) -> x
  app(cons(x,y),z) -> cons(x,app(y,z))
  rev(nil) -> nil
  rev(cons(x,y)) -> app(rev(y),cons(x,nil))

# let aut= read_automaton "completed_A0" "comp.txt";;
val aut : Taml.Specification.automaton =
  States qlba:0 qla:0 qnil:0 qrev:0 qlab:0 qlb:0 qa:0 qb:0

```

Description

```

qrev: "rev applied to lists where a are before b"
qlab: "lists where a are before b (possibly empty)"
qlb: "lists of b (poss. empty)"

```

Final States qrev

Prior

```

app(qla,qla) -> qla
rev(qnil) -> qnil
app(qnil,qla) -> qla
app(qnil,qlb) -> qlb
cons(qb,qnil) -> qlb
app(qlb,qla) -> qlba
rev(qlab) -> qlba
cons(qa,qnil) -> qla

```

```

nil -> qnil
rev(qlb) -> qlb

```

Transitions

```

rev(qlab) -> qrev
nil -> qlab
cons(qa,qlab) -> qlab
cons(qa,qlb) -> qlab
nil -> qlb
cons(qb,qlb) -> qlb
a -> qa
b -> qb
nil -> qrev
rev(qlab) -> qlba
app(qlba,qla) -> qrev
rev(qlb) -> qlb
nil -> qnil
cons(qa,qnil) -> qla
app(qlb,qla) -> qrev
cons(qa,qnil) -> qrev
cons(qb,qlba) -> qrev
nil -> qlba
app(qlba,qla) -> qlba
app(qlb,qla) -> qlba
cons(qb,qnil) -> qlb
app(qnil,qlb) -> qlb
app(qnil,qla) -> qla
rev(qnil) -> qnil
app(qla,qla) -> qla
app(qlb,qlb) -> qlb
cons(qa,qnil) -> qlba
app(qnil,qla) -> qlba
app(qla,qla) -> qlba
cons(qb,qlba) -> qlba
cons(qb,qla) -> qlba
cons(qa,qla) -> qla
app(qnil,qla) -> qrev
app(qla,qla) -> qrev
cons(qb,qla) -> qrev
cons(qa,qla) -> qlba
cons(qa,qla) -> qrev

```

Now we can compute the automaton recognizing the set of terms irreducible by TRS `current_TRS` by typing the following command:

```

# let aut_iff= irr f tt;;
val aut_iff : Tam1.Automaton.t =
  States q2:0 q1:0 q0:0

```

Final States q0 q1 q2

Transitions

```
b -> q2
a -> q2
nil -> q1
app(q2,q2) -> q2
app(q2,q1) -> q2
cons(q1,q1) -> q0
cons(q2,q2) -> q0
cons(q2,q1) -> q0
cons(q1,q2) -> q0
cons(q0,q0) -> q0
cons(q2,q0) -> q0
cons(q1,q0) -> q0
cons(q0,q2) -> q0
cons(q0,q1) -> q0
app(q2,q0) -> q2
```

Now, recall that in section 2.1.2 the automaton `completed_A0` (stored in the Ocaml variable `aut`) of the file `comp.txt` recognizes an over-approximation of  $R^*(\mathcal{L}(A0))$  where  $A0$  and  $R$  are respectively the automaton and the TRS defined in file `example2.txt` (and such that  $R = \text{current\_TRS}$ ). We can thus construct the automaton recognizing an over approximation of the set of normal forms  $R^!(\mathcal{L}(A0))$  as follows:

```
# let norm= inter aut aut_iff;;
```

However, the intersection automaton is very big and not cleaned (it may have some unnecessary states). Furthermore, for efficiency reasons, our implementation of intersection does not build explicitly the set of states of the intersection automaton. To obtain a finalized automaton, it is necessary to use cleaning functions such as `simplify`:

```
# let norm2= simplify norm;;
val norm2 : Taml.Automaton.t =
  States q7:0 q6:0 q5:0 q4:0 q3:0 q2:0 q1:0 q0:0
```

Final States q6 q7

Transitions

```
nil -> q1
nil -> q0
nil -> q7
b -> q3
a -> q4
cons(q4,q0) -> q5
cons(q4,q0) -> q6
cons(q3,q1) -> q6
```



```

cons(q4,q0) -> q2
cons(q3,q1) -> q2
cons(q4,q5) -> q2
cons(q3,q5) -> q2
cons(q3,q2) -> q2
cons(q3,q2) -> q6
cons(q4,q5) -> q5
cons(q3,q5) -> q6
cons(q4,q5) -> q6

```

This automaton represents an over-approximation of  $R^!(\mathcal{L}(A0))$ . To have a more precise idea of the recognized language, one can browse it using Tabi:

```
# browse norm2;;
```

Then click on the **Start** symbol and then on the button **choose random** to build some randomized representatives of the language. The representatives are all lists where **b**'s are always before **a**'s which corresponds to the definition of the reverse function applied on lists of **a**'s followed by some **b**'s. Details on Tabi use will be given in the next section. For the moment, just quit random and quit Tabi. Note that in the automaton, there remains only constructor symbols (functional symbols **app** and **rev** have disappeared). This proves that definition of reverse is complete w.r.t. the lists we have considered (see [9] for details). To conclude on this tutorial for Tam1, note that Tam1 provides a small online help on the most used functions by typing:

```
# help();;
```

## 2.3 Tabi

### 2.3.1 Basic

To start Tabi, simply type **tabi** in a command line window. Then open the automaton  $A0^3$  of the **example3.txt** Timbuk specification file using the file browser: choose the **Open File** item of the **File** menu and browse the directories to open the file **example3.txt**. After a while the **Start** symbol is displayed in the Tabi window. Click on it and choose in the list the final state to start from. For instance, click on final state **qf1**. Now we are going to browse the automaton to build some representatives of the language recognized by this final state. Click with the left mouse button on the state **qf1**. A window opens. It contains a list of configurations (or terms) leading to this state. Choose configuration **times(q0,q0)**. The state **qf1** is replaced by the selected configuration. This is what we call *unfolding* of a state.

Now click on a state **q0**, replace it by the unique possible configuration: **O**. Then do the same for the other occurrence of state **q0**. We have obtained a ground term recognized by state **qf1** in the tree automaton  $A0$ . Note that moving the mouse pointer over the term and its subterms displays in red the state recognizing the selected subterm.

Instead of building terms by hand, it is also possible to produce random representatives. Use a middle click over term **times(O,O)** to *fold* it back to the state **qf1**. Now use left click on **qf1** again to open the configuration list window. Then instead of choosing a particular configuration, click on button **choose random**. A new window opens containing a list of randomly generated representatives

---

<sup>3</sup>Tabi always reads the first automaton of the specification file

for state **qf1**. In our case, this list should contain exactly 3 representatives which is in fact exactly the language recognized by **qf1**. Click on one of them to use it to replace state **qf1**.

Now assume, that you want to restart browsing from a different final state, say **qnew23**. This can be done by clicking on button **Restart** which reinitializes the browsing from the **Start** symbol seen at the beginning. Click on **Start** symbol and select the final state **qnew23**. Produce a random representative for **qnew23** as seen before. This state does not recognize a finite language and the randomly generated terms are bigger and more numerous. Now change the values for random upper bound for term depth, random upper bound for time and random upper bound for random term number using respectively the items **Random max depth**, **Random max time** and **Random max term number** of the **Options** menu and see the effect on randomized term generation. For instance, refold state **qnew23** with a right click on the top of the term and set **Random max depth** and **Random max term number** to 10. Then produce random terms for state **qnew23**. Produced terms are lesser and smaller.

Now assume that you want to browse state **qnew20** which is not final. Click on the **Browsing** style item of the **Options** menu, and click on the **All states** button. Then click on the **Restart** button. Now, by clicking on the start symbol, it is possible to browse any state of the automaton such as **qnew20**.

### 2.3.2 Display modes

The default displaying mode you are using is **autozoom** (at that moment **autozoom** should be selected in the Tabi windows) meaning that Tabi tends to display the whole browsed term as big as possible in the window. When the term size is getting bigger and bigger, the font is reduced so that it can still be displayed in the window. When the font is getting too tiny, Tabi automatically switches to **zoom** mode where only a part of the term is displayed and one can move from a part to another using the scrolling bars.

To show the different display modes we are going to browse some big terms. Let us first construct big terms. Set the values of **Random max depth**, **Random max time** and **Random max term number** respectively to 10000, 20 and 10000. Note that, to produce bigger random terms, it is not enough to increase the **Random max depth** value since it is only an upper bound for term depth. For instance, if **Random max time** is set to 10000 and **Random max term number** is set to 2, then random generation will stop when 2 random terms have been produced. Since this generation starts from the smaller terms that can be produced, the set of randomly generated terms is likely to contain the smallest possible terms. Similarly, it is necessary to increase the **Random max time** value in order to give Tabi the time necessary to consider deeper terms.

Now, produce randomly generated representatives for the state **qnew20** and choose the deepest one. This term is displayed and the font is reduced so that it fits in the window. If the term is too big (or the window too small) then Tabi switches to the **zoom** mode. The term is displayed in **linear mode**. Now hold the **CTRL** key pressed and do a left click on the whole term (the term should be entirely selected). Now the whole term is displayed in **tree mode**. It is also possible to mix both modes by switching from a mode to another on subterms. For instance, your term is likely to contain a tall subtree built on **s** symbols. Hold the **SHIFT** key pressed and do a left click on the top of this subtree to switch back its representation into linear mode. You should obtain something close to Figure 1. There are some other ways to switch from a mode to another (see Tabi's reference manual in section 6 for details).

The default displaying mode is the **linear mode** i.e. unfoldings are presented in linear mode. The user may switch from this mode to the **tree mode** by clicking on the corresponding button.



### 3 Specification language reference manual

In a Timbuk specification file, it is possible to define one alphabet (mandatory) and a set of variable. Those elements are followed by any number of Term Rewriting Systems, Tree Automata and Approximations all of them associated with a distinct name. Have a look to Figure 2 for a sample Timbuk specification file.

```
Ops
  f:2 g:1 a:0 b:0

Vars x y z u

TRS R1

  f(x, y) -> g(f(x, y))
  g(a) -> f(a, a)
  g(x) -> f(x, x)

Set A0
f(a, a)
f(b, b)
f(g(a), g(a))

Automaton A1
States qa q[1--4]

Description qa : "exactly a"
  q1 : "g*(a)"
  q2 : "g(g*(a))"
  q3 : "any term built on a and f"

Final States q4
Transitions
  a -> qa
  a -> q1
  g(q1) -> q1
  g(q1) -> q2
  a -> q3
  f(q3, q3) -> q3
  f(q2, q3) -> q4

Approximation first
Import A1
States qg
Rules
  [x -> y]          -> [a -> qa]
  [g(x)-> q2]       -> [x -> q2]
  [g(x)-> y]        -> [x -> qg]

