Tree Regular Model Checking for Lattice-Based Automata

Thomas Genet, Tristan Le Gall, Axel Legay, Valérie Murat

Tree Regular Model Checking for Lattice-Based Automata

Thomas Genet — Tristan Le Gall — Axel Legay — Valérie Murat

N° 424

March 2013

Thèmes SYM et NUM

Rapport technique

ISSN 0249-0803 ISRN INRIA/RT--424--FR+ENG

Tree Regular Model Checking for Lattice-Based Automata

Thomas Genet[⋆] , Tristan Le Gall[⋆⋆] , Axel Legay[⋆] , Valérie Murat[⋆]

Thèmes SYM et NUM — Systèmes symboliques et Systèmes numériques
Projet ??

Rapport technique n° 424 — March 2013 — 36 pages

Abstract: *Tree Regular Model Checking* (TRMC) is the name of a family of techniques for analyzing infinite-state systems in which states are represented by terms, and sets of states by Tree Automata (TA). The central problem in TRMC is to decide whether a set of bad states is reachable. The problem of computing a TA representing (an over-approximation of) the set of reachable states is undecidable, but efficient solutions based on completion or iteration of tree transducers exist. Unfortunately, the TRMC framework is unable to efficiently capture both the complex structure of a system and of some of its features. As an example, for JAVA programs, the structure of a term is mainly exploited to capture the structure of a state of the system. On the counter part, integers of the java programs have to be encoded with Peano numbers, which means that any algebraic operation is potentially represented by thousands of applications of rewriting rules. In this paper, we propose Lattice Tree Automata (LTAs), an extended version of tree automata whose leaves are equipped with lattices. LTAs allow us to represent possibly infinite sets of interpreted terms. Such terms are capable to represent complex domains and related operations in an efficient manner. We also extend classical Boolean operations to LTAs. Finally, as a major contribution, we introduce a new completion-based algorithm for computing the possibly infinite set of reachable interpreted terms in a finite amount of time.

Key-words: verification of infinite state systems, tree automata, abstract interpretation, lattice automata, completion, regular tree model checking

[⋆] INRIA/IRISA, Rennes
[⋆⋆] CEA, LIST, Centre de recherche de Saclay

Model Checking régulier pour automate d'arbres à treillis

Résumé : Le model checking régulier sur termes (TRMC) est une famille de techniques permettant d'analyser les systèmes à espace d'états infini dans lequel les états sont représentés par des termes, et les ensembles de termes par des automates d'arbres. Le problème principal du TRMC est de savoir si un ensemble d'états erreur est accessible ou non. Le calcul d'un automate d'arbres représentant (une sur-approximation de) l'ensemble des états accessibles est un problème indécidable. Mais des solutions efficaces basées sur la complétion ou l'itération de transducteurs d'arbres existent. Malheureusement, les techniques actuelles liées au TRMC ne permettent pas de capturer efficacement à la fois la structure complexe d'un système et certaines de ces caractéristiques. Si on prend par exemple les programmes Java, la structure d'un terme est principalement exploitée pour modéliser la structure d'un état du système. En contrepartie, les entiers présents dans le programmes Java doivent être encodés par des entiers de Peano, donc chaque opération algébrique est potentiellement modélisée par une centaine d'applications de règles de réécriture. Dans ce rapport, nous proposons des automates d'arbres à treillis (LTAs), une version étendue des automates d'arbres dont les feuilles sont équipés avec des éléments d'un treillis. Les LTAs nous permettent de représenter des ensembles possiblement infinis de termes pouvant être interprétés. Ces termes "interprétables" permettent de représenter efficacement des domaines complexes et leurs opérations associées. Nous étendons également les opérations booléennes classiques aux LTAs. Enfin, en tant que contribution principale, nous définissons un nouvel algorithme de complétion permettant de calculer l'ensemble possiblement infini des termes interprétables accessibles en un temps fini.

Mots-clés : vérification de systèmes à espace d'états infini, automate d'arbres, interprétation abstraite, treillis, complétion, model checking régulier

1 Introduction

Infinite-state models are often used to avoid potentially artificial assumptions on data structures and architectures, e.g. an artificial bound on the size of a stack or on the value of an integer variable. At the heart of most of the techniques that have been proposed for exploring infinite state spaces, is a symbolic representation that can finitely represent infinite sets of states.

In early work on the subject, this representation was domain specific, for example linear constraints for sets of real vectors [33]. For several years now, the idea that a generic automata-based representation for sets of states could be used in many settings has gained ground starting with finite-word automata [14,15,29,2], and then moving to the more general setting of Tree Regular Model Checking (TRMC) [1,17,3]. In TRMC, states are represented by trees, set of states by tree automata, and behavior of the system by rewriting rules or tree transducers. Contrary to specific approaches, TRMC is generic and expressive enough to describe a broad class of communication protocols [3], various C programs [16] with complex data structures, multi-threaded programs, and even cryptographic protocols [26,6]. Any Tree Regular Model Checking approach is equipped with an acceleration algorithm to compute possibly infinite sets of states in a finite amount of time. Among such algorithms, one finds completion by equational abstraction [27] that computes successive automata obtained by application of the rewriting rules, and merge intermediary states according to an equivalence relation to enforce the termination of the process.

In [11], the authors proposed an exact translation of the semantic of the *Java* Virtual Machine to tree automata and rewriting rules. This translation permits to analyze *Java* programs with classical Tree Regular Model checkers. One of the major difficulties of this encoding is to capture and handle the two-side infinite dimension that can arise in *Java* programs. Indeed, in such models, infinite behaviors may be due to unbounded calls to method and object creation, or simply because the program is manipulating unbounded data such as integer variables. While multiple infinite behaviors can be over-approximated with completion and equational abstraction [27], their combinations may require the use of artificially large-size structures. As an example in [11], the structure of a configuration is represented in a very concise manner as the structure of terms is mainly designed to efficiently capture program counters, stacks, ... On the other hand, integers and their related operations have to be encoded in Peano arithmetic, which has an exponential impact on the size of automata representing sets of states as well as on the computation process. As an example, the addition of x to y requires the application of x rewriting rules.

A solution to the above problem would be to follow the solution of Kaplan [28], and represent integers in bases greater or equal to 2, and the operations between them in the alphabet of the term directly. In such a case, the term could be interpreted and returns directly the result of the operation without applying any rewriting rule. The study of new Tree Regular Model Checking approaches for such interpreted terms is the main objective of this paper. Our first contribution is the definition of *Lattice Tree Automata* (LTA), a new class of tree automata that is capable of representing possibly infinite sets of interpreted terms. Roughly speaking, LTA are classical Tree Automata whose leaves may be equipped with lattice elements to abstract possibly infinite sets of values. Nodes of LTA can either be defined on an uninterpreted alphabet, or represent lattice operations, which will allow us to interpret possibly infinite sets of terms in a finite amount of time. We also propose a study of all the classical automata-based operations for LTA. The model of LTA is not closed under determinization. In such case, the best that can be done is to propose an over-approximation of the resulting automaton through abstract interpretation. As a third contribution, we propose a new acceleration algorithm to compute the set of reachable states of systems whose states are encoded with interpreted terms and sets of states with LTA. Our algorithm extends the classical completion approach by considering conditional term rewriting systems for lattices. We show that dealing with such conditions requires to merge existing completion algorithm with a solver for abstract domains. We also propose a new type of equational abstraction for lattices, which allows us to enforce termination in a finite amount of time. Finally, we show that our algorithm is correct in the sense that it computes an over-approximation of the set of reachable states. This latter property is only guaranteed providing that each completion step is followed by an evaluation operation. This operation, which relies on a widening operator, adds terms that may be lost during the completion step. Finally, we briefly describe how our solution can drastically improve the encoding of *Java* programs in a TRMC environment.

Related Work This work is inspired by [24], where the authors proposed to use finite-word lattice automata to solve the Regular Model Checking problem. Our major differences are that (1) we work with trees, (2) we propose a more general acceleration algorithm, and (3) we do consider operations on lattices while they only consider to label traces with lattices without permitting to combine them. Some Regular Model Checking approaches can be found in [4,14,5,18]. However, none of them can capture the two infinite-dimensions of complex systems in an efficient manner. Other models, like modal automata [9] or data trees [23,25], consider infinite alphabets, but do not exploit the lattice structure as in our work. Lattice (-valued) automata [30], whose transitions are labelled by lattice elements, map words over a finite alphabet to a lattice value. Similar automata may define fuzzy tree languages [21]. Other verification of particular classes of properties of *Java* programs with interpreted terms can be found in [32].

2 Background

Rewriting Systems and Tree Automata. Let $\mathcal{F}$ be a finite set of functional symbols, where each symbol is associated with an arity, and let $\mathcal{X}$ be a countable set of *variables*. $\mathcal{T}(\mathcal{F}, \mathcal{X})$ denotes the set of *terms* and $\mathcal{T}(\mathcal{F})$ denotes the set of *ground terms* (terms without variables). The set of variables of a term t is denoted by $\mathcal{V}ar(t)$. The set of functional symbols of arity n is denoted by $\mathcal{F}^n$. A *position* p for a term t is a word over $\mathbb{N}$. The empty sequence ε denotes the top-most position. We denote by $Pos(t)$ the set of positions of a term t. If $p \in \mathcal{P}os(t)$, then $t|_p$ denotes the subterm of t at position p and $t[s]_p$ denotes the term obtained by replacement of the subterm $t|_p$ at position p by the term s.

A *Term Rewriting System* (TRS) $\mathcal{R}$ is a set of *rewrite rules* $l \to r$, where $l, r \in \mathcal{T}(\mathcal{F}, \mathcal{X})$, and $\mathcal{V}ar(l) \supseteq \mathcal{V}ar(r)$. A rewrite rule $l \to r$ is *left-linear* if each variable of l occurs only once in l. A TRS $\mathcal{R}$ is left-linear if every rewrite rule $l \to r$ of $\mathcal{R}$ is left-linear.

We now define Tree Automata (TA for short) that are used to recognize possibly infinite sets of terms. Let $\mathcal{Q}$ be a finite set of symbols of arity 0, called *states*, such that $\mathcal{Q} \cap \mathcal{F} = \emptyset$. The set of *configurations* is denoted by $\mathcal{T}(\mathcal{F} \cup \mathcal{Q})$. A *transition* is a rewrite rule $c \to q$, where c is a configuration and q is a state. A transition is *normalized* when $c = f(q_1, \ldots, q_n)$, $f \in \mathcal{F}$ is of arity n, and $q_1, \ldots, q_n \in \mathcal{Q}$. A bottom-up nondeterministic finite tree automaton (tree automaton for short) over the alphabet $\mathcal{F}$ is a tuple $\mathcal{A} = \langle \mathcal{Q}, \mathcal{F}, \mathcal{Q}_F, \Delta \rangle$, where $\mathcal{Q}_F \subseteq \mathcal{Q}$ is the set of final states, Δ is a set of normalized transitions.

The transitive and reflexive *rewriting relation* on $\mathcal{T}(\mathcal{F} \cup \mathcal{Q})$ induced by Δ is denoted by $\to^*_{\mathcal{A}}$. The tree language recognized by $\mathcal{A}$ in a state q is $\mathcal{L}(\mathcal{A}, q) = \{t \in \mathcal{T}(\mathcal{F}) \mid t \to^*_{\mathcal{A}} q\}$. We define $\mathcal{L}(\mathcal{A}) = \bigcup_{q \in \mathcal{Q}_F} \mathcal{L}(\mathcal{A}, q)$.

Lattices, atomic lattices, Galois connections. A partially ordered set $(\Lambda, \sqsubseteq)$ is a lattice if it admits a *smallest element* $\perp$ and a *greatest element* $\top$, and if any finite set of elements $X \subseteq \Lambda$ admits a *greatest lower bound (glb)* $\sqcap X$ and a *least upper bound (lub)* $\sqcup X$. A lattice is complete if the *glb* and *lub* operators are defined for all possibly infinite subsets of Λ. An element x of a lattice $(\Lambda, \sqsubseteq)$ is an atom if it is minimal, *i.e.* $\perp \sqsubset x \wedge \forall y \in \Lambda : \perp \sqsubset y \sqsubseteq x \Rightarrow y = x$. The set of atoms of Λ is denoted by $Atoms(\Lambda)$. A lattice $(\Lambda, \sqsubseteq)$ is atomic if all element $x \in \Lambda$ where $x \neq \perp$ is the least upper bound of atoms, *i.e.* $x = \sqcup\{a | a \in Atoms(\Lambda) \wedge a \sqsubseteq x\}$.

Considered two lattices $(C, \sqsubseteq_C)$ (the concrete domain) and $(A, \sqsubseteq_A)$ (the abstract domain). We say that there is a *Galois connection* between the two lattices if there are two monotonic functions $\alpha : C \to A$ and $\gamma : A \to C$ such that : $\forall x \in C, y \in A$, $\alpha(x) \sqsubseteq_A y$ if and only if $x \sqsubseteq_C \gamma(y)$. As an example, sets of integers $(2^{\mathbb{Z}}, \subseteq)$ can be abstracted by the atomic lattice $(\Lambda, \sqsubseteq)$ of intervals, whose bounds belong to $\mathbb{Z} \cup \{-\infty, +\infty\}$ and whose atoms are of the form $[x, x]$, for each $x \in \mathbb{Z}$. Any operation *op* defined on a concrete domain C can be lifted to an operation $op^{\#}$ on the corresponding abstract domain A, thanks to the Galois connection.