Equations
  f(f(x, y), z)= f(x, y)
```

Figure 2: A sample Timbuk specification

### 3.1 Comments

The comments in Timbuk specification files respect the Ocaml syntax, i.e. should be opened with `(*` and closed with `*)`.

### 3.2 Symbols

The symbols used in Timbuk are sequences of characters that should not contain the following characters: `'('`, `')`, `'*'`, `'-'`, `'='`, `':'`, `'['`, `']'` nor contain a comma, a space or one of the reserved keyword defined in the following.

### 3.3 Alphabets

Alphabets are sequences of pairs of symbols associated with an arity (a natural number). Symbols are associated to their arity using the `':'` character. In specification files, alphabets should be prefixed by the `Ops` keyword.

### 3.4 Variable sets

Variable sets are sequences of symbols that should be all distinct from the symbols of the alphabet. In specification files, alphabets should be prefixed by the `Var` keyword.

### 3.5 Term Rewriting Systems

Term rewriting systems are sequences of rewrite rules, where a rule is a pair of terms, built on the alphabet and the variable set of the specification, separated by `->`. Terms should be written in prefix notation. In specification files, every term rewriting systems declaration should begin with the `TRS` keyword followed by a name (following the symbol syntax defined above).

### 3.6 Tree Automata

There are two different manners to define tree automata in a Timbuk specification file: implicitly by giving the (finite) set of terms to be recognized or explicitly by giving the set of states, the set of final states and the set of transitions.

#### 3.6.1 Implicit definitions

It is now possible to define a tree automaton by giving the finite set of terms it should recognize, i.e. its finite language. In specification files, an implicit definition of an automaton consists in the keyword `Set` followed by a name (following the symbol syntax defined above) and by a finite sequence of terms built on the alphabet of the specification.

#### 3.6.2 Explicit definitions

Tree Automata are defined *explicitly* using the five keywords `States`, `Description`, `Final States`, `Prior` and `Transitions` in that order, where `Prior` and `Description` are optional:

**States** is followed by a sequence of state operators. Unlike in usual tree automata the state operators we used are not necessarily constant symbols. One may define constant states symbols:  $q1:0$   $q2:0$  ... but also some “state operators” which transform any term into a state:  $q:1$   $prod:2$  .... With such definitions, constants  $q1$ ,  $q2$  will denote states but assuming that  $f(a)$  is a term defined on the specification alphabet  $q(f(a))$ ,  $prod(f(a), q(f(a)))$ ,  $prod(q1, q2)$  will also be some valid states. Note that for convenience, when constant state operators are defined the notation  $q1:0$ ,  $q2:0$  can be abbreviated into  $q1$   $q2$ . Similarly, the notation  $q1$ ,  $q2$ ,  $q3$ ,  $q4$ ,  $q6$  can be abbreviated into  $q[1--6]$ .

**Description** is followed by a sequence of state description, where a state description is a pair composed with a state and a string separated by the  $:$  symbol. A description is any string delimited by two " symbols (See Figure 2 for an example).

**Final States** is followed by a sequence of *states*. A *state* is in fact a term rooted by a state operator. For instance, if the declared state operators are:  $q1:0$   $q2:0$   $q:1$   $prod:2$  and  $f(a)$  is a term defined on the specification alphabet, a valid sequence of states can be  $q1$   $q2$   $q(q1)$   $prod(f(a), q1)$   $prod(q1, q2)$ .

**Prior** is an optional keyword followed by a sequence of automata transitions. Those transition will represent some *prioritary* transitions for approximation construction, see section 4.2.1. Note that *prioritary* transitions are supposed to be a subset of the transitions. As a consequence, *prioritary* transitions are always added to the set of declared transitions of the automaton. Hence, if a transition is declared as *prioritary* it is not necessary to repeat it in the **Transitions** section since it will automatically be added. Syntax of transition sequences is detailed in the next item.

**Transitions** is followed by a sequence of transitions. A transition is a pair composed with a term (also called a configuration) and a state separated by  $\rightarrow$ . Timbuk only accept normalized transitions so the term on the left-hand side of the pair should be of the form  $f(q_1, \dots, q_n)$  where  $f$  is a symbol of the alphabet declared with arity  $n$  and  $q_1, \dots, q_n$  are states.

In specification files, every explicit tree automata declaration should begin with the **Automaton** keyword followed by a name (following the symbol syntax defined above).

### 3.7 Approximations

Approximations are defined using the three keywords **Import**, **States**, **Equations** and **Rules**. They are all optional. However, if **Import** and **States** are present, **Import** should always be placed before **States**.

**Import** is followed by a sequence of tree automaton names that should be defined above in the specification file. State operators of tree automata corresponding to the names are imported in the current approximation and do not need to be redefined.

**States** is followed by a sequence of state operators as for the **States** keyword of the tree automata description, see section 3.6.2 for details.

**Equations** is followed by a sequence of *equations* where an *equation* is a pair of terms separated by the character  $=$ . The terms on both sides of the equation can be built over the alphabet and the variables of the specification and the state operators, i.e. terms may contain symbols, variables and states.

**Rules** is followed by a sequence of *normalization rules*. The general form of a normalization rule is:

$$[s \rightarrow x] \rightarrow [l1 \rightarrow r1 \dots ln \rightarrow rn]$$

where  $s, l1, \dots, ln$  are terms that may contain symbols, variables and states, and  $x, r1, \dots, rn$  are either states or variables. If  $ri$  is a variable then it is equal to  $x$ . If  $ri$  is a state built with a state operator of arity greater to zero then any variable  $y$  of  $ri$  occurs in  $s$  or is equal to  $x$ . See section 4.2.2 for use of those rules.

## 4 Timbuk reference manual

In this part, we assume that the reader is familiar with term rewriting systems [1], tree automata [3] and the tree automata completion process described in [7]. Given a term rewriting system  $\mathcal{R}$ ,  $s \rightarrow_{\mathcal{R}} t$  will denote that  $s$  can be rewritten by  $\mathcal{R}$  in one step into  $t$ . Similarly,  $s \rightarrow_{\mathcal{R}}^* t$  will denote that  $s$  can be rewritten by  $\mathcal{R}$  in zero or more steps into  $t$ . The set of  $\mathcal{R}$ -descendants of a set of ground terms  $E \subseteq \mathcal{T}(\mathcal{F})$  is  $\mathcal{R}^*(E) = \{t \in \mathcal{T}(\mathcal{F}) \mid \exists s \in E \text{ s.t. } s \rightarrow_{\mathcal{R}}^* t\}$ .

Given a tree automaton  $\mathcal{A}$ , the rewriting relation induced by the transitions of  $\mathcal{A}$  is denoted by  $\rightarrow_{\mathcal{A}}$ . The tree language recognized by  $\mathcal{A}$  denoted by  $\mathcal{L}(\mathcal{A})$  is  $\mathcal{L}(\mathcal{A}) = \{t \in \mathcal{T}(\mathcal{F}) \mid t \rightarrow_{\mathcal{A}}^* q \text{ s.t. } q \text{ is a final state}\}$ .

### 4.1 Running Timbuk

To start a completion process and launch the Timbuk tool over a Timbuk specification file called `example.txt`, simply type:

```
Timbuk example.txt
```

in a command window. The specification file should at least contain one tree automaton and one term rewriting system. Depending on the way you obtained Timbuk, you may not be able to directly use 'timbuk' as a standalone command and you may need to type `ocamlrun timbuk example.txt` instead. Please refer to the README file of the distribution for details on how to run the Timbuk library tools. If launching Timbuk succeeds, then Timbuk reads the given specification file and starts a tree automata completion with

- the first term rewriting of the specification (let us denote it by  $\mathcal{R}$  in the following)
- the first tree automaton of the specification (let us denote it by  $\mathcal{A}$  in the following)
- the first approximation (if it exists) of the specification

The remaining tree automata of the specification file are also read and stored by Timbuk for (later) verification purpose. The general completion process [7] works by incremental completion of automaton  $\mathcal{A}$  into  $\mathcal{A}_1, \mathcal{A}_2, \dots$ . Each step from  $\mathcal{A}_i$  to  $\mathcal{A}_{i+1}$  is called the  $i + 1$ -th completion step. For obtaining  $\mathcal{A}_{i+1}$  from  $\mathcal{A}_i$ , one searches for every term  $s \in \mathcal{L}(\mathcal{A}_i)$  such that  $s \rightarrow_{\mathcal{R}} t$  and  $t \notin \mathcal{L}(\mathcal{A}_i)$ . Then  $\mathcal{A}_{i+1}$  is built from  $\mathcal{A}_i$  by adding transitions to  $\mathcal{A}_i$  such that  $\mathcal{L}(\mathcal{A}_{i+1}) \supseteq \mathcal{L}(\mathcal{A}_i)$  and  $t \in \mathcal{L}(\mathcal{A}_{i+1})$  for every term  $t$  such that  $s \in \mathcal{L}(\mathcal{A}_i)$ ,  $s \rightarrow_{\mathcal{R}} t$  and  $t \notin \mathcal{L}(\mathcal{A}_i)$ .

When completion converges, completion reaches a fixpoint  $\mathcal{A}_k$  such that for every term  $s \in \mathcal{L}(\mathcal{A}_k)$  such that  $s \rightarrow_{\mathcal{R}} t$  then  $t \in \mathcal{L}(\mathcal{A}_k)$ . Hence,  $\mathcal{L}(\mathcal{A}_k)$  is an over-approximation of  $\mathcal{R}$ -descendants of  $\mathcal{L}(\mathcal{A})$ , i.e.  $\mathcal{L}(\mathcal{A}_k) \supseteq \mathcal{R}^*(\mathcal{L}(\mathcal{A}))$ . In other words,  $\mathcal{A}_k$  recognizes a superset of terms reachable by

$\mathcal{R}$  from terms of  $\mathcal{L}(\mathcal{A})$ . In the next section, we present a collection of approximation techniques provided by Timbuk to make completion converge. For non left-linear TRS (i.e. TRS having at least two occurrences of the same variable in the left-hand side), for  $\mathcal{A}_k$  to be an over-approximation of  $\mathcal{R}^*(\mathcal{L}(\mathcal{A}))$  it is also necessary to check the left-linearity condition.

Note that for some particular cases of TRS and initial automaton  $\mathcal{A}$  the fixpoint is not only an over-approximation but it is exactly the set  $\mathcal{R}^*(\mathcal{L}(\mathcal{A}))$ . Those exact classes and associated specific completion strategy will be detailed in section 4.4.1.

## 4.2 Timbuk normalization and approximation tools

In this section, we present various techniques implemented in Timbuk to force completion to converge, i.e. to build an over-approximation of  $\mathcal{R}^*(\mathcal{L}(\mathcal{A}))$  (the set of reachable terms) when it cannot be computed exactly. In a typical  $i$ -th completion step, recall that each rule  $l \rightarrow r$  of  $\mathcal{R}$  is used to build critical pairs, i.e. find a  $\mathcal{Q}$ -substitution  $\sigma$  and a state  $q$  of  $\mathcal{A}_i$  such that  $l\sigma \rightarrow_{\mathcal{A}_i}^* q$  and  $r\sigma \not\rightarrow_{\mathcal{A}_i}^* q$ . Then, the transition  $r\sigma \rightarrow q$  is added to  $\mathcal{A}$  to build  $\mathcal{A}_{i+1}$ . But, the transition  $r\sigma \rightarrow q$  may not be normalized, i.e.  $r\sigma$  is a state (hence  $r\sigma \rightarrow q$  is an epsilon transition) or  $r\sigma = f(t_1, \dots, t_n)$  and there exists at least one  $j \in \{1, \dots, n\}$  such that  $t_j$  is not a state. If the transition is not normalized then it has to be normalized before being added to the tree automaton. Normalizing epsilon transitions is easy and does not make completion diverge: for a transition of the form  $q_1 \rightarrow q_2$  it is enough to add the set of transitions  $\{c \rightarrow q_2 \mid c \rightarrow q_1 \in \mathcal{A}_i\}$  to  $\mathcal{A}_i$ . For normalizing a transition  $f(t_1, \dots, t_n) \rightarrow q$  where  $t_j$  is not a state, it is necessary to introduce a state, say  $q_j$  and replace the transition  $f(t_1, \dots, t_n) \rightarrow q$  by the two transitions:  $f(t_1, \dots, q_j, \dots, t_n) \rightarrow q$  and  $t_j \rightarrow q_j$ . This process has to be continued until every transition is normalized. Depending on the choice of states used for normalization (for instance state  $q_j, \dots$ ) the addition of the new transition will be exact or approximated. For instance, if the state  $q_j$  is a new state (i.e. not occurring in  $\mathcal{A}_i$ ), then adding  $f(t_1, \dots, t_n) \rightarrow q$  or the two transitions:  $f(t_1, \dots, q_j, \dots, t_n) \rightarrow q$  and  $t_j \rightarrow q_j$  is similar. On the opposite, if we choose  $q_j = q$  then  $f(t_1, \dots, q, \dots, t_n) \rightarrow q$  and  $t_j \rightarrow q$  *over-approximate*  $f(t_1, \dots, t_n) \rightarrow q$ . Indeed, with the pair of transitions  $f(t_1, \dots, q, \dots, t_n) \rightarrow q$  and  $t_j \rightarrow q$ , one can build the transition  $f(t_1, \dots, t_n) \rightarrow q$  but also an infinite set of transition of the form  $f(t_1, \dots, f(t_1, \dots, t_n), \dots, t_n) \rightarrow q$  and so on.

Since, approximations are determined by normalization choices, the central tools used in Timbuk for building approximations are techniques for guiding the choice of states used in the normalization process.

### 4.2.1 Priority transitions

The priority transitions are a set of deterministic tree automata transitions used to simplify a new transition to be added by bottom-up rewriting. Let  $f(g(a)) \rightarrow q$  be the new transition to add and normalize. If the set of priority transitions contains  $a \rightarrow q_1$  then  $f(g(a)) \rightarrow q$  will be normalized into  $f(g(q_1)) \rightarrow q$  and  $a \rightarrow q_1$ . If the set of priority transitions does not contain a transition for simplifying  $g(q_1)$  then normalizing cannot go further with priority transitions.

Priority transitions can either be defined in the specification files (see 'Prior' field of tree automata explicit definition in section 3.6.2), interactively during completion (see `manual` and `manual_conf` strategy operators in section 4.4.1) or automatically with the 'auto\_prior' normalization strategy (see 'auto\_prior' strategy operator in section 4.4.1).

Any set of priority transitions can be expressed using normalization rules (defined in the next section) but priority transitions remain a syntactic facility avoiding the repetition of some



transitions that are already part of the automaton. Indeed, since priority transitions are generally transitions of the initial automaton, the 'Prior' field of the tree automaton permits to define them once as tree automata transitions and transitions for approximation.

#### 4.2.2 Normalization rules

Normalization rules (or norm rules) are a sequence of rules of the form:

$$[s \rightarrow x] \rightarrow [l_1 \rightarrow r_1 \dots l_n \rightarrow r_n]$$

where  $s, l_1, \dots, l_n$  are terms that may contain symbols, variables and states, and  $x, r_1, \dots, r_n$  are either states or variables. If  $r_i$  is a variable then it is equal to  $x$ . If  $r_i$  is a state built with a state operator of arity greater to zero then any variable  $y$  of  $r_i$  occurs in  $s$  or is equal to  $x$ . To normalize a transition of the form  $t \rightarrow q'$ , we match the pattern  $s$  on  $t$  and  $x$  on  $q'$ , obtain a given substitution  $\sigma$  and then we normalize  $t$  with the rewrite system  $\{l_1\sigma \rightarrow r_1\sigma, \dots, l_n\sigma \rightarrow r_n\sigma\}$  where  $r_1\sigma, \dots, r_n\sigma$  are necessarily states. For example, normalizing a transition  $f(h(q_1), g(q_2)) \rightarrow q_3$  with approximation rule  $[f(x, g(y)) \rightarrow z] \rightarrow [g(u) \rightarrow z]$  will give a substitution  $\sigma = \{x \mapsto h(q_1), y \mapsto q_2, z \mapsto q_3\}$ , an instantiated set of rewrite rules  $[g(u) \rightarrow q_3]$ . Thus,  $f(h(q_1), g(q_2)) \rightarrow q_3$  will be normalized into a normalized transition  $g(q_2) \rightarrow q_3$  and a partially normalized transition  $f(h(q_1), q_3) \rightarrow q_3$ .

Normalization rules are used in the order of the sequence: if a normalization rule does not apply then the following rule is used and so on. When a normalization rule succeeds in normalizing a transition (even partially) then the sequence is taken back from the beginning and the normalization process continues on partially normalized transitions.

Note that in dynamic mode (see section 4.4 for details about Timbuk completion modes) the syntax for normalization rules has been extended so that it is also possible to achieve the pattern matching under state operators. For instance, it is now possible to define a normalization rule of the form:

$$[encr(pubkey(q(x)), m) \rightarrow qstore] \rightarrow [m \rightarrow q(secret(x))]$$

where  $x$  and  $m$  are variables,  $q$  is here a state operator of arity 1 and *secret* is either a symbol or a state operator of arity 1. This rule will thus normalize transitions  $encr(pubkey(q(A)), cons(q_1, q_2)) \rightarrow q$  and  $encr(pubkey(q(B)), cons(q_3, q_4)) \rightarrow q$  respectively in  $encr(pubkey(q(A)), q(secret(A))) \rightarrow q$   $cons(q_1, q_2) \rightarrow q(secret(A))$  and  $encr(pubkey(q(B)), q(secret(B))) \rightarrow q$   $cons(q_3, q_4) \rightarrow q(secret(B))$ . The only syntactic constraint on those normalization rules is the following: for every rule of the form

$$[s \rightarrow x] \rightarrow [l_1 \rightarrow r_1 \dots l_n \rightarrow r_n]$$

either  $r_i$  is a state constant, either it is equal to  $x$ , or it is a term of the form  $q(t_1, \dots, t_n)$  where  $q$  is a state operator and variables of  $t_1, \dots, t_n$  are either equal to  $x$  or contained in  $s$ .

Normalisation rules can be defined both in the specification file (see section 3.7) or during completion in dynamic mode using the `manual_norm` strategy operator (see section 4.4.1) or the (g) Timbuk command (see section 4.3).

#### 4.2.3 Merging rules

Merging rules are a sequence of epsilon transitions of the form  $q_1 \rightarrow q_2$  between states  $q_1$  and  $q_2$ . The meaning of such a rule is that states (and thus corresponding recognized languages)  $q_1$  and  $q_2$

should be merged together. Applying a merging rule  $q_1 \rightarrow q_2$  on an automaton  $\mathcal{A}'$  simply consists in rewriting all the state labels of the tree automaton such that  $q_1$  is replaced everywhere by  $q_2$ . The resulting automaton  $\mathcal{A}$  is always such that  $\mathcal{L}(\mathcal{A}) \subseteq \mathcal{L}(\mathcal{A}')$ .

Unlike preceding tools, merging rules can only be given interactively and are applied after that transitions have been normalized. A typical use of merging rules is to normalize automatically new transitions by new states (see strategy operator 'auto' in section 4.4.1 for details) and then give interactively the merging rules for achieving the approximation. Note that merging rules can also be built graphically using Tabi see section 2.3.3.

#### 4.2.4 Approximation equations

Approximation equations are a sequence of equations of the form  $s = t$  where  $s$  and  $t$  are terms built on symbols, states and variables. The meaning of such a rule is that every terms matching this equation should be merged together. In practice, terms  $s$  and  $t$  are matched over the automaton  $\mathcal{A}_i$  of the current completion step and for every  $\mathcal{Q}$ -substitution  $\sigma$  and for every states  $q_1, q_2$  such that  $s\sigma \rightarrow_{\mathcal{A}_i}^* q_1$  and  $t\sigma \rightarrow_{\mathcal{A}_i}^* q_2$ , a merging rule  $q_1 \rightarrow q_2$  is produced and applied.

Like merging rules, equations are applied after that transitions have been normalized. Approximation equations can be defined both in the specification file (see section 3.7) or during completion in dynamic mode (see section 2.1.7).

### 4.3 Timbuk commands

When starting Timbuk on a valid specification file, the user is proposed the following menu:

Completion step: 0

Do you want to:

```
(c)omplete one step (use Ctrl-C to interrupt if necessary)
complete (a)ll steps (use Ctrl-C to interrupt if necessary)
(m)erge some states
(s)ee current automaton
(b)rowse current automaton with Tabi
(d)isplay the term rewriting system
(i)ntersection with verif automata
intersection with (o)ther verif automata on disk
search for a (p)attern in the automaton
(v)erify linearity condition on current automaton
(w)rite current automaton, TRS and approximation to disk
(f)orget old completion steps
(e)quation approximation in gamma
(g)amma normalisation rules
(det)erminise current automaton
(u)ndo last step
(q)uit completion
```

The first line gives the current completion step. Initially the completion step number is 0. Then the user have to type one of the following command:

- c performs one completion step. The completion can be stopped using a CTRL-C key combination.
- a performs all possible completion steps. If completion converges then this command is going to stop. Otherwise, the user may interrupt it using a CTRL-C key combination.
- m ask for a sequence of merging rules over states of the tree automaton. A merging rule is a pair of states separated by  $\rightarrow$ . The sequence has to be terminated by a dot '.' symbol. A merging rule of the form  $q1 \rightarrow q2$  will rename ever occurrence of the state  $q1$  by the state  $q2$ . The language recognized by the renamed automaton is always an over-approximation of the language recognized by the initial one.
- s displays the completed automaton at the current completion step.
- b browse the completed automaton at the current completion step using Tabi, if it has been installed. During browsing, merging rules can also be defined in a more graphical and more intuitive way (see section 2.3.3 for an example). If such rules are defined and applied under Tabi, then merging is performed when leaving Tabi (see Tabi documentation in section 6).
- d displays the term rewriting system used for completion.
- i computes intersection between the completed automaton at the current completion step and automata that were in the same specification file.
- o computes intersection between the completed automaton at the current completion step and some other tree automata stored in an other file.
- p searches for a given pattern in the completed automaton (say  $\mathcal{A}_j$ ) at the  $j$ -th completion step. A pattern  $p$  is a term built over symbols of the alphabet, variables and states of the current automaton. The result for pattern matching over the tree automaton is a sequence of solutions. Each solution consists of a state  $q$  and a set of  $\mathcal{Q}$ -substitutions  $\sigma_1, \dots, \sigma_n \in \Sigma(\mathcal{Q}, \mathcal{X})$  such that for all  $i = 1, \dots, n : p\sigma_i \rightarrow_{\mathcal{A}_j} q$ .
- v verify the left-linearity condition. For non left-linear TRS, the final completed automaton is an over approximation only if left-linearity condition is satisfied (see section 2.1.5 for an example and see [7] for theoretical details about left-linearity condition).
- w writes the current automaton, TRS, approximation and automaton list used for intersection to disk in Timbuk specification file syntax. This command also writes the initial automaton in the specification.
- f forgets the previous completion step. This is useful, when completion steps are getting bigger and bigger.
- e is used to consult and add approximation equations to the gamma approximation function. See section 3.7 for details about the syntax.
- g is used to consult and add normalization rules to the gamma approximation function. See section 3.7 for details about the syntax.
- det determinizes the current completed automaton.
- u undoes the last completion step.
- q quit completion

## 4.4 Timbuk modes and command line options

When executing Timbuk the user can use several command line options which depend on the major running mode of Timbuk. The two major modes for running Timbuk are *dynamic* and *static* modes. There is also a variant of the static mode which is called *forced static* or *fstatic* for short. The dynamic mode is the default completion mode of Timbuk. It can easily be parametrized by approximation functions, equations and strategies. The static mode is more constrained but permits to achieve a pre-compilation of the completion and is thus more efficient.

Some options do not depend on the Timbuk running mode:

- o followed by a file name prints all Timbuk output to that file.
- f followed by a file name reads all Timbuk commands input in that file.
- noapprox don't care of the approximations defined in the specification file.
- approx followed by an approximation name, starts the Timbuk completion process with the approximation denoted by the given name rather than the first of the specification.

All the other command line options depend on the used timbuk running mode.

### 4.4.1 Dynamic completion mode

In dynamic mode (default mode), the priority transitions, the normalization rules and the approximation equations can be given initially through a specification file or can be added during completion process. Approximation strategy can also be parametrized. Here are the dynamic mode command line options:

- dynamic used to toggle the dynamic mode on (default mode)
- strat followed by a sequence of normalization strategy operators (see below).

The `--strat` option permits to give explicitly the strategy to use for normalizing the new transition. Then, each new transition produced by the completion is normalized successively using the normalization strategy operators given in the strategy until every transition is normalized. If the end of the strategy operator sequence is reached and there remain some transitions to normalize then the normalization process continues and the strategy sequence is reinitialized from the beginning. The default Timbuk strategy in dynamic mode corresponds to the strategy operators sequence `prior norm_rules manual_norm_conf auto_conf`. Here are the definitions of the basic normalization strategy operators. Some of these operators always succeed (they always manage to normalize any set of transitions) and thus should be placed at the end of the sequence.

`exact` for exact normalization. This normalization strategy operator always succeeds. The automata  $\mathcal{A}_1, \mathcal{A}_2, \dots$  produced by completion steps recognize only terms  $\mathcal{R}$ -reachable from  $\mathcal{L}(\mathcal{A})$ , i.e. the automaton  $\mathcal{A}_i$  obtained after the  $i$ -th completion step is not an over-approximation (but an under-approximation) if:

- $\mathcal{R}$  is linear, or
- $\mathcal{R}$  is right-linear and  $\mathcal{R}$  and  $\mathcal{A}_i$  satisfy the left-linearity condition, or
- every state of  $\mathcal{A}$  recognizes at most one term<sup>4</sup> and  $\mathcal{R}$  is left-linear, or

---

<sup>4</sup>Note that this is trivially the case if  $\mathcal{A}$  is defined using the 'Set' keyword, see section 3.6.1.

- every state of  $\mathcal{A}$  recognizes at most one term and  $\mathcal{A}_i$  and  $\mathcal{R}$  satisfy the left-linearity condition.

Hence, for those classes, if completion converges on a fixpoint  $\mathcal{A}_k$  then  $\mathcal{L}(\mathcal{A}_k) = \mathcal{R}^*(\mathcal{L}(\mathcal{A}))$ . Furthermore, completion is guaranteed to converge on some known decidable classes:

- $\mathcal{R}$  is either a ground TRS [5, 2].
- a right-linear and monadic TRS [15], i.e. right-hand sides of the rules are either variables or terms of the form  $f(x_1, \dots, x_n)$  where  $f \in \mathcal{F}$  and  $x_1, \dots, x_n$  are variables.
- a linear and semi-monadic TRS [4], i.e. rules are linear and their right-hand sides are of the form  $f(t_1, \dots, t_n)$  where  $f \in \mathcal{F}$  and  $\forall i = 1, \dots, n, t_i$  is either a variable or a ground term.
- a “decreasing” TRS [11], where “decreasing” means that every right-hand side is either a variable, or a term  $f(t_1, \dots, t_n)$  where  $f \in \mathcal{F}$ ,  $ar(f) = n$ , and  $\forall i = 1, \dots, n, t_i$  is a variable, a ground term, or a term whose variables do not occur in the left-hand side.
- constructor-based rewrite systems [14] where the alphabet  $\mathcal{F}$  is separated into a set of *defined symbols*  $\mathcal{D} = \{f \mid \exists l \rightarrow r \in \mathcal{R} \text{ s.t. } \text{Root}(l) = f\}$  and constructor symbols  $\mathcal{C} = \mathcal{F} \setminus \mathcal{D}$ . The restriction on  $\mathcal{L}(\mathcal{A})$  is the following:  $\mathcal{L}(\mathcal{A})$  is the set of ground constructor instances of a linear term  $t$ , i.e.  $\mathcal{L}(\mathcal{A}) = \{t\sigma\}$  where  $t \in \mathcal{T}(\mathcal{F}, \mathcal{X})$  is linear and  $\sigma : \mathcal{X} \mapsto \mathcal{T}(\mathcal{C})$ . The restrictions on  $\mathcal{R}$  are the following: for each rule  $l \rightarrow r$ 
  1.  $r$  is linear
  2. for each position  $p \in \text{Pos}_{\mathcal{F}}(r)$  such that  $r|_p = f(t_1, \dots, t_n)$  and  $f \in \mathcal{D}$  we have that for all  $i = 1 \dots n, t_i$  is a variable or a ground term
  3. there is no nested function symbols in  $r$

**prior** for normalization with priority transitions. See section 4.2.1 for details.

**norm\_rules** for normalization with normalization rules. See section 4.2.2 for details.

**auto** automatically normalizes transitions with new states. This operator always succeeds.

**auto\_conf** same as **auto** but asks for confirmation first.

**auto\_prior** automatically normalizes transitions with new states and stores the new transitions as priority transitions. This operator always succeeds. Note however that if **prior** is not placed before **auto\_prior** in the strategy then the benefit of adding new priority transitions will be lost and **auto\_prior** will normalize every transitions with new states and thus will behave as **auto**.

**auto\_prior\_conf** same as **auto\_prior** but asks for confirmation first.

**manual\_norm** ask the user to give explicitly some normalization rules. Note that if **norm\_rules** is not placed before **manual\_norm** in the strategy then **manual\_norm** has no effect since normalization rules may be added but never triggered.

**manual\_norm\_conf** same as **manual** but asks for confirmation first.

**manual** ask the user to give explicitly some transitions to normalize the transitions. The user may also give some (normalized) priority transitions.

**manual\_conf** same as **manual** but asks for confirmation first.

### 4.4.2 Static completion mode

In static mode (and in its variant called *fstatic* for *forced static*), only priority transitions and normalization rules given in the specification file are used. In fact, the normalization strategy in static mode is fixed and corresponds to the sequence `prior norm_rules`. Moreover, in static mode, priority transitions and normalization rules should define an approximation function that is complete with regards to the right-hand sides of the rewrite rules. In other words, every possible new transition produced during completion by the instantiation of the right-hand side of a rewrite rule must be normalized using the prior transitions and the normalization rules given by the user in the specification file. If this is not the case then Timbuk fails and returns the transition that cannot be normalized using the user's approximation function. Note however that when Timbuk's static completeness is too restrictive (your approximation is complete but Timbuk has not detected it) it is possible to simply extend it by some additional rules (see section 2.1.6). Furthermore, in *fstatic* mode, if the approximation is not complete then it is automatically expanded for normalizing remaining transitions (not normalized using user's rules) with a specific state labeled by `#qstatic#`.

Apart from the common command line options described at the beginning of this section, the only static mode options are:

`--static` to activate the static compilation of matching and normalization (needs a complete set of prior and norm rules).

`--fstatic` to activate the static compilation of matching and normalization. If the set of prior and norm rules is not complete, a transition not covered by the rules is normalized using a single new state `#qstatic#`.

Note that merging rules and approximation equations may be applied on every completed automaton in static mode, but approximation equations are not taken into account for approximation pre-compilation.

## 5 Taml reference manual

Taml is an Ocaml toplevel equipped with Timbuk functions over terms, term rewriting systems and tree automata.

### 5.1 Running Taml

To start Taml, simply type:

```
taml
```

in a command window. Depending on the way you obtained Taml, you may not be able to directly use 'taml' as a standalone command and you may need to type `ocamlrun taml` instead. Please refer to the `README` file of the distribution for details on how to run the Timbuk library tools. Note that all the directives of Ocaml toplevel can be used in this particular one as `#use`. For instance, it is possible to load the tutorial file called `tutorial.ml` by typing the following directive in Taml toplevel:

```
#use "tutorial.ml";;
```

IMPORTANT: Taml has to be run in the same directory as the `.cmo` files and the `.ocamlinit` file of the Timbuk library.

## 5.2 Basic Taml functions

First, here are all the defined functions. A more precise description is given in the following. Note that Ocaml labels are only used here for clarity of the documentation and cannot be used at Ocaml level. For all the functions building objects (like alphabets, terms, term rewriting systems, tree automata, etc) from a string, the input syntax of the string should respect the timbuk syntax for any of this object which is described in section 3. The file `tutorial.ml` also contains several examples of this syntax. See section 2.2 for the Taml tutorial.

```
val browse : Automaton.t → unit
val alphabet : string → Alphabet.t
val varset : string → Variable_set.t
val term : Alphabet.t → Variable_set.t → string → Term.t
val state : string → Automaton.state
val tree_state : Alphabet.t → Alphabet.t → string → Term.t
val trs : Rewrite.alphabet → Rewrite.variable_set → string → Rewrite.t
val automaton : Automaton.alphabet → string → Automaton.t
val finite_set : Automaton.alphabet → string → Automaton.t
val inter : Automaton.t → Automaton.t → Automaton.t
val union : Automaton.t → Automaton.t → Automaton.t
val inverse : Automaton.t → Automaton.t
val subtract : Automaton.t → Automaton.t → Automaton.t
val is_included : Automaton.t → Automaton.t → bool
val is_language_empty : Automaton.t → bool
val is_finite : Automaton.t → bool
val run : t : Automaton.term → q : Automaton.state → a : Automaton.t → bool
val determinise : Automaton.t → Automaton.t
val irr : a : Automaton.alphabet → r : Automaton.transition_table → Automaton.t
val clean : Automaton.t → Automaton.t
val simplify : Automaton.t → Automaton.t
val save : Automaton.t → aut_name : string → file_name : string → unit
val read_alphabet : string → Alphabet.t
val read_spec : string → Specification.spec
val read_automaton : string → string → Automaton.t
val read_automaton_list : string → Automaton.t list
val read_trs : string → string → TRS.t
val read_trs_list : string → TRS.t list
val help : unit → unit
```

Here is for each of these functions a more detailed description.

### 1. Alphabets

To build an alphabet from a string

```
val alphabet : (s : string) → Alphabet.t
```

To read an alphabet in a Timbuk specification file.

```
val read_alphabet : (s : string) → Alphabet.t
```

## 2. Variable sets

To build a variable set from a string.

`val varset : (s : string) → Variable_set.t`

## 3. Terms

To build a term on alphabet  $a$  and variable set  $v$  from a string  $s$ .

`val term : (a : Alphabet.t) (v : Variable_set.t) (s : string) → Term.t`

## 4. Term rewriting systems

To build a TRS on alphabet  $a$ , variable set  $v$  and from a string  $s$ .

`val trs : (a : Alphabet.t) (v : Variable_set.t) (s : string) → Rewrite.t`

To read a TRS of name  $n$  in a specification file  $f$ .

`val read_trs : (n : string) → (f : string) → Rewrite.t`

To read all the TRS in specification file  $f$ .

`val read_trs_list : (f : string) → Rewrite.t list`

## 5. Tree automata

To build a state from string.

`val state : (s : string) → Automaton.state`

To build a (tree) state on alphabet  $a$ , state operators  $sop$  and from a string  $s$ . A tree state is a state built on state operators of arity greater than 0. For instance, if  $p$  is a state operator of arity 2 and  $q$  is a state operator of arity 0, then  $p(q, q)$  is a tree state.

`val tree_state : (a : Alphabet.t) (sop : Alphabet.t) (s : string) → Automaton.state`

To build an automaton on alphabet  $a$  from a string  $s$ .

`val automaton : (a : Alphabet.t) (s : string) → Automaton.t`

To build an automaton on alphabet  $a$  from a string  $s$  representing the finite of terms to be recognized by the automaton.

`val finite_set : (a : Alphabet.t) (s : string) → Automaton.t`

To read an automaton of name  $n$  in a specification file  $f$

`val read_automaton : (n : string) → (f : string) → Automaton.t`

To read all the automaton in specification file named  $f$

`val read_automaton_list : (f : string) → Automaton.t list`

To browse automaton  $a$  (if Tabi is installed).



`val browse : (a : Automaton.t) → unit`

To build the intersection automaton between *a1* and *a2*. Sets of states are not explicitly built. To obtain them explicitly use cleaning afterwards.

`val inter : (a1 : Automaton.t) → (a2 : Automaton.t) → Automaton.t`

To build the union automaton for *a1* and *a2*.

`val union : (a1 : Automaton.t) → (a2 : Automaton.t) → Automaton.t`

The complement operation.

`val inverse : Automaton.t → Automaton.t`

To build an automaton recognizing  $L(a1) - L(a2)$ .

`val subtract : (a1 : Automaton.t) → (a2 : Automaton.t) → Automaton.t`

Is  $L(a1)$  included in  $L(a2)$ ?

`val is_included : (a1 : Automaton.t) → (a2 : Automaton.t) → bool`

Is  $L(a)$  empty?

`val is_language_empty : (a : Automaton.t) → bool`

Is  $L(a)$  finite?

`val is_finite : (a : Automaton.t) → bool`

Is *t* recognized into state *q* in *a*?

`val run : (t : Term.t) → (q : State.t) → (a : Automaton.t) → bool`

Determinisation of a tree automaton.

`val determinise : Automaton.t → Automaton.t`

To build a tree automaton recognising the set of terms irreducible by TRS *t*.

`val irr : (a : Alphabet.t) → (t : Rewrite.t) → Automaton.t`

Accessibility cleaning followed by utility cleaning for a tree automaton.

`val clean : Automaton.t → Automaton.t`

Accessibility cleaning followed by utility cleaning and renumbering.

`val simplify : Automaton.t → Automaton.t`

To save automaton *a* with name *n* in file named *f*.

`val save : (a : Automaton.t) → (n : string) → (f : string) → unit`

## 6. Specifications

To read a full Timbuk specification in file of name *s*

`val read_spec : (s : string) → Specification.t`

### 5.3 Using all Timbuk library functions through Taml

The functions proposed by Taml at toplevel are only a part of all the Timbuk library functions. To have an access to the other functions dispatched in the Timbuk modules, you can call them directly (if the module has been opened first, using the `open` Ocaml keyword) or use the usual prefixed notation. For instance, to call the `left_inner_norm` function of the `Rewrite` module, used for normalizing a term with a term rewriting system using leftmost innermost strategy, one can access this function with the function name prefixed by the module name:

```
Rewrite.left_inner_norm
```

For details on the modules and offered functions, have a look to section 7

## 6 Tabi reference manual

The aim of Tabi is to ease tree automata understanding. When tree automata are getting bigger and bigger, Tabi helps in figuring out what is the recognized language. Tabi stands for *Tree Automata Browsing Interface*: Starting from any state  $q$  of an automaton, Tabi provides an interactive and graphical way to build some of the terms recognized by  $q$  in the automaton. Tabi can represent terms in the usual linear way (with parenthesis and comas) way as well as trees, or even in a mixture of both representations (See Figure 3). Recognized terms can be built interactively by state expansion and transition selection or automatically using a randomized representative generator.

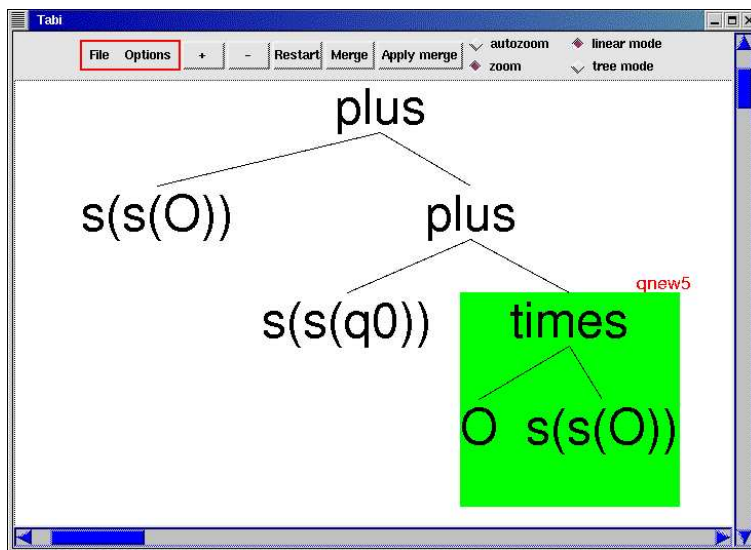


Figure 3: Tabi graphical user interface

Tabi can be used as an independent program or as a graphical interface for the Timbuk and Taml tools. When using Tabi from Timbuk, Tabi also permits to build merging rules over terms that are built. Tabi has been developed with Labltk (Ocaml with Tk functions) in collaboration with a group of students in 4th year of Computer Science of Rennes University (see `README` file for credits)

*Note on Automaton loading:* when using Tabi from Timbuk (resp. Taml), Tabi starts on the current completed automaton (resp. the automaton parameter of the `browse` function). When using Tabi

as a standalone program, one has to open a Timbuk specification file where the *first* automaton is read. When an automaton is loaded the **Start** symbol is displayed.

## 6.1 Mouse actions

**Moving the mouse pointer** over a term or a state highlights it in green. If it is a term, then the state recognizing this term is shown in red. See Figure 3 for an example with term  $times(O, s(s(O)))$  and state  $qnew5$ .

**Left click** over a state  $q$  *unfolds*  $q$ , i.e. propose configurations or terms to replace  $q$ . Clicking on  $q$  opens a window containing a list of possible configuration leading to  $q$  as well as a button **choose random**. A left click on a configuration of the list replaces  $q$  by the chosen configuration. Clicking on **choose random** opens a new window containing a list of randomly generated ground terms recognized by  $q$ . Clicking on one of these terms replaces  $q$  by the chosen term. Note that if  $q$  recognizes an empty language or if the depth or time for random search is not sufficient to produce random terms, an error message is produced. See Options menu in section 6.4 for changing depth or time for random term generation.

**Middle click** over a term  $t$  *folds* it, i.e. replace it by the state recognizing  $t$ .

**CTRL + Left click** over a term  $t$  changes the whole graphical representation of  $t$  from linear mode to tree mode. This operation does not affect the term embedding  $t$ .

**SHIFT + Left click** over a term  $t$  changes the whole graphical representation of  $t$  from tree mode to linear mode. This operation does not affect the term embedding  $t$ .

**Right click** over a term  $t$  switches from linear and tree mode on the top of  $t$ . This operation does not affect the term embedding  $t$  nor subterms of  $t$ .

**CTRL + Right click** over a term  $t$  draws a blue rectangle over  $t$  and select it for merging. After selecting two terms  $t_1$  and  $t_2$  for merging, it is possible to press on button **Merge** in order to add a merging rule  $q_1 \rightarrow q_2$  where  $q_1$  and  $q_2$  recognize respectively  $t_1$  and  $t_2$ . Note that merging rules can only be used if Tabi has been launched from Timbuk.

## 6.2 Buttons

**+ and - Buttons** are used to increase/decrease the zoom factor for displaying the terms.

**Restart** permits to restart the automata browsing from the beginning, i.e. from the **Start** symbol. This is useful when the automaton has several final states to restart browsing from a different final state.

**Merge** builds a merging rule from to terms selected for merging (see CTRL + Right Click action in section 6.1).

**Apply merge** quits Tabi and apply the list of merging rules defined by the user to the current automaton (Only if using Tabi from Timbuk).

**Autozoom/Zoom Buttons** switches between automatic and manual zoom. When Autozoom is selected Tabi automatically changes the zoom factor in order to keep the whole term visible in the window. Note that when the zoom factor is getting to small Tabi automatically switches

to manual zoom. On the opposite, with the manual zoom it is possible to focus on a smaller part of the term.

**Linear/Tree mode Buttons** switches between Linear and Tree mode (default modes) for displaying terms obtained by unfolding.

### 6.3 File menu

**Open** browse in current directory for a Timbuk specification file containing a tree automaton (See section 3 for precise syntax). Note that only the *first* automaton of the specification file is taken into account.

**Print** produce a file `tabi.ps` containing a postscript version of the term displayed in the Tabi window.

**Exit** quits Tabi (without applying merging rules).

### 6.4 Options menu

**Undo** Undo last folding or unfolding.

**Redo** Redo last folding or unfolding.

**Browsing style** switches between *Final states* or *All states* browsing style. In *Final states* style (default), when left-clicking on the **Start** symbol one is only proposed the final states of the automaton, whereas in *All states* style all the states of the automaton are proposed for browsing.

**See merging rules** displays the merging rules already defined.

**Random max depth** changes the upper bound for depth of terms built by random representatives generation.

**Random max time** changes the upper bound on time for random representatives generation.

**Random max term number** changes the upper bound on the number of representatives to be randomly generated.

**Show history** opens a window with an ordered list of the terms built during the previous steps. When clicking on any term of the list, the selected term becomes the current term.

**Help** displays a short help on the mouse actions.

## 7 How to use Ocaml functions of the Timbuk library?

Since this software is a modular library, we wanted to have a separated documentation for each module. That is why we chose to generate this documentation using `ocamlweb` [8]. In the following you will find one section for each main module: tree automaton, term, term rewriting systems, etc. To see an example showing how to call those functions from Taml, have a look to section 2.2. To see how to import modules and call those functions from some other Ocaml code see Taml main

Ocaml file: `taml.ml` or Timbuk mail Ocaml file: `main.ml` in the source distribution. Note that labels in function declarations are only used for clarity of the documentation and cannot be used in functions calls as in the Ocaml syntax extension.

## Interface for module Automaton

7. This is the interface for bottom-up tree automata. A bottom-up tree automata is usually defined as a tuple:  $\langle \mathcal{F}, \mathcal{Q}, \mathcal{Q}_f, \Delta \rangle$  where  $\mathcal{F}$  is an alphabet of symbols,  $\mathcal{Q}$  is a set of states,  $\mathcal{Q}_f$  a set of final states and  $\Delta$  is a set of transitions (also called a transition table). Here, the tree automata module is defined w.r.t.

- a symbol type
- an alphabet type (the type of  $\mathcal{F}$ ) whose symbols are of symbol type
- a variable type. It is used for defining variables occuring in matching on tree automata
- a configuration type i.e. left-hand side of transitions
- a state content type which can be anything assigned to states: formulas, or simply text
- a transition type which is a term rewriting system and defined the type of  $\Delta$  we use
- a state set type defining the type of  $\mathcal{Q}$  and  $\mathcal{Q}_f$  we use. In fact, in practice its major role is to assign state contents to states.

module *TreeAutomata*

```
(Symbol_type : PRINTABLE_TYPE)
(Alphabet_type : ALPHABET_TYPE with type symbol = Symbol_type.t)
(Variable_type : PRINTABLE_TYPE)
(Configuration_type : TERM_TYPE with type symbol = Symbol_type.t
                        and type variable = Variable_type.t
                        and type alphabet = Alphabet_type.t)
(State_content : STATE_CONTENT_TYPE)
(Transition_type : TRS_TYPE with type alphabet = Alphabet_type.t
                  and type term = Configuration_type.t)
(State_set_type : STATE_SET_TYPE with type state = Configuration_type.t
                  and type state_content = State_content.t
                  and type alphabet = Alphabet_type.t
                  and type symbol = Symbol_type.t) :
```

sig

```
exception Not_a_state of string
exception Not_in_folder
exception Multiply_defined_symbol of string
exception Linearity_problem of string
exception Normalisation_problem of string

type symbol = Symbol_type.t
```

```

type alphabet = Alphabet_type.t
type term = Configuration_type.t
type rule = Transition_type.rule
type substitution = Configuration_type.substitution

type mini_subst = (term × term)

type sol_filt =
  — Empty
  — Bottom
  — S of mini_subst
  — Not of sol_filt
  — And of sol_filt × sol_filt
  — Or of sol_filt × sol_filt

type state = term
type state_set = State_set_type.t

type transition_table = Transition_type.t
type tree_automata

type t = tree_automata
type ('a, 'b) folder

```

**8.** Constructor of tree automata. The main difference with usual definitions of tree automata is that we here use an extended definition of states. States are terms (gasp!). States are terms constructed on a specific alphabet which is what we call *state operators*. This make no difference with usual definition of states and tree automata if you consider only state operators of arity 0 (i.e. constant state symbol) then if  $q$ ,  $state123$ ,  $q0$ ,  $q1$ , etc... are state operators of arity 0, then  $q$ ,  $state123$ ,  $q0$ ,  $q1$ , etc... are states. However, if you define a state operator  $q$  of arity 1, and  $qa$  of arity 0, then  $qa$ ,  $q(qa)$ ,  $q(q(qa))$ , ... are states. In fact, you can even define more complicated states, since state operators can transform any term (constructed on the alphabet and on state operators) into a state. For example, assume that your alphabet  $\mathcal{F}$  contains operators:  $f$  of arity 2 and  $b$  of arity 0, and your state operators contain at least  $q$  of arity 1 and  $qa$  of arity 0, then  $a$ ,  $q(qa)$ ,  $q(q(qa))$ ,  $q(b)$ ,  $q(f(b,b))$ ,  $q(f(qa, b))$ ,  $q(q(f(qa), b))$ , etc... are states.

In most cases, state operators of arity greater than 0 are not needed. Nevertheless, note that to define a simple tree automaton with state set  $\mathcal{Q} = \{q0, q2, q3\}$  and final states  $\mathcal{Q}_f = \{q2\}$ , you will need to define state operators  $q0, q2, q3$  of arity 0, and to give to the *make\_automaton* function the state operators (of alphabet type), the state set corresponding to  $\mathcal{Q}$  and then the set of final states representing  $\mathcal{Q}_f$ . However, it is much easier to use the parsing function of tree automata or, even simpler, the parsing function of the specification module, please have a look to the file *tutorial.mlml* for more details.

```

val make_automaton :
  alphabet →
  alphabet →
  state_set →
  state_set →
  transition_table →
  transition_table → t

```

build an automaton from a finite term list, a string label for states and an integer

```
val term_set_to_automaton : alphabet → term list → string → int → (t × int)
```

#### 9. accessors of automata

```
val get_alphabet : t → alphabet
val get_state_ops : t → alphabet
val get_states : t → state_set
val get_final_states : t → state_set
val get_transitions : t → transition_table
val get_prior : t → transition_table
```

#### 10. Prettyprint of tree automata. The first thing to be able to do with an automaton is to display it.

```
val print : t → unit
val to_string : t → string
```

#### 11. Now, we find the **boolean operations** on tree automata.

First of all, intersection of two tree automata. This function produces a tree automaton with structured states (states that are in fact products of states) and structured state sets (symbolic form of state set products). In order to obtain a full tree automaton with constructed state sets, apply accessibility cleaning (defined in the following) on it.

```
val inter : t → t → t
```

union of two tree automata (by renaming and union of transition tables, state set, final state sets etc...).

```
val union : t → t → t
```

The complement operation.

```
val inverse : t → t
```

The automaton recognizing the subtraction of languages: subtract  $L(a_2)$  to  $L(a_1)$

```
val subtract : t → t → t
```

Decision of inclusion between two languages: is  $L(a_1)$  included in  $L(a_2)$ ?

```
val is_included : t → t → bool
```

Decision of the emptiness of a *language* recognized by a tree automaton.

```
val is_language_empty : t → bool
```

Are the transitions recursive?

```
val is_recursive : transition_table → bool
```

Is the recognised language finite?

```
val finite_recognized_language : t → bool
```

#### 12. Make a run of a tree automaton: verify if a term $t$ rewrites into state $q$ with regards to transitions of automaton $a$ . This is not the usual definition of a run, but the usual one can easily be obtained from this one.

`val run : term → state → t → bool`

**13.** The determinisation function: given a tree automaton it gives an equivalent deterministic one.

`val determinise : t → t`

**14.** Completion of tree automaton... in a non-deterministic way i.e., the result is a non-deterministic tree automaton. If a deterministic one is needed, it needs to be determinised afterwards.

`val make_complete : t → t`

**15.** Construction of an automaton recognizing *reducible terms*. Starting from an alphabet  $a$  and a TRS  $r$  built on  $a$ , this function constructs the tree automaton recognizing terms reducible by  $r$ .

`val make_red_automaton :  
alphabet → Transition_type.t → t`

**16.** Construction of an automaton recognizing *irreducible terms*. Starting from an alphabet  $a$  and a TRS  $r$  built on  $a$ , this function constructs the tree automaton recognizing terms irreducible by  $r$ . The result is a deterministic complete tree automaton, it may be cleaned afterwards with *simplify* if necessary.

This implements a standard algorithm that is usually not efficient at all. For a better efficiency, use the next function called *nf\_opt*.

`val nf_automaton :  
alphabet → transition_table → t`

This one is usually more efficient than the previous one in practice. However the result is also slightly different: the produced tree automaton is not necessarily deterministic nor complete!

`val nf_opt :  
alphabet → transition_table → t`

**17.** Cleaning of tree automata

Accessibility cleaning of tree automaton: retrieves all states that do not recognize any term.

`val accessibility_cleaning : t → t`

Utility cleaning: retrieves all dead states. For utility cleaning on an automaton with structured state sets (obtained for example by application of an intersection operation use *accessibility\_cleaning* before this one.

`val utility_cleaning : t → t`

Accessibility cleaning followed by utility cleaning

`val clean : t → t`

Simplification of tree automaton: a renumbering of the result of cleaning (accessibility + utility) of the tree automaton. Useful for deciding if the language recognized by an automaton  $a$  is empty. If it is then *is\_empty(simplify a)* is true

`val simplify : t → t`

**18.** State Renumbering



This function rewrites state labels in a tree automaton  $a$  thanks to a term rewriting system  $r$  on states. Be careful! for state sets including states  $q1$ ,  $q2$  for example and if you use structured states labels like  $q(f(q1,q2))$ , if  $q1$  and  $q2$  are to be renamed into  $q3$  and  $q4$  respectively, then so is  $q(f(q1,q2))$  which is renamed into  $q(f(q3,q4))$ !!

```
val rewrite_state_labels :  
  t → transition_table → t
```

This function transforms a rewriting rule list (over states) used for state rewriting into an equivalent terminating one (by building some equivalence classes first)

```
val simplify_equivalence_classes : rule list → rule list
```

Automatic renumbering of a tree automaton. To apply this function on an automaton with structured state sets (obtained by intersection for example), use *accessibility\_cleaning* before this one.

```
val automatic_renum : t → t
```

19. For saving an automaton to disk, see function *save\_automaton* in the module specification.

## Low level functions

20. Emptiness of an *automaton*, i.e. emptiness of its transition table. For checking of the language apply *simplify* function before) i.e.,  $a$  is a tree automaton recognizing an empty language if and only if *is\_empty(simplify a)* is **true**.