3 Lattice Tree Automata

In this section, we first explain how to add elements of a concrete domain into terms, which has been defined in [28], and how to derive an abstract domain from a concrete one. Then we propose a new type of tree automata recognizing terms with elements of a lattice and study its properties.

3.1 Interpreted Symbols and Evaluation

In what follows, elements of a concrete and possibly infinite domain $\mathcal{D}$ will be represented by a set of *interpreted* symbols $\mathcal{F}_\bullet$. The set of symbols is now denoted by $\mathcal{F} = \mathcal{F}_\circ \cup \mathcal{F}_\bullet$, where $\mathcal{F}_\circ$ is the set of *passive* (uninterpreted) symbols. The set of *interpreted* symbols $\mathcal{F}_\bullet$ is composed of elements of $\mathcal{D}$ ($i.e$ $\mathcal{D} \subseteq \mathcal{F}_\bullet$) whose arity is 0, and is also composed of some predefined operations $f : \mathcal{D}^n \rightarrow \mathcal{D}$, where $f \in \mathcal{F}^n$. For example, if $\mathcal{D} = \mathbb{Z}$, then $\mathcal{F}_\bullet$ can be $\mathbb{Z} \cup \{+, -, *\}$. Passive symbols can be seen as usual non-interpreted functional operators, and interpreted symbols stand for *built-in* operations on the domain $\mathcal{D}$.

The set $\mathcal{T}(\mathcal{F}_\bullet)$ of terms built on $\mathcal{F}_\bullet$ can be evaluated by using an eval function $eval : \mathcal{T}(\mathcal{F}_\bullet) \rightarrow \mathcal{D}$. The purpose of *eval* is to simplify a term using the built-in operations of the domain $\mathcal{D}$. The *eval* function naturaly extends to $\mathcal{T}(\mathcal{F})$ in the following way: $eval(f(t_1, \dots, t_n)) = f(eval(t_1), \dots, eval(t_n))$ if $f \in \mathcal{F}_\circ$ or $\exists i = 1 \dots n : t_i \notin \mathcal{T}(\mathcal{F}_\bullet)$. Otherwise, $f(t_1, \dots, t_n) \in \mathcal{T}(\mathcal{F}_\bullet)$ and the evaluation returns an element of $\mathcal{D}$.

We can define tree automata to recognize sets of interpreted terms. For example, the set $\{f(1), f(2), f(3), f(4), f(5), f(6)\}$ is recognized by a tree automaton with the transitions : $1 \rightarrow q, 2 \rightarrow q, 3 \rightarrow q, 4 \rightarrow q, 5 \rightarrow q, 6 \rightarrow q, f(q) \rightarrow q_f$. This encoding is inefficient, and we would prefer to have special transitions to handle sets of integers. So we can define a tree automata with only two transitions : $\{1, \dots, 6\} \rightarrow q, f(q) \rightarrow q_f$.

Our main idea is to generalize this encoding and to define tree automata with some transitions to recognize elements of a lattice (sets of integers are elements of the lattice $2^{\mathbb{Z}}$). By considering tree automata with a generic lattice, we can also improve the efficiency of the approach. General sets of integers are indeed hard to handle, and we often only need an over-approximation of the set of reachable states. That is why we prefer to label the leaves of the tree by elements of an abstract lattice $(\Lambda, \sqsubseteq)$ such as the lattice of intervals. Moreover, we assume that this abstract lattice is atomic (cf. Section 2).

To each built-in operation $op \in OP$ defined on $\mathcal{D}$, we define the corresponding operation $op^\# \in OP^\#$ defined on Λ. Since we have that $\mathcal{F}_\bullet = \mathcal{D} \cup OP$, the set of *abstract* symbol is $\mathcal{F}_\bullet^\# = \Lambda \cup OP^\# \cup \{\sqcup, \sqcap\}$. The arity of $\sqcup$ and $\sqcap$ is 2 and the arity of $op^\#$ is the same as the one of *op*. For example, let I be the set of intervals with bounds belonging to $\mathbb{Z} \cup \{-\infty, +\infty\}$. The set $\mathcal{F}_\bullet = \mathbb{Z} \cup \{+, -\}$ can be abstracted by $\mathcal{F}_\bullet^\# = I \cup \{+^\#, -^\#, \sqcup, \sqcap\}$. Terms containing some operators extended to the abstract domain have to be evaluated, like explained in section 3.2 for the concrete domain. If there is a Galois connection between the concrete domain and the abstract one, $eval^\# : \mathcal{T}(\mathcal{F}_\bullet^\#) \mapsto \Lambda$ is the best approximation of *eval* w.r.t. this Galois connection.

Example 1 ($eval^\#$ function). There is a Galois connection between $(2^{\mathbb{Z}}, \subseteq)$ and the lattice of intervals $(I, \sqsubseteq)$. $eval^\#$ is defined by:

- $eval^\#(i) = i$ for any interval i, so $eval^\#(\bot) = \emptyset$ and $eval^\#(\top) = \mathbb{Z}$,
- For any $f \in \{+^\#, -^\#, \sqcup, \sqcap\}$ $eval^\#(f(i_1, i_2))$ is defined, given $eval^\#(i_1) = [a, b]$ and $eval^\#(i_2) = [c, d]$, by:
  - $eval^\#([a, b] \sqcup [c, d]) = [min(a, c), max(b, d)]$, $eval^\#([a, b] \sqcap [c, d]) = [max(a, c), min(b, d)]$ if $max(a, c) \leq min(b, d)$, else $eval^\#([a, b] \sqcap [c, d]) = \bot$,
  - $eval^\#([a, b] +^\# [c, d]) = [a + c, b + d]$, $eval^\#([a, b] -^\# [c, d]) = [a - d, b - c]$.

3.2 The Lattice Tree Automata Model

Lattice tree automata are extended tree automata recognizing terms defined on $\mathcal{F}_\circ \cup \mathcal{F}_\bullet^\#$.

Definition 1 (lattice tree automaton). *A bottom-up non-deterministic finite tree automaton with lattice (lattice tree automaton for short, LTA) is a tuple* $\mathcal{A} = \langle \mathcal{F} = \mathcal{F}_\circ \cup \mathcal{F}_\bullet^\#, \mathcal{Q}, \mathcal{Q}_F, \Delta \rangle$*, where* $\mathcal{F}$ *is a set of passive and interpreted symbols,* $\mathcal{Q}$ *a set of states and* $\mathcal{Q}_F$ *a set of final states,* $\mathcal{Q}_F \subseteq \mathcal{Q}$*, and* Δ *is a set of normalized transitions.*

The set of *lambda transitions* is defined by $\Delta_\Lambda = \{\lambda \to q \mid \lambda \to q \in \Delta \;\wedge\; \lambda \neq \bot \wedge\; \lambda \in \Lambda\}$. The set of *ground transitions* is the set of other transitions of the automaton, and is formally defined by $\Delta_G = \{f(q_1, \dots, q_n) \to q \mid f(q_1, \dots, q_n) \to q \in \Delta \;\wedge\; q, q_1, \dots, q_n \in Q\}$.

We extend the partial ordering $\sqsubseteq$ (on Λ) on $\mathcal{T}(\mathcal{F})$:

Definition 2. *Given* $s, t \in \mathcal{T}(\mathcal{F})$*,* $s \sqsubseteq t$ *iff :*

1. $s \sqsubseteq t$ *(if both* s *and* t *belong to* Λ*),*
2. $eval(s) \sqsubseteq eval(t)$ *(if both* s *and* t *belong to* $\mathcal{T}(\mathcal{F}_\bullet)$*),*
3. $s = t$ *(if both* s *and* t *belong to* $\mathcal{F}_\circ^0$*), or*
4. $s = f(s_1, \dots, s_n)$*,* $t = f(t_1, \dots, t_n)$*,* $f \in \mathcal{F}_\circ^n$ *and* $s_1 \sqsubseteq t_1 \wedge \dots \wedge s_n \sqsubseteq t_n$.

Example 2. $f(g(a, [1,5]) \sqsubseteq f(g(a, [0,8])$, and $h([0,4] +^\# [2,6]) \sqsubseteq h([1,3] +^\# [1,9])$.

In what follows we will omit # when it is clear from the context. We now define the transition relation and recognized language induced by an LTA. The difference with TA is that a term t is recognized by an LTA if $eval(t)$ can be reduced in the LTA.

Definition 3 ($t_1 \to_\mathcal{A} t_2$ for lattice tree automata). *Let* $t_1, t_2 \in \mathcal{T}(\mathcal{F} \cup \mathcal{Q})$.
$t_1 \to_\mathcal{A} t_2$ *iff, for any position* $p \in Pos(t_1)$ *:*

- *if* $t_1|_p \in \mathcal{T}(\mathcal{F}_\bullet)$*, there is a transition* $\lambda \to q \in \mathcal{A}$ *such that* $eval(t_1|_p) \sqsubseteq \lambda$ *and* $t_2 = t_1[q]_p$
- *if* $t_1|_p = a$ *where* $a \in \mathcal{F}_\circ$*, there is a transition* $a \to q \in \mathcal{A}$ *such that* $t_2 = t_1[q]_p$.
- *if* $t_1|_p = f(s_1, \dots, s_n)$ *where* $f \in \mathcal{F}^n$ *and* $s_1, \dots s_n \in \mathcal{T}(\mathcal{F} \cup \mathcal{Q})$*,* $\exists s'_i \in \mathcal{T}(\mathcal{F} \cup \mathcal{Q})$ *such that* $s_i \to_\mathcal{A} s'_i$ *and* $t_2 = t_1[f(s_1, \dots, s_{i-1}, s'_i, s_{i+1}, \dots, s_n)]_p$.

$\to_\mathcal{A}^*$ is the reflexive transitive closure of $\to_\mathcal{A}$. There is a run from t_1 to t_2 if $t_1 \to_\mathcal{A}^* t_2$.

The set $\mathcal{T}(\mathcal{F}, Atoms(\Lambda))$ denotes the set of ground terms built over $(\mathcal{F} \setminus \Lambda) \cup Atoms(\Lambda)$. Tree automata with lattice recognize a tree language over $\mathcal{T}(\mathcal{F}, Atoms(\Lambda))$.

Definition 4 (Recognized language). *The tree language recognized by* $\mathcal{A}$ *in a state* q *is* $\mathcal{L}(\mathcal{A}, q) = \{t \in \mathcal{T}(\mathcal{F}, Atoms(\Lambda)) \mid \exists\ t'$ *such that* $t \sqsubseteq t'$ *and* $t' \to_\mathcal{A}^* q\}$*. The language recognized by* $\mathcal{A}$ *is* $\mathcal{L}(\mathcal{A}) = \bigcup_{q \in \mathcal{Q}_f} \mathcal{L}(\mathcal{A}, q)$.

Example 3 (Run, recognized language). Let $\mathcal{A} = \langle \mathcal{F} = \mathcal{F}_\circ \cup \mathcal{F}_\bullet^\#, \mathcal{Q}, \mathcal{Q}_f, \Delta \rangle$ be an LTA where $\Delta = \{[0,4] \to q_1, f(q_1) \to q_2\}$ and final state q_2. We have: $f([1,4]) \to^* q_2$ and $f([0,2]) \to^* q_2$, and the recognized langage of $\mathcal{A}$ is given by $\mathcal{L}(\mathcal{A}, q_2) = \{f([0,0]), f([1,1]), \dots, f([4,4])\}$.

There are two reasons why we consider only atomic abstract lattices, and why the language of an LTA is defined on terms built with the atoms rather that with any elements of the lattice. The first one is that we are mostly interested in representing sets of integers. Since the atoms are the integers, the semantics of a lambda transition is to recognize a set of integers. The other reason is a technical one : It ensures that when we transform a LTA according to a partition (in the next subsection), we do not change the recognized language since the set of atoms are preserved by this transformation.

3.3 Operations on LTA

Most of the algorithms for Boolean operations on LTA are straightforward adaptations of those defined on TA (see [19]).

LTA are closed by union and intersection, and we shortly explain how these two operations $\cup$ and $\cap$ can be performed on two LTAs $\mathcal{A} = \langle \mathcal{F}, \mathcal{Q}, \mathcal{Q}_f, \Delta \rangle$ and $\mathcal{A}' = \langle \mathcal{F}, \mathcal{Q}', \mathcal{Q}'_f, \Delta' \rangle$:

- $\mathcal{A} \cup \mathcal{A}' = \langle \mathcal{F}, \mathcal{Q} \cup \mathcal{Q}', \mathcal{Q}_f \cup \mathcal{Q}'_f, \Delta \cup \Delta' \rangle$ assuming that the sets $\mathcal{Q}$ and $\mathcal{Q}'$ are disjoint.
- $\mathcal{A} \cap \mathcal{A}'$ is recognized by the LTA $\mathcal{A} \cap \mathcal{A}' = \langle \mathcal{F}, \mathcal{Q} \times Q', \mathcal{Q}_f \times \mathcal{Q}'_f, \Delta_\cap \rangle$ where the transitions of $\Delta_\cap$ are defined by the rules:

$$\frac{\lambda \to q \in \Delta \quad \lambda' \to q' \in \Delta' \quad \lambda \sqcap \lambda' \neq \bot}{\lambda \sqcap \lambda' \to (q, q')}$$

and

$$\frac{f(q_1, \ldots, q_n) \to q \in \Delta \quad f(q'_1, \ldots, q'_n) \to q' \in \Delta'}{f((q_1, q'_1), \ldots, (q_n, q'_n)) \to (q, q')}$$

TRMC also requires a complement operation and the emptyness test. The complement automaton is obtained by complementing the set of final states, but this algorithm only works if the input is a deterministic LTA. To decide if the language described by an LTA is empty or not, it suffices to observe that an LTA accepts at least one tree if and only if there is a reachable final state. A reduced automaton is an automaton without inaccessible state. The language recognized by a reduced automaton is empty if and only if the set of final states is empty. As a first step we thus have to reduce the LTA, that is to remove the set of unreachable states.

Let us recall the reduction algorithm:

```
Reduction Algorithm
input: LTA $\mathcal{A} = \langle \mathcal{F}, \mathcal{Q}, \mathcal{Q}_f, \Delta \rangle$
begin
    $Marked := \emptyset$
    /* Marked is the set of accessible states */
    repeat
        if $\exists a \in \mathcal{F}^0 = \mathcal{F}^0_\circ \cup \mathcal{F}^{\#^0}_\bullet$ such that $a \to q \in \Delta$
          or $\exists f \in \mathcal{F}^n = \mathcal{F}^n_\circ \cup \mathcal{F}^{\#^n}_\bullet$ such that $f(q_1, \ldots, q_n) \to q \in \Delta$
```

```
            where $q_1, \ldots, q_n \in Marked$
        then $Marked := Marked \cup \{q\}$
    until no state can be added to $Marked$
    $\mathcal{Q}_r := Marked$
    $\mathcal{Q}_{r_f} := \mathcal{Q}_f \cap Marked$
    $\Delta_r := \{f(q_1, \ldots, q_n) \to q \in \Delta | q, q_1, \ldots, q_n \in Marked\}$
    **output:** Reduced LTA $\mathcal{A}_r = \langle \mathcal{F}, \mathcal{Q}_r, \mathcal{Q}_{r_f}, \Delta_r \rangle$
**end**
```

Then, let $\mathcal{A}$ be an LTA and $\mathcal{A}_r = \langle \mathcal{F}, \mathcal{Q}_r, \mathcal{Q}_{r_f}, \Delta_r \rangle$ the corresponding reduced LTA, $\mathcal{L}(\mathcal{A})$ is empty iff $\mathcal{Q}_{r_f} = \emptyset$.

Let $\mathcal{A}_1$, $\mathcal{A}_2$ be two LTA. We have $\mathcal{L}(\mathcal{A}_1) \subseteq \mathcal{L}(\mathcal{A}_2) \Leftrightarrow \mathcal{L}(\mathcal{A}_1 \cap \overline{\mathcal{A}_2}) = \emptyset$.

So complementation and inclusion operations require deterministic inputs. However, by adapting the proof of finite-word lattice automata given in [24], one can show that LTA are not closed under determinization. In the next section, we propose an algorithm that computes an over-approximation deterministic automaton for any given *LTA*. This algorithm, which extends the one of [24], relies on a partition function that can be refined to make the overapproximation more precise.

3.4 Determinization

As we shall now see, an LTA $\mathcal{A} = \langle \mathcal{F}, \mathcal{Q}, \mathcal{Q}_f, \Delta \rangle$ is *deterministic* if there is no transition $f(q_1, \ldots, q_n) \to q$, $f(q_1, \ldots, q_n) \to q'$ in Δ such that $q \neq q'$, where $f \in \mathcal{F}_n$, and no transition $\lambda_1 \to q$, $\lambda_2 \to q'$ such that $q \neq q'$ and $\lambda_1 \sqcap \lambda_2 \neq \bot$, where $\lambda_1, \lambda_2 \in \Lambda$. As an example, if $\Delta = \{[1,3] \to q_1, [2,5] \to q_2\}$, then we have that $\mathcal{A}$ is not deterministic.

Determinizing an LTA requires complementation on elements on lattice. Indeed, consider the LTA $\mathcal{A}$ having the following transitions $[-3,2] \to q_1$ and $[1,6] \to q_2$. The deterministic LTA corresponding to $\mathcal{A}$ should have the following transitions: $[-3,1[\to q_1$, $[1,2] \to \{q_1,q_2\}$ and $]2,6] \to q_2$. To produce those transitions, we have to compute $[-3,2] \sqcap [1,6] = [1,2]$, and then $[-3,2] \setminus [1,2]$ and $[1,6] \setminus [1,2]$. Unfortunately, there are lattices that are not closed under complementation. As a consequence, determinization of an LTA does not preserve the recognized language.

The solution proposed in [24] for word automata is to use a finite partition of the lattice Λ, which commands when two transitions should be merged using the *lub* operator. The fusion of transitions may induce an over-approximation controlled by the fineness of the partition.

Partitioned *LTA*. Π is a *partition* of an atomic lattice Λ if $\Pi \subseteq 2^{\Lambda}$ and $\forall \pi_1, \pi_2 \in \Pi$, $\pi_1 \sqcap \pi_2 = \bot$, and $\forall a \in Atoms(\Lambda), \exists \pi \in \Pi : a \sqsubseteq \pi$. As an example, if Λ is the lattice of intervals, we can have a partition $\Pi = \{]-\infty, 0[, [0,0],]0, +\infty[\}$.

Definition 5 (Partitioned lattice tree automaton (PLTA)). *A PLTA $\mathcal{A}$ is an LTA $\mathcal{A} = \langle \Pi, \mathcal{Q}, \mathcal{F}, \mathcal{Q}_f, \Delta \rangle$ equipped with a partition Π, such that for all lambda transitions $\lambda \to q \in \Delta$, $\exists \pi \in \Pi$ such that $\lambda \sqsubseteq \pi$.*

A PLTA is merged if $\lambda_1 \to q$, $\lambda_2 \to q \in \Delta \wedge \lambda_1 \sqsubseteq \pi_1 \wedge \lambda_2 \sqsubseteq \pi_2 \implies \pi_1 \sqcap \pi_2 = \emptyset$, where $\lambda_1, \lambda_2 \in \Lambda$ and $\pi_1, \pi_2 \in \Pi$.

For example, if $\Pi = \{]-\infty, 0[, [0,0],]0, +\infty[\}$, a PLTA can have the following transition rules : $[-3,-1] \to q_1$, $[-5,-2] \to q_2$, $[3,4] \to q_4$. This PLTA is not merged because of the two lambda transitions $[-3,-1] \to q_1$ and $[-5,-2] \to q_2$, because $[-3,-1]$ and $[-5,-2]$ are in the same partition. The merged corresponding one will have the following transition : $[-5,-1] \to q_{1,2}$, instead of the two transitions mentionned before.

Any LTA $\mathcal{A}$ can be turned into a PLTA $\mathcal{A}_p$ the following way : Let Π be the partition. For any lambda transition $\lambda \to q \in \mathcal{A}$, if $\exists \pi_1, \ldots, \pi_n \in \Pi$ such that $\lambda \sqcap \pi_1 \neq \emptyset, \ldots, \lambda \sqcap \pi_n \neq \emptyset$, where $\pi_1 \neq \ldots \neq \pi_n$, the transition $\lambda \to q$ will be replaced by n transitions $\lambda \sqcap \pi_1 \to q, \ldots, \lambda \sqcap \pi_n \to q$ in $\mathcal{A}_p$.

Example 4. Let $\mathcal{A} = \langle \mathcal{Q}, \mathcal{F}, \mathcal{Q}_f, \Delta \rangle$ be an LTA such that $\Delta = \{[3,4] \to q_1, [-3,2] \to q_2, f(q_1,q_2) \to q_f\}$, and $\Pi = \{]-\infty, 0[, [0,0],]0, +\infty[\}$ be a partition. Then the corresponding PLTA is $\mathcal{A}_p = \langle \mathcal{Q}, \mathcal{F}, \mathcal{Q}_f, \Delta_p \rangle$, where $\Delta_p = \{[3,4] \to q_1, [-3,0[\to q_2, [0,0] \to q_2,]0,2] \to q_2, f(q_1,q_2) \to q_f\}$.

Two lambda transitions $\lambda_1 \to q$, $\lambda_2 \to q$ of a PLTA can not be merged if λ_1 and λ_2 belong to different elements of the partition, whereas they might be merged in the opposite case.

Proposition 1 (Equivalence between LTA and PLTA). *Given an LTA $\mathcal{A} = \langle \mathcal{Q}, \mathcal{F}, \mathcal{Q}_f, \Delta \rangle$ and a partition Π, there exists a PLTA $\mathcal{A}' = \langle \Pi, \mathcal{Q}, \mathcal{F}, \mathcal{Q}_f, \Delta' \rangle$ recognizing the same language.*

Proof. $\mathcal{A}'$ is obtained from $\mathcal{A}$ by replacing each lambda transition $\lambda \to q \in \Delta$ by at most n_Π transitions $\lambda_i \to q$ where $\lambda_i = \lambda \sqcap \pi_i$, $\pi_i \in \Pi$, such that $\bigsqcup \lambda_i = \lambda$.

Any PLTA $\mathcal{A} = \langle \Pi, \mathcal{Q}, \mathcal{F}, \mathcal{Q}_f, \Delta \rangle$ can be transformed into a merged PLTA $\mathcal{A}_m = \langle \Pi, \mathcal{Q}, \mathcal{F}, \mathcal{Q}_f, \Delta m \rangle$ such that $\mathcal{L}(\mathcal{A}) \subseteq \mathcal{L}(\mathcal{A}_m)$ by merging transitions as follows :

$$\frac{q \in \mathcal{Q} \quad \pi \in \Pi \quad \lambda_m = \bigsqcup \{\lambda \sqcap \pi, \lambda \in \Lambda | \lambda \to q \in \Delta\}}{\lambda_m \to q \in \Delta_m}$$

Example 5. If $\mathcal{A} = \langle \Pi, \mathcal{Q}, \mathcal{F}, \mathcal{Q}_f, \Delta \rangle$, where $\Pi =]-\infty, 0[, [0, +\infty$ and $\Delta = \{[0,2] \to q_1, [5,8] \to q_2, [-3,-2] \to q_3, [-4,-1] \to q_4, h(q_1,q_2,q_3,q_4) \to q_f\}$, the merged automaton $\mathcal{A}_m = \langle \Pi, \mathcal{Q}, \mathcal{F}, \mathcal{Q}_f, \Delta m \rangle$ corresponding to $\mathcal{A}$ has the following transitions: $\Delta_m = \{[0,8] \to q_{1,2}, [-4,-1] \to q_{3,4}, h(q_{1,2}, q_{1,2}, q_{3,4}, q_{3,4}) \to q_f\}$.

We are now ready to sketch the determinization algorithm. The determinization of a PLTA, which transforms a PLTA $\mathcal{A}$ to a merged Deterministic Partitioned LTA $\mathcal{A}_d$ according to a partition Π, mimics the one on usual TA. The difference is that two λ-transitions

$\lambda_1 \to q_1$ and $\lambda_2 \to q_2$ are merged in $\lambda_1 \sqcap \lambda_2 \to \{q_1, q_2\}$ when λ_1 and λ_2 are included in the same element π of the partition Π. Consequently, the resulting automaton recognizes a larger language : $\mathcal{L}(\mathcal{A}) \subseteq \mathcal{L}(\mathcal{A}_d)$.This algorithm produces the best approximation in term of inclusion of languages.

```
Determinization Algorithm :
input: PLTA $\mathcal{A} = \langle \Pi, \mathcal{Q}, \mathcal{F}, \mathcal{Q}_f, \Delta \rangle$
begin
    $\mathcal{Q}_d := \emptyset$; $\Delta_d = \emptyset$;
    for all $\pi \in \Pi$ do
        $Trans(\pi) := \{\lambda \to q \in \Delta | \lambda \in \Lambda, \lambda \sqsubseteq \pi\}$;
        $s := \{q \in \mathcal{Q} | \lambda \to q \in Trans(\pi)\}$;
        $\mathcal{Q}_d := \mathcal{Q}_d \cup \{s\}$;
        $\lambda_m := \bigsqcup\{\lambda | \lambda \to q \in Trans(\pi)\}$;
        $\Delta_d := \Delta_d \cup \{\lambda_m \to s\}$;
    end for
    repeat
        Let $f \in \mathcal{F}_n$, $s_1, \ldots, s_n \in \mathcal{Q}_d$,
        $s := \{q \in \mathcal{Q} | \exists q_1 \in s_1, \ldots, q_n \in s_n, f(q_1, \ldots, q_n) \to q \in \Delta\}$;
        $\mathcal{Q}_d := \mathcal{Q}_d \cup \{s\}$;
        $\Delta_d := \Delta_d \cup \{f(s_1, \ldots, s_n) \to s\}$;
    until no more rule can be added to $\Delta_d$
$\mathcal{Q}_{df} := \{s \in \mathcal{Q}_d | s \cap \mathcal{Q}_f \neq \emptyset\}$
output merged DPLTA $\mathcal{A}_d = \langle \Pi, \mathcal{Q}_d, \mathcal{F}, \mathcal{Q}_{df}, \Delta_d \rangle$
end
```

Example 6. Let $\Delta = \{[-3,-1] \to q_1, [-5,-2] \to q_2, [3,4] \to q_3, [-3,2] \to q_4, f(q_1,q_2) \to q_5, f(q_3,q_4) \to q_6, f(q_5,q_6) \to q_{f1}, f(q_5,q_6) \to q_{f2}\}$, and $\Pi = \{]-\infty,0[, [0,0],]0,+\infty[\}$

With the determinization algorithm defined above, we obtain this set of transition for the deterministic corresponding PLTA : $\Delta_d = \{, [-5,0[\to q_{1,2,4},]0,4] \to q_{3,4}, [0,0] \to q_4, f(q_{1,2,4}, q_{1,2,4}) \to q_5, f(q_{3,4}, q_{3,4}) \to q_6, f(q_{3,4}, q_4) \to q_6, f(q_{3,4}, q_{1,2,4}) \to q_6, f(q_5, q_6) \to q_{f1,f2}\}$.