```
val is_empty : t → bool
```

21. Modification of final state set.

```
val modify_final : t → state_set → t
```

22. Modification of prior transitions.

```
val modify_prior : t → transition_table → t
```

23. Modification of state operators.

```
val modify_state_ops : t → alphabet → t
```

24. Modification of state set.

```
val modify_states : t → state_set → t
```

25. Modification of state operators.

```
val modify_transitions : t → transition_table → t
```

26. Construction of a state from a symbol with arity 0. Recall that a state is a term!

```
val make_state : symbol → state
```

27. Construction of a state config from a state. A state config is a configuration (i.e. a lhs or a rhs of a transition) that is a state. For example: in  $q1 \rightarrow q2$ ,  $q1$  is a state configuration.

`val make_state_config : state → term`

**28.** Construction of a new transition

`val new_trans : symbol → state list → state → rule`

**29.** Is a configuration a state configuration? A state config is a configuration (i.e. a lhs or a rhs of a transition) that is a state. For example: in  $q1 \rightarrow q2$ ,  $q1$  is a state configuration.

`val is_state_config : term → bool`

**30.** State label of a state in a state configuration.

`val state_label : term → state`

`val lhs : rule → term`

`val rhs : rule → term`

**31.** Top symbol of a transition

`val top_symbol :  
rule → symbol`

**32.** Is a transition normalized? i.e. of the form  $f(q1, \dots, qn) \rightarrow q'$  where  $q1, \dots, qn$  are states.

`val is_normalized : rule → bool`

**33.** Construction of the list of states of the left hand side of a transition.

`val list_state : rule → state list`

**34.** Construction of the state set formed by the states of all the transition of the transition table.

`val states_of_transitions :  
transition_table → state_set`

**35.** Normalization of epsilon transitions of the form  $q1 \rightarrow q2$  with regards to a given transition table  $\delta$ .

`val normalize_epsilon :  
state →  
state →  
transition_table → transition_table`

**36.** Normalization of a transitions table  $ltrans$  with new states whose labels are  $label^j$  where  $j$  starts from  $i$ . It returns a triple with the new normalized transition table and the new state operator alphabet as well as the integer  $n+1$  where  $n$  is the number of the last assigned new state.  $\delta$  is simply used to normalise epsilon transitions found in  $ltrans$

`val normalize :  
transition_table →  
transition_table →  
string → int → transition_table × int × alphabet`

**37.** Similar to normalize but produces a deterministic set of transition

```

val normalize_deterministic :
  transition_table →
  transition_table →
  string → int → transition_table × int × alphabet

```

**38.** Matching of a term (ground or with variables) on a tree automaton configuration with regard to a transition list (here given as a folder of transitions sorted by top symbol and right-hand side (state)).

```

val matching :
  term →
  term →
  (symbol, (state, rule list) folder) folder →
  substitution list

```

**39.** Puts a *sol\_filt* (matching solution) in disjunctive normal form

```

val dnf : sol_filt → sol_filt

```

**40.** checks if a list of associations is a substitution i.e., a same variable cannot be mapped to different values. The substitution has to be given in a singleton list. The result is the empty list if the substitution is not valid

```

val check_subst : substitution list → substitution list

```

**41.** Simplification of matching solutions, by propagating Bottom solutions into the formula and retrieving Bottom occurring in disjunctions and retrieving conjunctions where Bottom occurs

```

val simplify_sol : sol_filt → sol_filt

```

**42.** Constructs the disjointness constraint. This is used to check that there is no non-linear lhs of a rule (say  $f(x,x)$ ) and no lhs of a transition (say  $f(q1,q2)$ ) such that the language recognized by  $q1$  and  $q2$  are not disjoint. The non-linear lhs are given in a list of terms  $l$ , the transitions are given as a folder  $f$  of transitions sorted by top symbol and right-hand side (state), and the result is a list of list of states whose disjointness has to be checked.

```

val disjointness_constraint :
  term list →
  (symbol, (state, rule list) folder) folder →
  state list list

```

**43.** Is a term  $t1$  rewritten into a state  $q$  (special term) by transitions contained in the folder  $f$ .

```

val is_recognized_into :
  term →
  state →
  (symbol, (state, rule list) folder) folder → bool

```

**44.** similar to the *is\_recognized\_into* but in the particular case where the transition is an epsilon transition  $q1 \rightarrow q2$ , this consists in verifying that all the transitions going to  $q1$  are already going to  $q2$ . Transitions are given into a transition table *delta*.

```

val is_covered :
  Configuration_type.t →
  Configuration_type.t →
  transition_table → bool

```

**45.** Parsing of a tree automaton with regards to an alphabet. For syntax, have a look to the *example.txt* file. See also the *file\_parse* function of the module specification *specification.mli*.

```

val parse : alphabet → Genlex.token Stream.t → t
end

```

## Interface for module Specification

**46.** This is the interface for specifications. What we call a specification is a collection of term rewriting systems and bottom-up tree automata all defined on a common alphabet. Consequently, this module is defined thanks to an alphabet type, a variable set type (used to define rewrite rules), a term rewriting system type and an automata type. Term rewriting system and tree automata are all assigned with a name (a string) in the specification. The simplest way to construct a specification is to write it in a file and parse it thanks to the *file\_parse* function. For a **sample specification file**, please look at the file *example.txt* contained in the distribution.

module *Specification*

```
(Alphabet_type : ALPHABET_TYPE)
(Variable_set_type : VARIABLE_SET_TYPE)
(Term_type : TERM_TYPE with type alphabet = Alphabet_type.t)
(TRS_type : TRS_TYPE with type alphabet = Alphabet_type.t
               and type variable_set = Variable_set_type.t)
(Automata_type : AUTOMATA_TYPE with type alphabet = Alphabet_type.t
               and type term = Term_type.t)
(Gamma_type : GAMMA_TYPE with type variable_set = TRS_type.variable_set
               and type alphabet = TRS_type.alphabet) :
```

sig

```
type alphabet = Alphabet_type.t
type variable_set = Variable_set_type.t
type trs = TRS_type.t
type automaton = Automata_type.t
type gamma_content = Gamma_type.gamma_content
type spec = {alphabet : alphabet; variables : variable_set;
             trs_list : (string × trs) list;
             automata_list : (string × automaton) list;
             gamma_list : (string × gamma_content) list}

type t = spec
exception Name_used_twice of string
exception No_TRS_of_that_name of string
exception No_automaton_of_that_name of string
exception No_approximation_of_that_name of string
exception No_name of string
```

**47.** Parsing of a specification in a file of name *file\_name*.

```
val file_parse : string → spec
```

**48.** Lexer for specifications

```
val lexer : char Stream.t → Genlex.token Stream.t
```

**49.** Get the alphabet of a specification *s*.

```

    val get_alphabet : spec → alphabet

50. Get the set of variables of a specification s.
    val get_variables : spec → variable_set

51. Get the term rewriting system named name in the specification s.
    val get_trs : string → spec → trs

52. Get the list of named term rewriting systems of a specification s.
    val get_list_trs : spec → (string × trs) list

53. Get the automaton named name in the specification s.
    val get_automaton : string → spec → automaton

54. Get the list of named automata of a specification s.
    val get_list_automata : spec → (string × automaton) list

55. Get the approximation named name in the specification s.
    val get_approximation : string → spec → gamma_content

56. Get the list of named approximation of a specification s.
    val get_list_approximation : spec → (string × gamma_content) list

57. Pretty print of a specification s.
    val to_string : spec → string

58. Writing a specification s to a file named file_name.
    val write_to_disk : spec → string → unit

59. Saving an automaton a under the name aut_name in a specification file named file_name.
    val save_automaton : automaton → string → string → unit
end

```

## Interface for module Term

**60.** This is the interface for terms of  $\mathcal{T}(\mathcal{F}, \mathcal{X})$  constructed on an alphabet  $\mathcal{F}$  and a set of variables  $\mathcal{X}$

module *Term*

(*Symbol\_type* : *PRINTABLE\_TYPE*)

(*Alphabet\_type* : *ALPHABET\_TYPE* with type *symbol* = *Symbol\_type.t*)

(*Variable\_type* : *PRINTABLE\_TYPE*)

(*Variable\_set\_type* : *VARIABLE\_SET\_TYPE* with type *variable* = *Variable\_type.t*) :

sig

**61.** A term is either a variable, a constant, a functionnal symbol with a list of subterms, or a special term. A special term is build on a union of the alphabet and a special alphabet.

For example, let  $\mathcal{F} = \{f : 2, g : 1, a : 0\}$  an alphabet and  $\mathcal{F}' = \{prod : 2, q : 0, h : 2\}$  a special alphabet.

Then  $f(g(a), g(prod(q, h(g(q), q))))$  is a term where  $prod(q, h(g(q), q))$  is a special subterm. The *Special()* constructor is used in the implementation to separate the special subterms in a term.

type *symbol* = *Symbol\_type.t*

type *variable* = *Variable\_type.t*

type *alphabet* = *Alphabet\_type.t*

type *variable\_set* = *Variable\_set\_type.t*

type *term* = (*Symbol\_type.t*, *Variable\_type.t*) *term\_const*

type *t* = *term*

type *substitution* = (*Variable\_type.t*  $\times$  *term*) *list*

exception *Terms\_do\_not\_match* of *string*  $\times$  *string*

exception *Terms\_do\_not\_unify* of *string*  $\times$  *string*

exception *Badly\_formed\_term* of *string*

exception *Parse\_error* of *string*

exception *Undefined\_symbol* of *string*

exception *Bad\_operation\_on\_special\_term* of *string*

exception *Bad\_operation\_on\_var* of *string*

val *equal* : *t*  $\rightarrow$  *t*  $\rightarrow$  *bool*

**62.** Depth of a term, where depth of Special terms, variables and constant is 0

val *depth* : *t*  $\rightarrow$  *int*

**63.** Pretty printing of terms into strings

val *to\_string* : *t*  $\rightarrow$  *string*

val *top\_symbol* : *t*  $\rightarrow$  *Symbol\_type.t*

**64.** the direct subterms of a term

- `val list_arguments : t → t list`
- 65.** is a term a variable?
- `val is_variable : t → bool`
- 66.** is a term a constant?
- `val is_constant : t → bool`
- 67.** is a term special: its top constructor is a Special constructor
- `val is_special : t → bool`
- 68.** get the term *t* from *Special*(*t*)
- `val get_special : t → t`
- 69.** mapping function *f1* on every symbol of term *t1* and *f2* on every constant, variable or special term
- `val map : ((Symbol_type.t → Symbol_type.t)) → ((t → t)) → (t) → t`
- 70.** get the list of the leaves of a term
- `val list_leaves : t → t list`
- 71.** get the list of variables of a term
- `val list_variables : t → Variable_type.t list`
- 72.** get the list of non linear variables of a term (with no redundancy)
- `val list_non_linear_variables : t → variable list`
- 73.** rename a variable: add a string to the end of the variable
- `val var_change : variable → string → variable`
- 74.** rename variables of a term: add a string to the end of every variable name
- `val rename_var : term → string → term`
- 75.** linearize a term: produces a linear version of a term *t* associated with the variable renamings that have been operated in order to make the term linear.
- `val linearize : term → (term × (variable × (variable list)) list)`
- 76.** is a term ground? i.e. with no variables. Note that a special term can be ground
- `val is_ground : t → bool`
- 77.** is a term linear? i.e. there is only one occurrence of each variable in the term
- `val is_linear : t → bool`
- 78.** get the list of all terms *t* such that *Special*(*t*) is a subterm of *t1*



**val** *list\_special* :  $t \rightarrow t \text{ list}$

**79.** Check the consistency of a term with regards to an alphabet. i.e. checks that for every subterm  $f(s_1, \dots, s_n)$  of  $t_1$ ,  $f$  has an arity  $n$  in the alphabet  $a$ . This function returns the term itself if it is correct, raise a *Badly\_formed\_term* exception if arity of the symbol does not correspond to its number of arguments, and raise a *Undefined\_symbol* exception if the term contains a symbol that does not belong to the alphabet.

**val** *check* : *Alphabet\_type.t*  $\rightarrow t \rightarrow t$

**80.** apply a substitution to a term (at every variable position in it)

**val** *apply* : *substitution*  $\rightarrow t \rightarrow t$

**81.** returns the list of terms ( $s \ t_1$ ) (substitution  $s$  applied to  $t_1$ ) for every substitution  $s$  of  $l$

**val** *apply\_several* : (*substitution list*)  $\rightarrow t \rightarrow t \text{ list}$

**82.** returns the list of terms ( $s \ t_1$ ) (substitution  $s$  applied to  $t_1$ ) for every substitution  $s$  of  $l$  and every term  $t_1$  of  $lt$

**val** *apply\_substs\_on\_terms* : (*substitution list*)  $\rightarrow t \text{ list} \rightarrow t \text{ list}$

**83.** Parsing of terms w.r.t. an alphabet  $a$  and a set of variable *varset*

**val** *parse* :  
     *Alphabet\_type.t*  $\rightarrow$   
     *Variable\_set\_type.t*  $\rightarrow \text{Genlex.token Stream.t} \rightarrow t$

**84.** Parsing of ground terms w.r.t. an alphabet  $a$

**val** *parse\_ground* :  
     *Alphabet\_type.t*  $\rightarrow$   
     *Genlex.token Stream.t*  $\rightarrow t$

**85.** Parsing of ground terms sets w.r.t. an alphabet  $a$

**val** *parse\_ground\_term\_set* :  
     *Alphabet\_type.t*  $\rightarrow$   
     *Genlex.token Stream.t*  $\rightarrow t \text{ list}$

**86.** Verify the coherence of a substitution: a variable must not be mapped to distinct terms. Otherwise a *Term\_do\_not\_match* exception is raised

**val** *coherent* : *substitution*  $\rightarrow \text{substitution}$

**87.** matching of *term1* on *term2*, such that *term2* is ground or at least with variables disjoint from those of *term1*.

**val** *matching* :  $t \rightarrow t \rightarrow \text{substitution}$

**88.** unification of *term1* on *term2*. No unification on Special terms. Variables of *term1* and *term2* are to be disjoint

`val unify : t → t → substitution`

**89.** similar functions for special terms

Check the consistency of a term with regards to an alphabet  $a$  and a special alphabet  $spa$  i.e. checks that for every subterm  $f(s1, \dots, sn)$  of  $t1$ ,  $f$  has an arity  $n$  in the alphabet if  $f(s1, \dots, sn)$  is a term or in  $spa$  if  $f(s1, \dots, sn)$  is below a Special constructor. This function returns the term itself if it is correct, raise a *Badly\_formed\_term* exception if arity of the symbol does not correspond to its number of arguments, and raise a *Undefined\_symbol* exception if the term contains a symbol that does not belong to the alphabets.

`val check_special : Alphabet_type.t → Alphabet_type.t → t → t`

replacement in special terms: for every pair  $(t1, t2)$  of  $l$ , replace every  $Special(t1)$  by  $Special(t2)$  at every Special position in  $t3$

`val replace_special : ((t × t) list) → t → t`

the map combinator on special terms

`val map_special : (Symbol_type.t → Symbol_type.t) → (t → t) → t → t`

Generalisation of substitution to special terms with any depth thanks to the combinator on terms: `Term.map_special`

`val apply_special : substitution → term → term`

Parsing of a term with special subterms w.r.t. alphabet  $a$  and special alphabet  $spa$ .

`val parse_special :  
 Alphabet_type.t →  
 Alphabet_type.t →  
 Variable_set_type.t → Genlex.token Stream.t → t`

Parsing of ground special terms w.r.t. alphabet  $a$  and special alphabet  $spa$ .

`val parse_ground_special :  
 Alphabet_type.t →  
 Alphabet_type.t →  
 Genlex.token Stream.t → t`

Applying matching on  $term1$  and  $term2$ , such that  $term2$  is ground or at least with disjoint set of variables. Special terms may contain variables

`val matching_special : t → t → substitution`

end

## Interface for module Rewrite

**90.** This is the interface for rewrite rules and rewrite systems constructed on an alphabet  $\mathcal{F}$ , a set of variables  $\mathcal{X}$  and a set of terms  $\mathcal{T}(\mathcal{F}, \mathcal{X})$