Proposition 2. *Deterministic PLTA is the best upper-approximation*

Let $\mathcal{A}_1$ be a PLTA and $\mathcal{A}_2$ the PLTA obtained with the determinization algorithm. Then $\mathcal{A}_2$ is a best upper-approximation of $\mathcal{A}_1$ as a merged and deterministic PLTA.

1. $\mathcal{L}(\mathcal{A}_1) \subseteq \mathcal{L}(\mathcal{A}_2)$
2. *For any merged and deteministic PLTA $\mathcal{A}_3$ based on the same partition as $\mathcal{A}_1$, $\mathcal{L}(\mathcal{A}_1) \subseteq \mathcal{L}(\mathcal{A}_3) \implies \mathcal{L}(\mathcal{A}_2) \subseteq \mathcal{L}(\mathcal{A}_3)$*

Proof (Proposition 2).

(1) Base case : for all lambda transitions of $\mathcal{A}_1$ $\lambda \rightarrow q$, let $\pi \in \varPi$ such that $\lambda \sqsubseteq \pi$. Then $Trans(\pi) = \{\lambda \rightarrow q \in \varDelta | \lambda \in \varLambda, \lambda \sqsubseteq \pi\}$. Then there is a transition $\lambda' \rightarrow Q$ in $\mathcal{A}_2$ such that $\lambda' = \bigsqcup\{\lambda | \lambda \rightarrow q \in Trans(\pi)\}$ and $Q = \{q | \lambda \rightarrow q \in Trans(\pi)\}$, so $q \in Q$.
induction case : for all non lambda transition of $\mathcal{A}_1$ $f(q_1, \ldots, q_n) \rightarrow q$, there is the corresponding transition $f(Q_1, \ldots, Q_n) \rightarrow Q$ such that $q \in Q$. We have $q_1 \in Q_1, \ldots, q_n \in Q_n$ thanks to the induction hypothesis.
So $\mathcal{L}(\mathcal{A}_1) \subseteq \mathcal{L}(\mathcal{A}_2)$.$\square$

(2) $\mathcal{A}_1 = \langle \varPi, \mathcal{Q}_1, \mathcal{F}, \mathcal{Q}_{f_1}, \varDelta_1 \rangle$, $\mathcal{A}_2 = \langle \varPi, \mathcal{Q}_2, \mathcal{F}, \mathcal{Q}_{f_2}, \varDelta_2 \rangle$ and $\mathcal{A}_3 = \langle \varPi, \mathcal{Q}_3, \mathcal{F}, \mathcal{Q}_{f_3}, \varDelta_3 \rangle$

As $\mathcal{L}(\mathcal{A}_1) \subseteq \mathcal{L}(\mathcal{A}_2)$ (1) and $\mathcal{L}(\mathcal{A}_1) \subseteq \mathcal{L}(\mathcal{A}_3)$, let $\mathcal{R}_1 : \mathcal{Q}_1 \times \mathcal{Q}_2$ and $\mathcal{R}_2 : \mathcal{Q}_1 \times \mathcal{Q}_3$ be two simulation relations defining these properties as follows.
Let $q_1 \in \mathcal{Q}_1$ and $q_2 \in \mathcal{Q}_2$, $(q_1, q_2) \in \mathcal{R}_1$ iff

- $\lambda_1 \rightarrow q_1 \in \varDelta_1,\ \lambda_2 \rightarrow q_2 \in \varDelta_2$ and $\lambda_1 \sqsubseteq \lambda_2$, where $\lambda_1, \lambda_2 \in \varLambda$,
  or
  $f(q_{i_1}, \ldots, q_{i_n}) \rightarrow q_1 \in \varDelta_1,\ f(q'_{i_1}, \ldots, q'_{i_n}) \rightarrow q_2 \in \varDelta_2$ and $\forall j \in [1, n],\ (q_{i_j}, q'_{i_j}) \in \mathcal{R}_1$, where $f \in \mathcal{F}_n$
- $q_1 \in \mathcal{Q}_{f_1} \Longleftrightarrow q_2 \in \mathcal{Q}_{f_2}$

Let $q_1 \in \mathcal{Q}_1$ and $q_3 \in \mathcal{Q}_3$, $(q_1, q_3) \in \mathcal{R}_2$ iff

- $\lambda_1 \rightarrow q_1 \in \varDelta_1,\ \lambda_3 \rightarrow q_3 \in \varDelta_3$ and $\lambda_1 \sqsubseteq \lambda_3$, where $\lambda_1, \lambda_3 \in \varLambda$,
  or
  $f(q_{i_1}, \ldots, q_{i_n}) \rightarrow q_1 \in \varDelta_1,\ f(q'_{i_1}, \ldots, q'_{i_n}) \rightarrow q_3 \in \varDelta_2$ and $\forall j \in [1, n],\ (q_{i_j}, q'_{i_j}) \in \mathcal{R}_2$, where $f \in \mathcal{F}_n$
- $q_1 \in \mathcal{Q}_{f_1} \Longleftrightarrow q_3 \in \mathcal{Q}_{f_3}$

Let $\mathcal{R} : \mathcal{Q}_2 \times \mathcal{Q}_3$ be a simulation relation such that $(q_2, q_3) \in \mathcal{R}$ iff $\exists q_1 \in \mathcal{Q}_1.(q_1, q_2) \in \mathcal{R}_1 \land (q_1, q_3) \in \mathcal{R}_2$, where $q_2 \in \mathcal{Q}_2$, $q_3 \in \mathcal{Q}_3$.

Let $(q_2, q_3) \in \mathcal{R}$. This means that :

- $\lambda_1 \rightarrow q_1 \in \varDelta_1,\ \lambda_2 \rightarrow q_2 \in \varDelta_2,\ \lambda_3 \rightarrow q_3 \in \varDelta_2$ and $\lambda_1 \sqsubseteq \lambda_2$ and $\lambda_1 \sqsubseteq \lambda_3$, where $\lambda_1, \lambda_2, \lambda_3 \in \varLambda$ (a)
  , or
  $f(q_{i_1}, \ldots, q_{i_n}) \rightarrow q_1 \in \varDelta_1,\ f(q'_{i_1}, \ldots, q'_{i_n}) \rightarrow q_2 \in \varDelta_2,\ f(q''_{i_1}, \ldots, q''_{i_n}) \rightarrow q_3 \in \varDelta_3$ and $\forall j \in [1, n],\ (q_{i_j}, q'_{i_j}) \in \mathcal{R}_1$ and $(q_{i_j}, q''_{i_j}) \in \mathcal{R}_2$, where $f \in \mathcal{F}_n$ (b)
- $q_1 \in \mathcal{Q}_{f_1} \Longleftrightarrow q_2 \in \mathcal{Q}_{f_2}$ and $q_1 \in \mathcal{Q}_{f_1} \Longleftrightarrow q_3 \in \mathcal{Q}_{f_3}$ (c),

by definition of $\mathcal{R}_1$ and $\mathcal{R}_2$.

(a) Let $\pi \in \Pi$ be the element of the partition such that $\lambda_1 \sqsubseteq \pi$. Then $Trans(\pi) = \{\lambda \to q \in \Delta | \lambda \in \Lambda, \lambda \sqsubseteq \pi\}$, i.e the set of all the lambda transitions $\lambda \to q$ in Δ_1 such that $\lambda \sqsubseteq \pi$. Of course $\lambda_1 \sqsubseteq Trans(\pi)$, because $\lambda_1 \sqsubseteq \pi$. Then λ_2 is the least upper bound of all $\lambda \in \Lambda$ such that $\lambda \to q \in Trans(\pi)$, i.e $\lambda_2 = \bigsqcup\{\lambda | \lambda \to q \in Trans(\pi)\}$, according to the determinization algorithm.

As $\mathcal{A}_3$ is deterministic and contains $\mathcal{A}_1$, then λ_3 has to contain at least all the $\lambda \in \Lambda$ such that $\lambda \to q \in \Delta_1$ and $\lambda \sqsubseteq \pi$, or else $\mathcal{A}_3$ is not deterministic.

So $\lambda_3 \sqsupseteq \bigsqcup\{\lambda | \lambda \to q \in Trans(\pi)\}$, so $\lambda_2 \sqsubseteq \lambda_3$.

(b) We can immediately deduce that $\forall j \in [1, n]$, $(q'_{i_j}, q''_{i_j}) \in \mathcal{R}$ by the definition of $\mathcal{R}$.

(c) So $q_2 \in \mathcal{Q}_{f_2} \iff q_3 \in \mathcal{Q}_{f_3}$

And thanks to these properties deduced on $\mathcal{R} : \mathcal{Q}_1 \times \mathcal{Q}_2$, we can deduce that $\mathcal{L}(\mathcal{A}_2) \subseteq \mathcal{L}(\mathcal{A}_3)$.

As the least upper bound of two elements of a lattice is the best and unique upper-approximation, this determinization algorithm returns the best upper-approximation.$\square$

3.5 Minimization

To define the minimization algorithm, we first have to define a *Refine* recursive algorithm which refines an equivalence relation P on states, according to the PLTA $\mathcal{A}$.

```
Refine($P, \mathcal{A}$)
begin
    Let $P'$ be a new equivalence relation;
    For all $(q, q') \in \mathcal{Q}$ such that $qPq'$ do
        IF $(\forall f \in \mathcal{F}^n$,
            $\Delta(f(q_1, \ldots, q_{i-1}, q, q_{i+1}, \ldots, q_n)) P \Delta(f(q_1, \ldots, q_{i-1}, q', q_{i+1}, \ldots, q_n))$,
            where $q_1, \ldots, q_{i-1}, q_{i+1}, \ldots, q_n \in \mathcal{Q})$
            AND $(\forall a \in \mathcal{F}^0_\circ,\ a \to q \Rightarrow a \to q')$
            AND $(\forall \lambda_1, \lambda_2 \in \Lambda,\ \exists \pi \in \Pi$
                such that $\lambda_1 \to q \Rightarrow \lambda_2 \to q'$ and $\lambda_1, \lambda_2 \in \pi)$
        THEN $qP'q$
        ELSE if $P = \{\mathcal{Q}_1, \ldots, \mathcal{Q}_i, \ldots, \mathcal{Q}_n\}$ and $q, q' \in \mathcal{Q}_i$
            then $P := \{\mathcal{Q}_1, \ldots, \mathcal{Q}_{i-1}, \mathcal{Q}_{i_1}, \mathcal{Q}_{i_2}, \mathcal{Q}_{i+1}, \ldots, \mathcal{Q}_n\}$;
                $q \in \mathcal{Q}_{i_1}$; $q' \in \mathcal{Q}_{i_2}$;
                Refine($P'$);
end
```

We are now ready to define the minimization algorithm of a PLTA $\mathcal{A}$.

```
**MinimizationAlgorithm($\mathcal{A}$)**
**input:** Determinized $P$LTA $\mathcal{A} = \langle \Pi, \mathcal{Q}, \mathcal{F}, \mathcal{Q}_f, \Delta \rangle$
      An equivalence relation $P = \{\mathcal{Q}_f, \mathcal{Q} \setminus \mathcal{Q}_f\}$
**output:** Minimized and determinized $P$LTA $\mathcal{A}_{\Updownarrow} = \langle \Pi, \mathcal{Q}_m, \mathcal{F}, \mathcal{Q}_{f_m}, \Delta_m \rangle$
**begin**
    Refine($P$, $\mathcal{A}$);
    Set $\mathcal{Q}_m$ to the set of equivalence classes of $P$;
    /* we denote by $[q]$ the equivalence class of state $q$ w.r.t. $P$ */
    For all $\lambda$-transitions, for all $\lambda_1, \lambda_2 \in \Lambda$,
        if $\lambda_1 \to q$, $\lambda_2 \to q' \in \Delta$ and $qPq'$
        then $\lambda_1 \sqcup \lambda_2 \to [q, q'] \in \Delta_m$;
    For all other transitions, $\Delta_m := \{(f, [q_1], \ldots, [q_n]) \to [f(q_1, \ldots, q_n)]\}$;
    $\mathcal{Q}_{m_f} := \{[q] | q \in \mathcal{Q}_f\}$;
**end**
```

A **normalized PLTA** is an LTA that is a merged, deterministic and minimized PLTA.

Proposition 3. *Normalized PLTA is the best upper-approximation Let $\mathcal{A}_1$ be a PLTA and $\mathcal{A}_2$ the PLTA obtained with the minimization algorithm. Then $\mathcal{A}_2$ is a best upper-approximation of $\mathcal{A}_1$ as a normalized PLTA.*

1. $\mathcal{L}(\mathcal{A}_1) \subseteq \mathcal{L}(\mathcal{A}_2)$
2. *For any normalized PLTA $\mathcal{A}_3$ based on the same partition as $\mathcal{A}_1$,* $\mathcal{L}(\mathcal{A}_1) \subseteq \mathcal{L}(\mathcal{A}_3) \implies \mathcal{L}(\mathcal{A}_2) \subseteq \mathcal{L}(\mathcal{A}_3)$

Proof :

Let P be the equivalence relation at the end of the minimization algorithm.

(1) Base case : for all lambda transitions of $\mathcal{A}_1$ $\lambda \to q$, there is a transition $\lambda' \to [q]$ in $\mathcal{A}_2$ such that $\lambda' = \bigsqcup\{\lambda | \lambda \to q' \in \Delta_1 \wedge q'Pq\}$.
induction case : for all non lambda transitions of $\mathcal{A}_1$ $f(q_1, \ldots, q_n) \to q$, there is the corresponding transition $f([q_1], \ldots, [q_n]) \to [q]$ (where $q \in [q]$, $q_1 \in [q_1], \ldots, q_n \in [q_n]$).
So $\mathcal{L}(\mathcal{A}_1) \subseteq \mathcal{L}(\mathcal{A}_2)$.$\square$

(2) $\mathcal{A}_1 = \langle \Pi, \mathcal{Q}_1, \mathcal{F}, \mathcal{Q}_{f_1}, \Delta_1 \rangle$, $\mathcal{A}_2 = \langle \Pi, \mathcal{Q}_2, \mathcal{F}, \mathcal{Q}_{f_2}, \Delta_2 \rangle$ and $\mathcal{A}_3 = \langle \Pi, \mathcal{Q}_3, \mathcal{F}, \mathcal{Q}_{f_3}, \Delta_3 \rangle$

As $\mathcal{L}(\mathcal{A}_1) \subseteq \mathcal{L}(\mathcal{A}_2)$ (1) and $\mathcal{L}(\mathcal{A}_1) \subseteq \mathcal{L}(\mathcal{A}_3)$, let $\mathcal{R}_1 : \mathcal{Q}_1 \times \mathcal{Q}_2$ and $\mathcal{R}_2 : \mathcal{Q}_1 \times \mathcal{Q}_3$ be two simulation relations defining these properties as follows.
Let $q_1 \in \mathcal{Q}_1$ and $q_2 \in \mathcal{Q}_2$, $(q_1, q_2) \in \mathcal{R}_1$ iff

- $\lambda_1 \rightarrow q_1 \in \Delta_1,\ \lambda_2 \rightarrow q_2 \in \Delta_2$ and $\lambda_1 \sqsubseteq \lambda_2$, where $\lambda_1, \lambda_2 \in \Lambda$,
  or
  $f(q_{i_1}, \ldots, q_{i_n}) \rightarrow q_1 \in \Delta_1,\ f(q'_{i_1}, \ldots, q'_{i_n}) \rightarrow q_2 \in \Delta_2$ and $\forall j \in [1,n],\ (q_{i_j}, q'_{i_j}) \in \mathcal{R}_1$, where $f \in \mathcal{F}_n$
- $q_1 \in \mathcal{Q}_{f_1} \Longleftrightarrow q_2 \in \mathcal{Q}_{f_2}$

Let $q_1 \in \mathcal{Q}_1$ and $q_3 \in \mathcal{Q}_3$, $(q_1, q_3) \in \mathcal{R}_2$ iff

- $\lambda_1 \rightarrow q_1 \in \Delta_1,\ \lambda_3 \rightarrow q_3 \in \Delta_3$ and $\lambda_1 \sqsubseteq \lambda_3$, where $\lambda_1, \lambda_3 \in \Lambda$,
  or
  $f(q_{i_1}, \ldots, q_{i_n}) \rightarrow q_1 \in \Delta_1,\ f(q'_{i_1}, \ldots, q'_{i_n}) \rightarrow q_3 \in \Delta_2$ and $\forall j \in [1,n],\ (q_{i_j}, q'_{i_j}) \in \mathcal{R}_2$, where $f \in \mathcal{F}_n$
- $q_1 \in \mathcal{Q}_{f_1} \Longleftrightarrow q_3 \in \mathcal{Q}_{f_3}$

Let $\mathcal{R} : \mathcal{Q}_2 \times \mathcal{Q}_3$ be a simulation relation such that $(q_2, q_3) \in \mathcal{R}$ iff $\exists q_1 \in \mathcal{Q}_1.(q_1, q_2) \in \mathcal{R}_1 \wedge (q_1, q_3) \in \mathcal{R}_2$, where $q_2 \in \mathcal{Q}_2$, $q_3 \in \mathcal{Q}_3$.

Let $(q_2, q_3) \in \mathcal{R}$. This means that :

- $\lambda_1 \rightarrow q_1 \in \Delta_1,\ \lambda_2 \rightarrow q_2 \in \Delta_2,\ \lambda_3 \rightarrow q_3 \in \Delta_2$ and $\lambda_1 \sqsubseteq \lambda_2$ and $\lambda_1 \sqsubseteq \lambda_3$, where $\lambda_1, \lambda_2, \lambda_3 \in \Lambda$ (a)
  , or
  $f(q_{i_1}, \ldots, q_{i_n}) \rightarrow q_1 \in \Delta_1,\ f(q'_{i_1}, \ldots, q'_{i_n}) \rightarrow q_2 \in \Delta_2,\ f(q''_{i_1}, \ldots, q''_{i_n}) \rightarrow q_3 \in \Delta_3$ and $\forall j \in [1,n],\ (q_{i_j}, q'_{i_j}) \in \mathcal{R}_1$ and $(q_{i_j}, q''_{i_j}) \in \mathcal{R}_2$, where $f \in \mathcal{F}_n$ (b)

- $q_1 \in \mathcal{Q}_{f_1} \Longleftrightarrow q_2 \in \mathcal{Q}_{f_2}$ and $q_1 \in \mathcal{Q}_{f_1} \Longleftrightarrow q_3 \in \mathcal{Q}_{f_3}$ (c),

by definition of $\mathcal{R}_1$ and $\mathcal{R}_2$.

(a) We have $\lambda_1 \rightarrow q_1 \in \Delta_1$, $\lambda_2 \rightarrow q_2 \in \Delta_2$ and $\lambda_1 \sqsubseteq \lambda_2$. According to the minization algorithm, λ_2 is the least upper bound of all $\lambda \in \Lambda$ such that there exists $q \in \mathcal{Q}_1$ such that $\lambda \rightarrow q \in \Delta_1$ and q is in the same equivalence classe as q_1 (i.e., $q \in [q_1]$ or qPq_1). Formally, $\lambda_2 = \bigsqcup\{\lambda | \lambda \rightarrow q \in \Delta_1 \wedge qPq_1\}$.
As $\mathcal{A}_3$ is minimized and contains $\mathcal{A}_1$, then λ_3 has to contain at least all the $\lambda \in \Lambda$ such that $\lambda \rightarrow q \in \Delta_1$ and qPq_1, or else $\mathcal{A}_3$ is not minimized.
So $\lambda_3 \sqsupseteq \bigsqcup\{\lambda | \lambda \rightarrow q \in \Delta_1 \wedge qPq_1\}$, so $\lambda_2 \sqsubseteq \lambda_3$.

(b) We can immediately deduce that $\forall j \in [1,n],\ (q'_{i_j}, q''_{i_j}) \in \mathcal{R}$ by the definition of $\mathcal{R}$.

(c) So $q_2 \in \mathcal{Q}_{f_2} \Longleftrightarrow q_3 \in \mathcal{Q}_{f_3}$

And thanks to these properties deduced on $\mathcal{R} : \mathcal{Q}_1 \times \mathcal{Q}_2$, we can deduce that $\mathcal{L}(\mathcal{A}_2) \subseteq \mathcal{L}(\mathcal{A}_3)$.

As the least upper bound of two elements of a lattice is the best and unique upper-approximation, this minimization algorithm returns the best upper-approximation.$\square$

3.6 Refinement of the partition

In the previous paragraphs, the partition Π was fixed. The precision of the upper-approximations made during the determinization algorithm depends on the finess of Π. For example, if Π is of size 1, all λ-transitions will be merged into one.

Definition 6 (Refinement of a partition).
A partition Π_2 refines a partition Π_1 if :

$$\forall \pi_2 \in \Pi_2,\ \exists \pi_1 \in \Pi_1 :\ \pi_2 \sqsubseteq \pi_1$$

Let $\mathcal{A}_1 = \langle \Pi_1, \mathcal{Q}, \mathcal{F}, \mathcal{Q}_f, \Delta_1 \rangle$ be a PLTA. The PLTA $\mathcal{A}_2 = \langle \Pi_2, \mathcal{Q}, \mathcal{F}, \mathcal{Q}_f, \Delta_2 \rangle$ refines $\mathcal{A}_1$ if :

1. *Π_2 refines Π_1*
2. *the transitions of Δ_2 are obtained by : $\forall \lambda \rightarrow q \in \Delta_1,\ \forall \pi_2 \in \Pi_2,\ \lambda \sqcap \pi_2 \rightarrow q \in \Delta_2$*

Refining an automaton does not modify immediatly the recognized language, but leads to a more precise upper-approximation in the determinization, as illustrated herafter.

Example 7. Given Π and Δ of example 6 and a partition $\Pi_2 = \{]-\infty,-1[, [-1,0[, [0,0],]0,+\infty[\}$ that refines Π, the set of transitions Δ_2 of PLTA obtained with Π_2 is $\Delta_2 = \{[-3,-1[\rightarrow q_1, [-1,-1] \rightarrow q_1, [-5,-2] \rightarrow q_2, [3,4] \rightarrow q_3, [-3,-1[\rightarrow q_4, [-1,0[\rightarrow q_4, [0,0] \rightarrow q_4,]0,2] \rightarrow q_4, f(q_1,q_2) \rightarrow q_5, f(q_3,q_4) \rightarrow q_6, f(q_5,q_6) \rightarrow q_{f1}, f(q_5,q_6) \rightarrow q_{f2}\}$.
We now obtain this set of transitions for the deterministic corresponding PLTA with $\Pi_2 : \Delta_{2_d} = \{[-5,-1[\rightarrow q_{1,2,4}, [-1,0[\rightarrow q_{1,4},]0,4] \rightarrow q_{3,4}, [0,0] \rightarrow q_4, f(q_{1,2,4},q_{1,2,4}) \rightarrow q_5, f(q_{1,4},q_{1,2,4}) \rightarrow q_5, f(q_{3,4},q_{3,4}) \rightarrow q_6, f(q_{3,4},q_4) \rightarrow q_6, f(q_{3,4},q_{1,2,4}) \rightarrow q_6, f(q_{3,4},q_{1,4}) \rightarrow q_6, f(q_5,q_6) \rightarrow q_{f1,f2}\}$.

4 A Completion Algorithm for LTA

We are interested in computing the set of reachable states of an infinite state system. In general this set is neither representable nor computable. In this paper, we suggest to work within the Tree Regular Model Checking framework for representing possibly infinite sets of state. More precisely, we propose to represent configurations by (built-in)terms and set of configurations (or set of states) by an LTA.

In addition, we assume that the behavior of the system can be represented by conditional term rewriting systems (TRS), that are term rewriting systems equipped with conjunction of conditions used to restrain the applicability of the rule. Our conditional TRS, which extends the classical definition of [7], rewrites terms defined on the concrete domain. This makes them independent from the abstract lattice. We first start with the definition of predicates that allows us to express conditions on TRS.

Definition 7 (Predicates). *Let $\mathcal{P}$ be the set of predicates over $\mathcal{D}$. For instance if ρ is a n-ary predicate of $\mathcal{P}$ then $\rho : \mathcal{D}^n \mapsto \{true, false\}$. We extend the domain of ρ to $\mathcal{T}(\mathcal{F}, \mathcal{X})^n$ in the following way:*

$$\rho(t_1, \dots, t_n) = \begin{cases} \rho(u_1, \dots, u_n) \text{ if } \forall i = 1 \dots n : t_i \in \mathcal{T}(\mathcal{F}_\bullet) \\ \qquad\qquad \text{where } \forall i = 1 \dots n : u_i = eval(t_i) \\ false \text{ if } \exists j = 1 \dots n : t_j \notin \mathcal{T}(\mathcal{F}_\bullet) \end{cases}$$

Observe that predicates are defined on built-in terms of the concrete domain. If one of the predicate parameters cannot be evaluated into a built-in term, then the predicate returns false and the rule is not applied.

Definition 8 (Conditional Term Rewriting System on $\mathcal{T}(\mathcal{F}_\circ \cup \mathcal{F}_\bullet, \mathcal{X})$). *In our setting, a Term Rewriting System (TRS) $\mathcal{R}$ is a set of* rewrite rules $l \rightarrow r \Leftarrow c_1 \wedge \dots \wedge c_n$, *where $l \in \mathcal{T}(\mathcal{F}_\circ, \mathcal{X})$, $r \in \mathcal{T}(\mathcal{F}_\circ \cup \mathcal{F}_\bullet, \mathcal{X})$, $l \notin \mathcal{X}$, $\mathcal{V}ar(l) \supseteq \mathcal{V}ar(r)$ and $\forall i = 1 \dots n : c_i = \rho_i(t_1, \dots, t_m)$ where ρ_i is a m-ary predicate of $\mathcal{P}$ and $\forall j = 1 \dots m : t_j \in \mathcal{T}(\mathcal{F}_\bullet, \mathcal{X}) \wedge \mathcal{V}ar(t_j) \subseteq \mathcal{V}ar(l)$.*

Example 8. Using conditional rewriting rules, the factorial can be encoded as follows:

$$\begin{array}{l} fact(x) \rightarrow 1 \Leftarrow x \geq 0 \wedge x \leq 1 \\ fact(x) \rightarrow x * fact(x-1) \Leftarrow x \geq 2 \end{array}$$

In what follows, we will use different types of substitutions. Let $\mathcal{X}$ a set of variables, $\mathcal{Q}$ a set of states, and $\mathcal{F}$ a set of symbols, a substitution σ is a function $\sigma : \mathcal{X} \mapsto \mathcal{Q} \cup \mathcal{T}(\mathcal{F})$ that can be extended to $\mathcal{T}(\mathcal{F}, \mathcal{X})$ in this way; for all $t \in \mathcal{T}(\mathcal{F}, \mathcal{X})$, we define $t\sigma$ as:

1. if $t = f(t_1, \dots, t_n)$ then $t\sigma = f(t_1\sigma, \dots, t_n\sigma)$, where $t, t_1, \dots, t_n \in \mathcal{T}(\mathcal{F}, \mathcal{X}), f \in \mathcal{F}^n$,
2. if $t = x \in \mathcal{X}$ then $t\sigma = \sigma(x)$.