```
module RewriteSystem
  (Alphabet_type : ALPHABET_TYPE)
  (Variable_set_type : VARIABLE_SET_TYPE)
  (Term_type : TERM_TYPE with type variable_set = Variable_set_type.t
    and type alphabet = Alphabet_type.t) :
sig
  type alphabet = Alphabet_type.t
  type variable_set = Variable_set_type.t
  type term = Term_type.t
  type ruleSystem
  type t = ruleSystem
  type rule

  exception Variable_rhs_not_included_in_lhs of string
  exception Does_not_rewrite_on_top
  exception Badly_formed_rule of string
```

**91.** the empty trs and other constructors

```
val empty : t
val new_rule : term → term → rule
val is_empty : t → bool
val mem : rule → t → bool
```

**92.** adding a rule in a trs, and union of two trs

```
val add : rule → t → t
val union : t → t → t
```

**93.** if the rule is not in the trs we can catenate without testing membership

```
val add_fast : rule → t → t
```

**94.** if trs are known to be disjoint we can catenate without testing membership for union

```
val union_fast : t → t → t
```

**95.** first rule of a ruleSystem and remainder of the system

```
val first : t → rule
val remainder : t → t
```

nth rule of the system (in the parsing order)

- val** *nth* : *t* → *int* → *rule*
- 96.** right-hand side and left-hand side of a rule
- val** *rhs* : *rule* → *term*  
**val** *lhs* : *rule* → *term*
- 97.** equality on rules
- val** *rule\_equal* : *rule* → *rule* → *bool*
- 98.** is a rule left or right or left and right linear ?
- val** *is\_ground* : *rule* → *bool*  
**val** *is\_left\_linear* : *rule* → *bool*  
**val** *is\_right\_linear* : *rule* → *bool*  
**val** *is\_linear* : *rule* → *bool*
- 99.** list of non linear lhs of a ruleSystem
- val** *non\_linear\_lhs* : *t* → *term list*
- 100.** intersection of two trs
- val** *inter* : *t* → *t* → *t*
- 101.** moving from list to ruleSystem and conversely
- val** *to\_list* : *t* → *rule list*  
**val** *list\_to\_trs* : *rule list* → *t*
- 102.** prettyprint
- val** *rule\_to\_string* : *rule* → *string*  
**val** *to\_string* : *t* → *string*
- 103.** renaming every variable of a rule: adding a string to the end of every variable label
- val** *rename\_rule\_var* : *rule* → *string* → *rule*
- 104.** renaming every variable of a rewrite system: adding a string to the end of every variable label
- val** *rename\_var* : *t* → *string* → *t*
- 105.** Checking one rule with regards to an alphabet: checks construction of lhs and rhs as well as inclusion of var(rhs) in var(lhs)
- val** *check\_rule* : *rule* → *alphabet* → *rule*
- 106.** Checking a trs with regards to an alphabet: checks construction of lhs and rhs as well as inclusion of var(rhs) in var(lhs)
- val** *check* : *t* → *alphabet* → *t*
- 107.** parsing of a rule, given an alphabet *a* variable set *varset*

```

val parse_rule :
  alphabet →
    variable_set → Genlex.token Stream.t → rule

```

108. parsing of a trs given an alphabet *a* variable set *varset*

```

val parse :
  alphabet →
    variable_set → Genlex.token Stream.t → t

```

109. rewrite once on top position of term *t1* with any rule of trs *r*

```

val rewrite_top_once : t → Term_type.t → Term_type.t

```

110. leftmost innermost normalisation of the term *t1* thanks to a trs *r*. Of course TRS should terminate!

```

val left_inner_norm : t → Term_type.t → Term_type.t

```

111. bottom up normalisation of term *t1* thanks to trs *r*. Useful when the trs is a transition table of an automaton

```

val bottom_up_norm : t → Term_type.term → Term_type.t

```

112. similar functions but for rules and trs built on special terms ...

```

val check_special_rule :
  rule → alphabet → alphabet → rule
val check_special :
  t → alphabet → alphabet → t
val parse_special_rule :
  alphabet →
    alphabet →
      variable_set → Genlex.token Stream.t → rule
val parse_special :
  alphabet →
    alphabet →
      variable_set → Genlex.token Stream.t → t
val parse_ground_special :
  alphabet →
    alphabet → Genlex.token Stream.t → t
val parse_ground_special_rule :
  alphabet → alphabet → Genlex.token Stream.t → rule
val left_inner_norm_special :
  t → Term_type.t → Term_type.t
val left_inner_norm_special_system :
  t → t → t

```

end

## Interface for module Alphabet

**113.** This is the interface for alphabets which are sets of symbols associated with their arity i.e. their number of arguments

```
module Alphabet :  
  functor (Symbol_type : PRINTABLE_TYPE) →  
sig  
  type symbol = Symbol_type.t  
  type t  
  
  exception Symbol_not_in_alphabet of string  
  exception Multiply_defined_symbol of string
```

**114.** One alphabet constructor

```
val new_alphabet : t
```

**115.** Parsing of alphabets (another constructor)

```
val parse : Genlex.token Stream.t → t
```

**116.** Testing the occurrence of a symbol in an alphabet

```
val occur : symbol → t → bool
```

**117.** Adding a symbol with its arity in an alphabet

```
val add_symbol : symbol → int → t → t
```

**118.** Getting the arity of a symbol in an alphabet. This function raises the exception *Symbol\_not\_in\_alphabet(s)* where *s* is the string associated with the symbol if it is not in the alphabet

```
val get_arity : symbol → t → int  
val to_list : t → (symbol × int) list  
val to_string_list : t → string list
```

**119.** Testing disjointness of two alphabets

```
val disjoint : t → t → bool
```

**120.** Construct the union of two disjoint alphabets

```
val union_fast : t → t → t
```

**121.** Construct the union of two alphabets, possibly non-disjoint

```
val union : t → t → t
```

**122.** Pretty print

```
val to_string : t → string
```

```
end
```

## Interface for module State\_set

**123.** This is the interface for state sets. See the automaton module for a detailed description of the representation of states. State sets are sets of states associated with a state content which can be of various form: formulas, text, automaton (why not?)

```
module State_set
  (Symbol_type : PRINTABLE_TYPE)
  (Alphabet_type : ALPHABET_TYPE with type symbol = Symbol_type.t)
  (State_type : TERM_TYPE with type symbol = Symbol_type.t
    and type alphabet = Alphabet_type.t)
  (State_content : STATE_CONTENT_TYPE) :
sig
  type alphabet = Alphabet_type.t
  type symbol = Symbol_type.t
  type state_content = State_content.t
  type state = State_type.t
  type t

  exception State_not_in_state_set of string
  exception Not_a_state of string
  exception Structured_state_sets of string
```

**124.** Is a state set structured?

```
val is_structured : t → bool
```

**125.** The empty state set

```
val empty : t
```

**126.** Add a state with no state content to a state set

```
val add : state → t → t
```

**127.** Add a state with its content

```
val add_verb : state → state_content → t → t
```

**128.** Transform a list of state into a state set

```
val list_to_set : state list → t
```

**129.** Extract the list of states from a state set

```
val to_list : t → state list
```

**130.** Add all states of a list to a state set

- val** *add\_all* : *state list*  $\rightarrow$  *t*  $\rightarrow$  *t*
- 131.** Adds a list of states to a set *s1*, using their description coming from another set *s2*
- val** *add\_all\_verb* : *state list*  $\rightarrow$  *t*  $\rightarrow$  *t*  $\rightarrow$  *t*
- 132.** Is a state set empty? and is a state member of a state set?
- val** *is\_empty* : *t*  $\rightarrow$  *bool*  
**val** *mem* : *state*  $\rightarrow$  *t*  $\rightarrow$  *bool*
- 133.** The first element of a state set and the remainder
- val** *first* : *t*  $\rightarrow$  *state*  
**val** *remainder* : *t*  $\rightarrow$  *t*
- 134.** pretty print
- val** *to\_string* : *t*  $\rightarrow$  *string*
- 135.** pretty print in verbose mode, where the content is also printed in front of its corresponding state
- val** *to\_string\_verb* : *t*  $\rightarrow$  *string*
- 136.** get the state content associated to a state in a state set
- val** *state\_description* : *state*  $\rightarrow$  *t*  $\rightarrow$  *state\_content*
- 137.** The default binary symbol used for representing product of states
- val** *default\_prod\_symbol* : *symbol*
- 138.** construction of a product state from two states
- val** *state\_product* : *state*  $\rightarrow$  *state*  $\rightarrow$  *state*
- 139.** construction of the cartesian product of two state sets (in a symbolic way i.e. the cartesian product is not computed)
- val** *symbolic\_product* : *t*  $\rightarrow$  *t*  $\rightarrow$  *t*
- 140.** boolean operations on state sets
- val** *inter* : *t*  $\rightarrow$  *t*  $\rightarrow$  *t*  
**val** *union* : *t*  $\rightarrow$  *t*  $\rightarrow$  *t*  
**val** *minus* : *t*  $\rightarrow$  *t*  $\rightarrow$  *t*  
**val** *union\_disjoint* : *t*  $\rightarrow$  *t*  $\rightarrow$  *t*
- 141.** are all states from the list member of the state set
- val** *all\_mem* : *state list*  $\rightarrow$  *t*  $\rightarrow$  *bool*
- 142.** produce and add to a state operator alphabet *s1* all symbols labeled by *str*<sup>*k*</sup> where *k* takes the values from *i* to *j*



```

val produce :
  int → int → string → Alphabet_type.t →
    Alphabet_type.t

```

**143.** Parsing of symbols of state set

```

val parse_ops : Genlex.token Stream.t → Alphabet_type.t

```

**144.** Parsing of a state set

```

val parse :
  alphabet →
    alphabet → Genlex.token Stream.t → t

```

**145.** Parsing of a state set with associated state contents

```

val parse_verb :
  alphabet →
    alphabet → Genlex.token Stream.t → t

```

end

## Interface for module `Variable_set`

**146.** This is the interface for variable sets