Let $\mathcal{F} = \mathcal{F}_\circ \cup \mathcal{F}_\bullet$, the TRS $\mathcal{R}$ and the *eval* function induces a rewriting relation $\rightarrow_\mathcal{R}$ on $\mathcal{T}(\mathcal{F})$ in the following way : for all $s, t \in \mathcal{T}(\mathcal{F})$, we have $s \rightarrow_\mathcal{R} t$ if there exist :

1. a rewrite rule $l \rightarrow r \Leftarrow c_1 \wedge \dots \wedge c_n \in \mathcal{R}$,
2. a position $p \in \mathcal{P}os(s)$, and
3. a substitution $\sigma : \mathcal{X} \mapsto \mathcal{T}(\mathcal{F})$ such that $s|_p = l\sigma$, $t = eval(s[r\sigma]_p)$ and $\forall i = 1 \dots n : c_i\sigma = true$.

The reflexive transitive closure of $\rightarrow_\mathcal{R}$ is denoted by $\rightarrow_\mathcal{R}^*$.

Let $\mathcal{A}$ be an LTA representing the set of initial states, and $\mathcal{R}$ be a rewriting system. Our objective is to compute another LTA representing the set (or an over-approximation of the set) $\mathcal{R}^*(\mathcal{L}(\mathcal{A})) = \{t | \exists t_0 \in \mathcal{L}(\mathcal{A}), t_0 \rightarrow^*_{\mathcal{R}} t\}$. In this paper, we adopt the completion approach of [27,22], which intends to compute a tree automaton $\mathcal{A}^k_{\mathcal{R}}$ such that $\mathcal{L}(\mathcal{A}^k_{\mathcal{R}}) \supseteq \mathcal{R}^*(\mathcal{L}(\mathcal{A}))$ for a left-linear TRS $\mathcal{R}$ (see background). The algorithm proceeds by computing the sequence of automata $\mathcal{A}^0_{\mathcal{R}}, \mathcal{A}^1_{\mathcal{R}}, \mathcal{A}^2_{\mathcal{R}}, ...$ that represents successive applications of $\mathcal{R}$. Computing $\mathcal{A}^{i+1}_{\mathcal{R}}$ from $\mathcal{A}^i_{\mathcal{R}}$ is called a *one-step completion*. In general the sequence of automata may not converge in a finite amount of time. To accelerate the convergence, we perform an abstraction operation which accelerates the computation. Our abstraction relies on merging states that are considered to be equivalent with respect to a certain equivalence relation defined by a set of equations. Depending on the objective, equations can either be defined by hand (e.g. [31]), or automatically generated from a static analysis of the TRS (e.g. [13]). Note that those approximation can automatically be refined using counterexample guided abstraction refinement ([12]). We now give details on the above constructions. Then, we show that, in order to be correct, our procedure has to be combined with an evaluation that may add new terms to the language of the automaton obtained by completion or equational abstraction. We shall see that this closure property may add an infinite number of transitions whose behavior is captured with a new widening operator for LTA.

4.1 Computation of $\mathcal{A}^{i+1}_{\mathcal{R}}$

In our setting, $\mathcal{A}^{i+1}_{\mathcal{R}}$ is built from $\mathcal{A}^i_{\mathcal{R}}$ by using a *completion step* that relies on finding critical pairs. Given a substitution $\sigma : \mathcal{X} \mapsto \mathcal{Q}$ and a rule $l \rightarrow r \Leftarrow c_1 \wedge \ldots \wedge c_n \in \mathcal{R}$, a critical pair is a pair $(r\sigma', q)$ where $q \in \mathcal{Q}$ and σ' is the greatest substitution w.r.t $\sqsubseteq$ such that $l\sigma \rightarrow^*_{\mathcal{A}^i_{\mathcal{R}}} q$, $\sigma \sqsupseteq \sigma'$ and $c_1\sigma' \wedge \ldots \wedge c_n\sigma'$. Observe that since $\mathcal{R}$, $\mathcal{A}^i_{\mathcal{R}}$, $\mathcal{Q}$ are finite, there is only a finite number of such critical pairs. For each critical pair such that $r\sigma' \not\rightarrow^*_{\mathcal{A}^i_{\mathcal{R}}} q$, the algorithm adds two new transitions $r\sigma' \rightarrow q'$ and $q' \rightarrow q$ to $\mathcal{A}^i_{\mathcal{R}}$, in order to enrich the language of the previous automaton.

Finding critical pairs for a rewriting rule $l \rightarrow r$ requires to detect all substitutions $\sigma : \mathcal{X} \mapsto \mathcal{Q}$ such that $l\sigma \rightarrow^* q$, where q is a state of the automaton. In what follows, we use the standard *matching algorithm* introduced in [22]. This algorithm $Matching(l, \mathcal{A}, q)$, which is described hereafter, matches a linear term l with a state q in the automaton $\mathcal{A}$. The solution returned by $Matching$ is a disjunction of possible substitutions $\sigma_1 \vee \ldots \vee \sigma_n$ so that $l\sigma_i \rightarrow^*_{\mathcal{A}} q$.

Let us recall the standard matching algorithm:

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

$$\text{(Config)}\ \frac{s \trianglelefteq q}{s \trianglelefteq u_1 \vee \cdots \vee s \trianglelefteq u_k \vee \bot}, \forall u_i,\ s.t.\ u_i \rightarrow q \in \Delta, \text{if } s \notin \mathcal{X}.$$

Moreover, after each application of one of these rules, the result is also rewritten into disjunctive normal form, using:

$$\frac{\phi_1 \wedge (\phi_2 \vee \phi_3)}{(\phi_1 \wedge \phi_2) \vee (\phi_1 \wedge \phi_3)} \quad \frac{\phi_1 \vee \bot}{\phi_1} \quad \frac{\phi_1 \wedge \bot}{\bot}$$

However, as our *TRS* relies on conditions, we have to extend this matching algorithm in order to guarantee that each substitution σ_i that is a solution of $l \rightarrow r \Leftarrow c_1 \wedge \ldots \wedge c_n$ satisfies $c_1 \wedge \ldots \wedge c_n$. For example, given the rule $f(x) \rightarrow f(g(x)) \Leftarrow x \geq 3 \wedge x \leq 7$ and the transitions $[2,8] \rightarrow q_1$, $f(q_1) \rightarrow q_2$, we have that the set of substitution returned by the matching algorithm is $\{x \mapsto [2,8]\}$, which is restricted to $[3,7]$.

Restricting substitutions is done by a solver on abstract domains. Such solver takes as input the lambda transitions of the automaton and all conditions of the rules, and outputs a set of substitutions of the form $\sigma' = \{x \mapsto \lambda_x, y \mapsto \lambda_y\}$. Such solvers exist for various abstract domains (see [20] for illustrations). In the present context, our solver has to satisfy the following property:

Property 1 (Correction of the solver). Let $\sigma = \{x_1 \mapsto q_1, \ldots, x_k \mapsto q_k\}$ be a substitution and $c = c_1 \wedge \cdots \wedge c_n$ a conjunction of constraints. We consider $\sigma/c = \{x_i \mapsto q_i \,|\, \exists 1 \leq j \leq n, x_i \in \mathcal{V}ar(c_j)\}$ the restriction of the substitution to the constrained variables. We also define $S_c = \{i \,|\, \exists 1 \leq j \leq n, x_i \in \mathcal{V}ar(c_j)\}$.

For any tuple $\langle \lambda_i | i \in S_c \rangle$ such that $\lambda_i \rightarrow^*_{\mathcal{A}} q_i$, $Solve_\Lambda(\sigma/c, \langle \lambda_i | i \in S_c \rangle, c)$ is a substitution σ' such that (1) if $i \notin S_c$, $\sigma'(x_i) = q_i$, and (2) if $i \in S_c$, $\sigma'(x_i) = \lambda'_i$. In addition, if a tuple of abstract values $\langle \lambda''_i | i \in S_c \rangle$, satisfies (a) $\forall i \in S_c$, $\lambda''_i \sqsubseteq \lambda_i$, and (b) $\forall 1 \leq j \leq n$, the substitution $\sigma''/c = \{x_i \mapsto \lambda''_i\}$ satisfies c_j, then $\forall i \in S_c$, $\lambda''_i \sqsubseteq \lambda'_i$.

Using Prop.1, the global function $Solve(\sigma, \mathcal{A}, c_1 \wedge \cdots \wedge c_n)$ is defined as:

$$Solve(\sigma, \mathcal{A}, c_1 \wedge \cdots \wedge c_n) = \bigcup_{\lambda_1 \rightarrow^*_{\mathcal{A}} q_1, \ldots, \lambda_k \rightarrow^*_{\mathcal{A}} q_k} Solve_\Lambda(\sigma/c, \langle \lambda_i | i \in S_c \rangle, c)$$

The following theorem ensures that $Solve(\sigma, \mathcal{A}, c_1 \wedge \cdots \wedge c_n)$ is an over-approximation of the solution of the constraints.

Theorem 1. *$Solve(\sigma, \mathcal{A}, c_1 \wedge \cdots \wedge c_n)$ is an over-approximation of the solutions of the constraints.*

Proof. By Prop.1, we have that for any tuple $\langle \lambda_i | i \in S_c \rangle$ such that $\lambda_i \rightarrow^*_{\mathcal{A}} q_i$, then $Solve_\Lambda(\sigma/c, \langle \lambda_i | i \in S_c \rangle, c)$ is a substitution σ' such that if $i \in S_c$, $\sigma'(x_i) = \lambda'_i$. Let $\langle \lambda''_i | i \in S_c \rangle$ be a tuple such that $\forall 1 \leq j \leq n$, we have that the sustitution $\sigma''/c = \{x_i \mapsto \lambda''_i\}$ satisfies c_j. Thanks to Prop.1, we have that $\forall i \in S_c$, $\lambda''_i \sqsubseteq \lambda'_i$. Since for all $i \in S_c$, λ'_i is returned by the solver, we can deduce that the set of substitutions returned by the solver is an over-approximations of the solutions of the constraints.

Depending of the abstract domain Λ and on the type of constraints of c, defining a solver that satisfies the above property may be complex. However, we shall now see that an easy solution can already be obtained if c is a conjunction of linear constraints and Λ the lattice of intervals. The algorithm computing $Solve_\Lambda(\sigma, \langle \lambda_1, \ldots, \lambda_k \rangle, c_1 \wedge \cdots \wedge c_n)$ is:

1. P_1 is the convex polyhedron defined by the constraints $c_1 \wedge \cdots \wedge c_n$,
2. P_2 is the box defined by the constraints $x_1 \in \lambda_1, \ldots x_k \in \lambda_k$,

3. if $P_1 \sqcap P_2 \neq \bot$, then we project $P_1 \sqcap P_2$ on each dimension (*i.e.* on each variable x_k) to obtain k new intervals. Otherwise, $Solve_\Lambda(\sigma, \langle\lambda_1, \ldots, \lambda_k\rangle, c_1 \wedge \cdots \wedge c_n) = \emptyset$.

Note that this algorithm also works for other abstract lattices such as octogons or convex polyhedra, but the approximation may be rough. We can however define finer solver thanks to linear programming.

Definition 9 (Matching solutions of conditional rewrite rules). *Let $\mathcal{A}$ be a tree automaton, $rl = l \to r \Leftarrow c_1 \wedge \ldots \wedge c_n$ a rewrite rule and q a state of $\mathcal{A}$. The set of all possible substitutions for the rewrite rule rl is $\Omega(\mathcal{A}, rl, q) = \{\sigma' \mid \sigma \in Matching(l, \mathcal{A}, q) \wedge \sigma' \in Solve(\sigma, \mathcal{A}, c_1 \wedge \ldots \wedge c_n) \wedge \nexists \sigma'' : r\sigma' \sqsubseteq r\sigma'' \to_{\mathcal{A}}^* q\}$.*

Once the set of all possible restricted substitutions σ_i has been obtained, we have to add the rules $r\sigma_i \to^* q$ in the automaton. However, the transition $r\sigma_i \to q$ is not necessarily a normalized ground transition of the form $f(q_1, \ldots, q_n) \to q$ or a lambda transition of the form $\lambda \to q$, which means that it has to be normalized first in order to be added to the LTA. For instance a transition $f(g([1,3]), 4) \to q$ is not normalized: 4 has to be abstracted and $g([1,3])$ has to be replaced by a state recognizing this term. This is the purpose of the following normalization algorithm.

Definition 10 (Normalization). *Let $s \in \mathcal{T}(\mathcal{F} \cup \mathcal{Q})$, $q \in \mathcal{Q}$, $\mathcal{A} = \langle \mathcal{F}, \mathcal{Q}, \mathcal{Q}_f, \Delta \rangle$ an LTA, where $\mathcal{F}_\bullet$ is the set of concrete interpretable symbols used in the TRS, $\mathcal{F}_\bullet^\#$ the set of abstract interpretable symbols used in $\mathcal{A}$, $\mathcal{F} = \mathcal{F}_\bullet^\# \cup \mathcal{F}_\circ$, and $\alpha : \mathcal{F}_\bullet^0 \to \mathcal{F}_\bullet^{\#^0}$ the abstraction function, transforming a concrete symbole to an element of the lattice Λ. A* new state *is a state of $\mathcal{Q}$ not occurring in Δ. $Norm(s \to^* q)$ returns the set of normalized transitions deduced from s. $Norm(s \to^* q)$ is inductively defined by:*

1. *if $s \in \mathcal{F}_\bullet^0$ (i.e., in the concrete domain used in rewrite rules), $Norm(s \to^* q) = \{\alpha(s) \to q\}$.*
2. *if $s \in \mathcal{F}_\circ^0 \cup \mathcal{F}_\bullet^{\#^0}$ then $Norm(s \to^* q) = \{s \to q\}$,*
3. *if $s = f(t_1, \ldots, t_n)$ where $f \in \mathcal{F}_\circ^n \cup \mathcal{F}_\bullet^n$, then $Norm(s \to^* q) = \{f(q_1', \ldots, q_n') \to q\} \cup Norm(t_1 \to q_1') \cup \ldots \cup Norm(t_n \to q_n')$ where for $i = 1 \ldots n$, q_i' is either:*
   - *the right-hand side of a transition of Δ such that $t_i \to_\Delta^* q_i'$*
   - *or a new state, otherwise.*

Observe that the normalization algorithm always terminates.

Example 9. Let q_1, q_2, q_3 be new states, $Norm(f(g([1,3]), 4) \to q) = \{[1,3] \to q_1, [4,4] \to q_2, g(q_1) \to q_3, f(q_3, q_2) \to q\}$

We conclude by the formal characterization of the one step completion.

Definition 11 (One step completed automaton $\mathcal{C}_\mathcal{R}(\mathcal{A})$). *Let $\mathcal{A} = \langle \mathcal{F}, \mathcal{Q}, \mathcal{Q}_f, \Delta \rangle$ be a tree automaton, $\mathcal{R}$ be a left-linear TRS. We denote by $\mathcal{C}_\mathcal{R}(\mathcal{A})$ the one step completed automaton $\mathcal{C}_\mathcal{R}(\mathcal{A}) = \langle \mathcal{F}, \mathcal{Q}', \mathcal{Q}_f, \Delta' \rangle$ where:*

$$\Delta' = \Delta \cup \bigcup_{l \to r \in \mathcal{R},\, q \in \mathcal{Q},\, \sigma \in \Omega(\mathcal{A}, l \to r, q)} Norm(r\sigma \to^* q') \cup \{q' \to q\}$$

where $\Omega(\mathcal{A}, l \to r, q)$ is the set of all possible substitutions defined in Def.9, $q' \notin \mathcal{Q}$ a new state and $\mathcal{Q}'$ contains all the states of Δ'.

4.2 Equational Abstraction

As we already said, completion may not terminate. In order to enforce termination of the process, we suggest to merge states according to a set *approximation equations* E([31]). An approximation equation is of the form $u = v$, where $u, v \in \mathcal{T}(\mathcal{F}_\circ, \mathcal{X})$. Let $\sigma : \mathcal{X} \mapsto \mathcal{Q}$ be a substitution such that $u\sigma \to_{\mathcal{A}_\mathcal{R}^{i+1}} q$, $v\sigma \to_{\mathcal{A}_\mathcal{R}^{i+1}} q'$ and $q \neq q'$, then we know that there exists at least two terms s and t such that $s \to_{\mathcal{A}_\mathcal{R}^{i+1}} q$, $t \to_{\mathcal{A}_\mathcal{R}^{i+1}} q'$ and s is equal to t w.r.t. equational theory defined by E. An over-approximation of $\mathcal{A}_\mathcal{R}^{i+1}$, which we denote $\mathcal{A}_{\mathcal{R},E}^{i+1}$, can be obtained by merging states q and q'.

Definition 12 (*merge*). *Let $\mathcal{A} = \langle \mathcal{F}, \mathcal{Q}, \mathcal{Q}_F, \Delta \rangle$ be an LTA and q_1, q_2 be two states of $\mathcal{A}$. We denote by $merge(\mathcal{A}, q_1, q_2)$ the tree automaton where each occurrence of q_2 is replaced by q_1.*

Equations on interpretable terms. In what follows, we need to extend approximation equations to built-in terms. Indeed, as illustrated in the following example, approximation equations defined on $\mathcal{T}(\mathcal{F}_\circ, \mathcal{X})$ are not powerful enough to ensure termination.

Example 10. Let $f(x) \to f(x + 1)$ be a rewrite rule, $\{[1, 1] \to q_1, [2, 2] \to q_2, f(q_2) \to q_f\}$ be transitions of an LTA, then a first completion step will add $f(q_2 + 1) \to^* q_f$, which means it will add transitions $q_2 + q_1 \to q_3$ and $f(q_3) \to q_f$ after normalization. Since we have now $f(q_3) \to q_f$, a second completion step will add, in the same manner, $q_3 + q_1 \to q_4$ and $f(q_4) \to q_f$. Then the next completion step will add $q_4 + q_1 \to q_5$ and $f(q_5) \to q_f$. And we can deduce that the $i - th$ completion step will add transition $q_i + q_1 \to q_{i+1}$ and $f(q_{i+1}) \to q_f$, ... Unfortunately, as classical equations do not work on terms with interpretable symbols, this infinite behaviour cannot be captured.

We define a new type of equation which works on interpretable terms, that are applied with conditions. Such equations have the form $u = v \Leftarrow c_1 \wedge \ldots \wedge c_n$, where $u, v \in \mathcal{T}(\mathcal{F}_\circ \cup \mathcal{F}_\bullet, \mathcal{X})$. We observe that we can almost use the same matching algorithm than for completion. The first main difference is that we need to match a term $t \in \mathcal{T}(\mathcal{F}_\circ \cup \mathcal{F}_\bullet, \mathcal{X})$ built on interpreted symbols on terms of $\mathcal{T}(\mathcal{F}_\circ \cup \mathcal{F}_\bullet^\#, \mathcal{X})$ recognized by the LTA $\mathcal{A}$. Let $\alpha : \mathcal{F}_\bullet \mapsto \mathcal{F}_\bullet^\#$ the abstraction function, the solution is to use the same matching algorithm on $\alpha(t)$ and $\mathcal{A}$, *i.e* $Matching(\alpha(t), \mathcal{A}, q)$. Contrary to the completion case, we do not need to restrict the substitutions obtained by the matching algorithm with respect to the constraints of the equation, but simply guarantee that such constraints are satisfiable, i.e., $Solve(\sigma, \mathcal{A}, c_1 \wedge \cdots \wedge c_n) \neq \emptyset$.

Example 11. We consider the same rewrite rule as in Ex. 10 and we apply the completion and normalization steps. If we use equation $x = x + 1 \Leftarrow x > 3$ in Ex. 10, that informally means that we can merge q_4 and q_5 since $q_4 + q_1 \rightarrow q_5$ and $[1, 1] \rightarrow q_1 \in \Delta$, and since q_4 and q_5 can respectively recognize the intervals $[4, 4]$ and $[5, 5]$ by transitivity and thus satisfy the condition "$x > 3$". We need a new evaluation operator to determine exactly when two states can satisfy the condition of the equation. This new operator is defined in the next subsection.

Theorem 2. *Let $\mathcal{A}$ be an LTA and E a set of equations. We denote by $\leadsto^!_E$ the transformation of $\mathcal{A}$ by merging equivalent states according to E. The language of the resulting automaton $\mathcal{A}'$ such that $\mathcal{A} \leadsto^!_E \mathcal{A}'$ is an over-approximation of the language of $\mathcal{A}$, i.e., $\mathcal{L}(\mathcal{A}) \subseteq \mathcal{L}(\mathcal{A}')$.*

Proof. Let $\mathcal{A}$ and $\mathcal{A}'$ two automata and E be a set of equations such that $\mathcal{A} \leadsto^!_E \mathcal{A}'$. The set of transition of $\mathcal{A}'$ is the same as $\mathcal{A}$ with states merged according to equivalence classes determined by E. For all $t \in \mathcal{T}(\mathcal{F}, \mathcal{X})$, for all states q of $\mathcal{A}$, let $Q = \{q_1, \ldots, q, \ldots, q_n\}$ an equivalence class determined by E. We have that $t \in \mathcal{L}(\mathcal{A}, q) \Rightarrow t \rightarrow^*_{\mathcal{A}} q \Rightarrow t \rightarrow^*_{\mathcal{A}'} Q \Rightarrow t \in \mathcal{L}(\mathcal{A}', Q)$.

4.3 Evaluation and Correctness

In this section, we formally define completion on LTA and its correctness. We first start with the evaluation of an LTA.

Evaluation of a Lattice Tree Automaton. We observe that any set of concrete terms that contains the term $1 + 2$ should also contains the term 3. While, this canonical property can be naturally assumed when building the initial set of states, it may eventually be broken when performing a completion step or by merging states. Indeed, let $f(x) \rightarrow f(x + 1)$ be a rewrite rule and $\sigma : x \mapsto q_2$ a substitution, a completion step applied on $\{[1, 1] \rightarrow q_1, [2, 3] \rightarrow q_2, f(q_2) \rightarrow q_f\}$ will add the rule $f(q_3) \rightarrow q_4$, $q_2 + q_1 \rightarrow q_3$, and $q_3 \rightarrow q_f$. Since the language recognized by q_3 contains the term $q_2 + q_1$, it should also contain the term $[3, 4]$. Evaluation of this set of transitions will add the transition $[3, 4] \rightarrow q_3$. This is done by applying the *propag* function.

Definition 13 (*propag*)**.**

$$propag(\Delta) = \begin{cases} \Delta \text{ } if \, \exists \lambda \rightarrow q \in \Delta \wedge eval(f(\lambda_1, \ldots, \lambda_k)) \sqsubseteq \lambda \\ \Delta \cup \{eval(f(\lambda_1, \ldots, \lambda_k)) \rightarrow q\}, otherwise. \end{cases}$$

$\forall f \in \mathcal{F}_\bullet^{\#^k} : \forall q, q_1, \ldots, q_k \in \mathcal{Q} : \forall \lambda_1, \ldots, \lambda_k \in \Lambda : f(q_1, \ldots, q_k) \rightarrow q \in \Delta \wedge \{\lambda_1 \rightarrow^*_\Delta q_1, \ldots, \lambda_k \rightarrow^*_\Delta q_k\} \subseteq \Delta$.

Example 12. Let $\Delta = \{[3, 6] \rightarrow q_1, [2, 8] \rightarrow q_2, q_1 + q_2 \rightarrow q_3, f(q_3) \rightarrow q_f\}$, then *propag* will evaluate the term $[3, 6]+[2, 8]$ contained in the transition $q_1+q_2 \rightarrow q_3$, and add the transition $[5, 14] \rightarrow q_3$ to the automaton.

Using *propag*, we can extend the *eval* function to sets of transitions and to tree automata in the following way.

Definition 14 (*eval* on transitions and automata).
Let μX the least fix-point obtained by iterating propag.

- $eval(\Delta) = \mu X.propag(X) \cup \Delta$ *and*
- $eval(\langle \mathcal{F}, \mathcal{Q}, \mathcal{Q}_f, \Delta \rangle) = \langle \mathcal{F}, \mathcal{Q}, \mathcal{Q}_f, eval(\Delta) \rangle$

Theorem 3. $\mathcal{L}(\mathcal{A}) \subseteq \mathcal{L}(eval(\mathcal{A}))$

Proof. By definition of *propag* (Def.13), we have that $propag(\Delta) = Delta$ if $\exists \lambda \to q \in \Delta \wedge eval(\lambda_1 \bullet \ldots \bullet \lambda_k) \sqsubseteq \lambda$ or $propag(\Delta) = \Delta \cup \{eval(\lambda_1 \bullet \ldots \bullet \lambda_k) \to q$. In each case, $\Delta \subseteq propag(\Delta)$.

By definition of *eval* (Def.14), $eval(\Delta) = \mu X.propag(X) \cup \Delta$. Since $\Delta \subseteq propag(\Delta)$, we have that $\Delta \subseteq eval(\Delta)$. Then we can deduce that $\mathcal{L}(\mathcal{A}) \subseteq \mathcal{L}(eval(\mathcal{A}))$.

Observe that the fixpoint computation may not terminate. Indeed, consider $\Delta = \{[3,6] \to q_1, [2,8] \to q_2, q_1 + q_2 \to q_2\}$. The first iteration of the fixpoint will evaluate the term $[3,6] + [2,8]$ recognized by $q_1 + q_2 \to q_2$, which adds the transition $[5,14] \to q_2$. Since a new element is in the state q_2, the second iteration will evaluate the term $[3,6] + [5,14]$ recognized by the transition $q_1 + q_2 \to q_2$, and will add the transition $[8,20] \to q_2$. The third iteration will evaluate the term $[3,6] + [8,20]$ to q_2 and this pattern will be repeated in further operations. Since there will always be a new element of the lattice that will be associated to q_2, the computation of the evaluation will not terminate. It is thus necessary to apply a widening operator $\nabla_\Lambda : \Lambda \times \Lambda \mapsto \Lambda$ to force the computation of *propag* to terminate. For example, if we apply such a widening operator on the example above, after 3 iterations of the *propag* function, the transitions: $[2,8] \to q_2$, $[5,14] \to q_2$, $[8,20] \to q_2$ could be replaced by $[2, +\infty[\to q_2$.