```
module Variable_set :  
  functor (Variable_type : PRINTABLE_TYPE) →  
sig  
  type variable = Variable_type.t  
  type t  
  val empty : t  
  val is_empty : t → bool  
  val mem : variable → t → bool  
  val to_string : t → string  
  val to_string_list : t → string list  
  val parse : Genlex.token Stream.t → t  
end
```



## References

- [1] F. Baader and T. Nipkow. *Term Rewriting and All That*. Cambridge University Press, 1998.
- [2] Walter S. Brainerd. Tree generating regular systems. *Information and Control*, 14:217–231, 1969.
- [3] 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.
- [4] J.L. Coquidé, M. Dauchet, R. Gilleron, and S. Vágvolgyi. Bottom-up tree pushdown automata and rewrite systems. In R. V. Book, editor, *Proceedings 4th Conference on Rewriting Techniques and Applications, Como (Italy)*, volume 488 of *Lecture Notes in Computer Science*, pages 287–298. Springer-Verlag, 1991.
- [5] M. Dauchet and S. Tison. The theory of ground rewrite systems is decidable. In *Proceedings 5th IEEE Symposium on Logic in Computer Science, Philadelphia (Pa., USA)*, pages 242–248, June 1990.
- [6] D. Dolev and A. Yao. On the security of public key protocols. In *Proc. IEEE Transactions on Information Theory*, pages 198–208, 1983.
- [7] G. Feullade, T. Genet, and V. Viet Triem Tong. Reachability Analysis over Term Rewriting Systems. Technical Report RR-4970, Institut National de Recherche en Informatique et Automatique, 2003. <http://www.irisa.fr/lande/genet/timbuk/#papers>.
- [8] J.-C. Filiâtre and C. Marché. ocamlweb: a literate programming tool for Objective Caml. Institut National de Recherche en Informatique et Automatique, 2000. <http://www.lri.fr/~filliatr/ocamlweb/>.
- [9] T. Genet. Decidable approximations of sets of descendants and sets of normal forms. In *Proceedings 9th Conference on Rewriting Techniques and Applications, Tsukuba (Japan)*, volume 1379 of *Lecture Notes in Computer Science*, pages 151–165. Springer-Verlag, 1998.
- [10] T. Genet and F. Klay. Rewriting for Cryptographic Protocol Verification. In *Proceedings 17th International Conference on Automated Deduction, Pittsburgh (Pen., USA)*, volume 1831 of *Lecture Notes in Artificial Intelligence*. Springer-Verlag, 2000.
- [11] F. Jacquemard. Decidable approximations of term rewriting systems. In H. Ganzinger, editor, *Proceedings 7th Conference on Rewriting Techniques and Applications, New Brunswick (New Jersey, USA)*, pages 362–376. Springer-Verlag, 1996.
- [12] X. Leroy, D. Doligez, J. Garrigue, D. Rémy, and J. Vouillon. The Objective Caml system release 3.00 – Documentation and user’s manual. Institut National de Recherche en Informatique et Automatique, 2000. <http://caml.inria.fr/ocaml/htmlman/>.
- [13] G. Lowe. Breaking and fixing the Needham-Schroeder public-key protocol using CSP and FDR. In *Proceedings of the 2nd International Workshop on Tools and Algorithms for the Construction and Analysis of Systems, Passau (Germany)*, volume 1055 of *Lecture Notes in Computer Science*, pages 147–166. Springer-Verlag, 1996.

- [14] P. Réty. Regular Sets of Descendants for Constructor-based Rewrite Systems. In *Proceedings of the 6th International Conference on Logic Programming and Automated Reasoning, Tbilisi (Georgia)*, volume 1705 of *Lecture Notes in Artificial Intelligence*. Springer-Verlag, 1999.
- [15] K. Salomaa. Deterministic Tree Pushdown Automata and Monadic Tree Rewriting Systems. *Journal of Computer and System Sciences*, 37:367–394, 1988.

# Index

- accessibility\_cleaning*, 17
- add*, 92, 126
- add\_all*, 130
- add\_all\_verb*, 131
- add\_fast*, 93
- add\_symbol*, 117
- add\_verb*, 127
- all\_mem*, 141
- alphabet* (field), 46
- alphabet* (type), 7, 46, 61, 90, 123, 7–9, 15, 16, 23, 36, 37, 45, 46, 49, 90, 105–108, 112, 123, 144, 145
- Alphabet* (module), 113
- And*, 7
- apply*, 80
- apply\_several*, 81
- apply\_special*, 89
- apply\_substs\_on\_terms*, 82
- automata\_list* (field), 46
- automatic\_renum*, 18
- automaton* (type), 46, 46, 53, 54, 59
- Automaton* (module), 7
- Badly\_formed\_rule* (exn), 90
- Badly\_formed\_term* (exn), 61
- Bad\_operation\_on\_special\_term* (exn), 61
- Bad\_operation\_on\_var* (exn), 61
- bi\_folder\_add*, 45
- bi\_folder\_add\_trans\_list*, 45
- bi\_folder\_flatten*, 45
- bi\_folder\_mem*, 45
- Bottom*, 7
- bottom\_up\_norm*, 111
- check*, 79, 106
- check\_rule*, 105
- check\_special*, 89, 112
- check\_special\_rule*, 112
- check\_subst*, 40
- clean*, 17
- coherent*, 86
- configs\_from\_symbol\_to\_state*, 45
- default\_prod\_symbol*, 137
- depth*, 62
- determinise*, 13
- disjoint*, 119
- disjointness\_constraint*, 42
- dnf*, 39
- Does\_not\_rewrite\_on\_top* (exn), 90
- empty*, 91, 125, 146
- Empty*, 7
- equal*, 61
- file\_parse*, 47
- finite\_recognized\_language*, 11
- first*, 95, 133
- folder* (type), 7, 38, 42, 43, 45
- folder\_add*, 45
- folder\_assoc*, 45
- folder\_cartesian\_product*, 45
- folder\_flatten*, 45
- folder\_hd*, 45
- folder\_replace*, 45
- folder\_tail*, 45
- gamma\_content* (type), 46, 46, 55, 56
- gamma\_list* (field), 46
- get\_alphabet*, 9, 49
- get\_approximation*, 55
- get\_arity*, 118
- get\_automaton*, 53
- get\_final\_states*, 9
- get\_list\_approximation*, 56
- get\_list\_automata*, 54
- get\_list\_trs*, 52
- get\_prior*, 9
- get\_special*, 68
- get\_states*, 9
- get\_state\_ops*, 9
- get\_transitions*, 9
- get\_trs*, 51
- get\_variables*, 50
- inter*, 11, 100, 140
- inverse*, 11
- is\_constant*, 66
- is\_covered*, 44
- is\_empty*, 20, 91, 132, 146
- is\_empty\_folder*, 45
- is\_ground*, 76, 98
- is\_included*, 11
- is\_language\_empty*, 11
- is\_left\_linear*, 98

*is\_linear*, 77, 98  
*is\_normalized*, 32  
*is\_recognized\_into*, 43  
*is\_recursive*, 11  
*is\_right\_linear*, 98  
*is\_special*, 67  
*is\_state\_config*, 29  
*is\_structured*, 124  
*is\_variable*, 65  
*left\_inner\_norm*, 110  
*left\_inner\_norm\_special*, 112  
*left\_inner\_norm\_special\_system*, 112  
*lexer*, 48  
*lhs*, 30, 96  
*Linearity\_problem* (exn), 7  
*linearize*, 75  
*list\_arguments*, 64  
*list\_leaves*, 70  
*list\_non\_linear\_variables*, 72  
*list\_special*, 78  
*list\_state*, 33  
*list\_to\_set*, 128  
*list\_to\_trs*, 101  
*list\_variables*, 71  
*make\_automaton*, 8  
*make\_complete*, 14  
*make\_fast\_union*, 45  
*make\_red\_automaton*, 15  
*make\_state*, 26  
*make\_state\_config*, 27  
*map*, 69  
*map\_special*, 89  
*matching*, 38, 87  
*matching\_special*, 89  
*mem*, 91, 132, 146  
*mini\_subst* (type), 7, 7  
*minus*, 140  
*modify\_final*, 21  
*modify\_prior*, 22  
*modify\_states*, 24  
*modify\_state\_ops*, 23  
*modify\_transitions*, 25  
*Multiply\_defined\_symbol* (exn), 7, 113  
*Name\_used\_twice* (exn), 46  
*new\_alphabet*, 114  
*new\_rule*, 91  
*new\_trans*, 28  
*nf\_automaton*, 16  
*nf\_opt*, 16  
*non\_linear\_lhs*, 99  
*Normalisation\_problem* (exn), 7  
*normalize*, 36  
*normalize\_deterministic*, 37  
*normalize\_epsilon*, 35  
*Not*, 7  
*Not\_a\_state* (exn), 7, 123  
*Not\_in\_folder* (exn), 7  
*No\_approximation\_of\_that\_name* (exn), 46  
*No\_automaton\_of\_that\_name* (exn), 46  
*No\_name* (exn), 46  
*No\_TRS\_of\_that\_name* (exn), 46  
*nth*, 95  
*occur*, 116  
*Or*, 7  
*parse*, 45, 83, 108, 115, 144, 146  
*Parse\_error* (exn), 61  
*parse\_ground*, 84  
*parse\_ground\_special*, 89, 112  
*parse\_ground\_special\_rule*, 112  
*parse\_ground\_term\_set*, 85  
*parse\_ops*, 143  
*parse\_rule*, 107  
*parse\_special*, 89, 112  
*parse\_special\_rule*, 112  
*parse\_verb*, 145  
*print*, 10  
*produce*, 142  
*remainder*, 95, 133  
*rename\_rule\_var*, 103  
*rename\_var*, 74, 104  
*replace\_special*, 89  
*Rewrite* (module), 90  
*RewriteSystem* (module), 90  
*rewrite\_state\_labels*, 18  
*rewrite\_top\_once*, 109  
*rhs*, 30, 96  
*rule* (type), 7, 90, 7, 18, 28, 30–33, 38, 42, 43, 45, 91–93, 95–98, 101–103, 105, 107, 112  
*ruleSystem* (type), 90, 90  
*rule\_equal*, 97  
*rule\_to\_string*, 102  
*run*, 12  
*save\_automaton*, 59

*simplify*, **17**  
*simplify\_equivalence\_classes*, **18**  
*simplify\_sol*, **41**  
*sol\_filt* (type), **7**, 7, 39, 41  
*spec* (type), **46**, 46, 47, 49–58  
*Specification* (module), **46**  
*state* (type), **7**, **123**, 7, 12, 26–28, 30, 33, 35, 38, 42, 43, 45, 126–133, 136, 138, 141  
*states\_of\_transitions*, **34**  
*state\_content* (type), **123**, 7, 127, 136  
*state\_description*, **136**  
*state\_label*, **30**  
*State\_not\_in\_state\_set* (exn), **123**  
*state\_product*, **138**  
*state\_set* (type), **7**, 8, 9, 21, 24, 34  
*State\_set* (module), **123**  
*Structured\_state\_sets* (exn), **123**  
*substitution* (type), **7**, **61**, 7, 38, 40, 80–82, 86–89  
*subtract*, **11**  
*symbol* (type), **7**, **61**, **113**, **123**, 7, 26, 28, 31, 38, 42, 43, 45, 60, 116–118, 123, 137  
*symbolic\_product*, **139**  
*Symbol\_not\_in\_alphabet* (exn), **113**  
*term* (type), **7**, **61**, **90**, 7, 8, 12, 27, 29, 30, 38, 42, 43, 45, 46, 61, 74, 75, 89, 91, 96, 99, 111  
*Term* (module), **60**  
*Terms\_do\_not\_match* (exn), **61**  
*Terms\_do\_not\_unify* (exn), **61**  
*term\_set\_to\_automaton*, **8**  
*top\_symbol*, **31**, **63**  
*to\_list*, **101**, **118**, **129**  
*to\_string*, **10**, **57**, **63**, **102**, **122**, **134**, **146**  
*to\_string\_list*, **118**, **146**  
*to\_string\_verb*, **135**  
*transitions\_by\_state*, **45**  
*transitions\_by\_state\_by\_symbol*, **45**  
*transitions\_by\_symbol*, **45**  
*transitions\_from\_symbol\_to\_state*, **45**  
*transition\_table* (type), **7**, 8, 9, 11, 16, 18, 22, 25, 34–37, 44  
*TreeAutomata* (module), **7**  
*tree\_automata* (type), **7**, 7  
*trs* (type), **46**, 46, 51, 52  
*trs\_list* (field), **46**  
*Undefined\_symbol* (exn), **61**  
*unify*, **88**  
*union*, **11**, **92**, **121**, **140**  
*union\_disjoint*, **140**  
*union\_fast*, **94**, **120**  
*utility\_cleaning*, **17**  
*variable* (type), **61**, **146**, 7, 60, 72, 73, 75, 146  
*variables* (field), **46**  
*Variable\_rhs\_not\_included\_in\_lhs* (exn), **90**  
*variable\_set* (type), **46**, **61**, **90**, 46, 50, 90, 107, 108, 112  
*Variable\_set* (module), **146**  
*var\_change*, **73**  
*write\_to\_disk*, **58**