Definition 15 (Automaton completion for LTA). *Let $\mathcal{A}$ be a tree automaton, $\mathcal{R}$ a TRS and E a set of equations. At a step i of completion, we denote by $\mathcal{A}^i_{\mathcal{R},E}$ the LTA such that $\mathcal{A}^i_{\mathcal{R}} \leadsto^!_E \mathcal{A}^i_{\mathcal{R},E}$.*

- $\mathcal{A}^0_{\mathcal{R},E} = \mathcal{A}$,
- *Repeat* $\mathcal{A}^{n+1}_{\mathcal{R},E} = \mathcal{A}'$ *with* $\mathcal{C}_{\mathcal{R}}(eval(\mathcal{A}^n_{\mathcal{R},E})) \leadsto^!_E \mathcal{A}''$ *and* $eval(\mathcal{A}'') = A'$,
- *Until a fixpoint* $\mathcal{A}^*_{\mathcal{R},E} = \mathcal{A}^k_{\mathcal{R},E} = \mathcal{A}^{k+1}_{\mathcal{R},E}$ *(with $k \in \mathbb{N}$) is joint.*

A running example is described in section 5.

Theorem 4 (Completeness). *Let $\mathcal{R}$ be a left-linear TRS, $\mathcal{A}$ be a tree automaton and E be a set of linear equations. If completion terminates on $\mathcal{A}^*_{\mathcal{R},E}$ then*

$$\mathcal{L}(\mathcal{A}^*_{\mathcal{R},E}) \supseteq \mathcal{R}^*(\mathcal{L}(\mathcal{A}))$$

Proof. We first show that $\mathcal{L}(\mathcal{A}^*_{\mathcal{R},E}) \supseteq \mathcal{L}(\mathcal{A})$. By definition, completion only adds transitions to $\mathcal{A}$. Hence, we trivially have $\mathcal{L}(\mathcal{A}^1_{\mathcal{R}}) \supseteq \mathcal{L}(\mathcal{A})$. Thanks to Theorem 2, we also know that $\mathcal{A}^1_{\mathcal{R},E}$, the transformation of $\mathcal{A}^1_{\mathcal{R}}$ by merging states equivalent w.r.t. E, is such that $\mathcal{L}(\mathcal{A}^1_{\mathcal{R},E}) \supseteq \mathcal{L}(\mathcal{A}^1_{\mathcal{R}})$. Hence, by transitivity of $\supseteq$, we know that $\mathcal{L}(\mathcal{A}^1_{\mathcal{R},E}) \supseteq \mathcal{L}(\mathcal{A})$. This can be successively applied to $\mathcal{A}^2_{\mathcal{R},E}, \mathcal{A}^3_{\mathcal{R},E}, \mathcal{A}^4_{\mathcal{R},E}, \ldots$ so that $\mathcal{L}(\mathcal{A}^*_{\mathcal{R},E}) \supseteq \mathcal{L}(\mathcal{A})$. Now, the next step of the proof consists in showing that for all term $s \in \mathcal{L}(\mathcal{A})$ if $s \to^*_{\mathcal{R}} t$ then $t \in \mathcal{L}(\mathcal{A}^*_{\mathcal{R},E})$. First, note that by definition of application of E final states are preserved, i.e. if q is a final state in $\mathcal{A}$ then if $\mathcal{A}'$ is the automaton where E are applied in $\mathcal{A}$ and q has been renamed in q', then q' is a final state of $\mathcal{A}'$. Hence we only have to prove that for all term $s \in \mathcal{L}(\mathcal{A}, q)$ if $s \to^*_{\mathcal{R}} t$ then $\exists q' : t \in \mathcal{L}(\mathcal{A}^*_{\mathcal{R},E}, q')$. We proceed by induction on the length of $\to^*_{\mathcal{R}}$:

- if length is zero then $s \to^*_{\mathcal{R}} s$ and we trivially have that $s \in \mathcal{L}(\mathcal{A}^*_{\mathcal{R},E}, q')$.
- assume now that the property is true for any rewriting derivation of length less or equal to n, we prove that the property remains valid for a derivation of length less or equal to $n+1$. Assume that we have $s \to^n_{\mathcal{R}} s' \to_{\mathcal{R}} t$. Using induction hypothesis, we obtain that $s' \in \mathcal{L}(\mathcal{A}^*_{\mathcal{R},E}, q')$. It remains to prove that $t \in \mathcal{L}(\mathcal{A}^*_{\mathcal{R},E}, q')$ can be deduced from $s' \to_{\mathcal{R}} t$. Since $s' \to_{\mathcal{R}} t$, we know that there exist a rewrite rule $l \to r \Leftarrow c_1 \wedge \ldots \wedge c_m$, a position p and a substitution $\mu : \mathcal{X} \mapsto \mathcal{T}(\mathcal{F})$ such that $s' = s'[l\mu]_p \to_{\mathcal{R}} eval(s'[r\mu]_p) = t$ and for all $j \in [1, m]$, $c_j\mu = true$. Since $s' \in \mathcal{L}(\mathcal{A}^*_{\mathcal{R},E}, q')$, $s'[l\mu]_p \to^*_{\mathcal{A}^*_{\mathcal{R},E}} q'$ and by definition of the langage of an LTA, we get that there exists s'' such that $s' \sqsubseteq s''$ and $s'' \to^*_{\mathcal{A}^*_{\mathcal{R},E}} q'$. We can deduce that $s''[l\mu]_p \to^*_{\mathcal{A}^*_{\mathcal{R},E}} q'$ and by definition of tree automata derivation, that there exists a state q'' such that $l\mu \to^*_{\mathcal{A}^*_{\mathcal{R},E}} q''$ and $s''[q'']_p \to^*_{\mathcal{A}^*_{\mathcal{R},E}} q'$. Let $Var(l) = \{x_1, \ldots, x_k\}$, $l = l[x_1, \ldots, x_k]$ and $t_1, \ldots, t_k \in \mathcal{T}(\mathcal{F})$ such that $\mu = \{x_1 \mapsto t_1, \ldots, x_k \mapsto t_k\}$. Since $l\mu = l[t_1, \ldots, t_k] \to^*_{\mathcal{A}^*_{\mathcal{R},E}} q''$, we know that there exist states $q_1, \ldots, q_k$ such that $\forall i \in [1, k]$, $t_i \to^*_{\mathcal{A}^*_{\mathcal{R},E}} q_i$ and $l[q_1, \ldots, q_k] \to^*_{\mathcal{A}^*_{\mathcal{R},E}} q''$. Let $\sigma = \{x_1 \mapsto q_1, \ldots, x_k \mapsto q_k\}$, we thus have that $l\sigma \to^*_{\mathcal{A}^*_{\mathcal{R},E}} q''$ thanks to left-linearity.
  We have that $\mu = \{x_1 \mapsto t_1, \ldots, x_k \mapsto t_k\}$ and $\forall i \in [1, k]$, $t_i \to^*_{\mathcal{A}^*_{\mathcal{R},E}} q_i$. Since for all $j \in [1, m]$, $c_j\mu = true$, by definition of predicates, all t_i are interpretable terms (or $c_j\mu$ would be equal to false). So for all $i \in [1, k]$ there exist $\lambda_i \in \Lambda$ such that $eval(t_i) = \lambda_i$. Let μ' be the substitution $\{x_1 \mapsto \lambda_1, \ldots, x_k \mapsto \lambda_k\}$, then we can deduce that for all $j \in [1, m]$, $c_j\mu' = true$. Thanks to evaluation step, we can deduce that for all $i \in [1, k]$, if $t_i \to^*_{\mathcal{A}^*_{\mathcal{R},E}} q_i$, then $eval(t_i) = \lambda_i \to^*_{\mathcal{A}^*_{\mathcal{R},E}} q_i$. The property on the solver states that $Solve(\sigma, \mathcal{A}, c_1 \wedge \cdots \wedge c_m) = \bigcup_{\lambda_1 \to^*_{\mathcal{A}} q_1, \ldots, \lambda_k \to^*_{\mathcal{A}} q_k} Solve_{\Lambda}(\sigma/c, \langle \lambda_i | i \in S_c \rangle, c)$. So we can deduce that for all $j \in [1, m]$ $c_i\sigma = true$ because $c_i\mu' = true$. Since $\mathcal{A}^*_{\mathcal{R},E}$ is a fixpoint of completion, from $l\sigma \to^*_{\mathcal{A}^*_{\mathcal{R},E}} q''$ and the fact that for all $j \in [1, m]$, $c_j\sigma = true$, we can deduce that $r\sigma \to^*_{\mathcal{A}^*_{\mathcal{R},E}} q''$. Furthermore, since $\forall i \in [1, k]$, $t_i \to^*_{\mathcal{A}^*_{\mathcal{R},E}} q_i$, then $r\mu \to^*_{\mathcal{A}^*_{\mathcal{R},E}} q''$. Since besides of this $s''[q'']_p \to^*_{\mathcal{A}^*_{\mathcal{R},E}} q'$, we have that $s''[r\mu]_p \to^*_{\mathcal{A}^*_{\mathcal{R},E}} q'$. Since $s' \sqsubseteq s''$, this means by definition that $eval(s') \sqsubseteq eval(s'')$. Finally, since $s''[r\mu]_p \to^*_{\mathcal{A}^*_{\mathcal{R},E}} q'$ and $eval(s') \sqsubseteq eval(s'')$, we can deduce that $t = eval(s'[r\mu]_p) \to^*_{\mathcal{A}^*_{\mathcal{R},E}} q'$, hence $t \in \mathcal{L}(\mathcal{A}^*_{\mathcal{R},E}, q')$.

Observe that the reverse does not hold as widening in evaluation may introduce over-approximations.

Remark 1. We have two infinite dimensions, the first one due to the state space, and the second one due to infinite domain. The infinite behaviour of the system (infinite state space) is abstracted thanks to the equations, and all the infinite behaviours due to the operations on elements of the lattice (e.g. $x \to x + 1$) are captured by the widening step included in the evaluation step. Indeed, if we have lambda transitions added at each completion step with increasing (or decreasing) elements of the lattice (for example $[0,2] \to q$, $[2,4] \to q$, $[4,6] \to q, \ldots$), we have to perform a widening (here $[0,+\infty[$) to ensure the terminaison of the computation. But an infinite increasing (or decreasing) sequence of lambda transitions is necessarily obtained from a predefined operation of the lattice used in the rewrite rules. For example, the increasing sequence described above is necessarily obtained from a rewrite rule of the form $u(\ldots,\mathbf{x},\ldots) \to v(\ldots,\mathbf{x} + \mathbf{2},\ldots)$. If we have the matching $x \mapsto q_1$, and the transition $[2,2] \to q_2$, then it will add the transition $q_1 + q_2 \to q_3$, and since this rewrite rule leads to an infinite behaviour (always adding 2), we would have an infinite sequence $q_3 + q_2 \to q_4$, $q_4 + q_2 \to q_5$, and so on. To solve this problem, it is necessary to use an equation of the form $x = x + 2$. Then, q_1 is merged to q_3 and we have a transition $q_1 + q_2 \to q_1$ with an infinite evaluation abstracted thanks to the widening step included in the evaluation step. To summarize, an infinite sequence of lambda transitions is necessarily obtained from an operation used in the rewriting system, and since the transitions of an LTA containing operations have to be evaluated, the infinite behavior is always solved during the evaluation step. We can observe this on the example described hereafter in 5.

5 A running example

Let $\mathbb{Z}$ be the concrete domain, the set of intervals on $\mathbb{Z}$ be the lattice, $\mathcal{R} = \{f(x) \to cons(x, f(x+1)) \Leftarrow x < 3_{(A)}, f(x) \to cons(x, f(x+2)) \Leftarrow x > 2_{(B)}\}$ be the TRS, $\mathcal{A}_0$ the LTA representing the set of initial configurations, with the following set of transitions : $\Delta_0 = \{[1,2] \to q_1, f(q_1) \to q_2\}$, and $E = \{x = x + 2 \Leftarrow x > 5\}$ the set of equations. We decide to use the widening operator after three steps of evaluation.

First step of completion
One step completed automaton: we can apply the rewrite rule (A) with the substitution $x \mapsto q_1$, and so add $Norm(cons(q_1, f(q_1 + 1)) \to q_2')$ and $q_2' \to q_2$ to Δ_1.

So we have $\Delta_2 = \Delta_1 \cup \{cons(q_1, q_3) \to q_2', q_2' \to q_2, f(q_4) \to q_3, q_1 + q_{[1,1]} \to q_4, [1,1] \to q_{[1,1]}\}$, where q_3 and q_4 are new states induced by normalisation.
Since there is new transitions, we have to perform the evaluation step : transition $q_1 + q_{[1,1]} \to q_4$ can be evaluated, so $eval(\Delta_2) = \Delta_2 \cup \{[2,3] \to q_4\}$.
Abstraction by merging states according to equations: we cannot apply the set of equations yet because there is no state recognizing "$x + 2$" such that $x > 5$.

Second step of completion
One step completed automaton: we can apply the rewrite rules (A) and (B) with the substitution $x \mapsto q_4$, but this will be restricted by the solver. In fact, (A) will be applied on $[2, 2]$ (condition $x < 3$), and (B) will be applied on $[3, 3]$. So $Norm(cons([2, 2], f([2, 2]+1)) \to q'_3)$ and $q'_3 \to q_3$, $Norm(cons([3, 3], f([3, 3] + 2)) \to q''_3)$ and $q''_3 \to q_3$ will be add to $eval(\Delta_2)$.

So we have $\Delta_3 = eval(\Delta_2) \cup \{[2, 2] \to q_{[2,2]}, cons(q_{[2,2]}, q_5) \to q'_3, q'_3 \to q_3, f(q_6) \to q_5, q_{[2,2]} + q_{[1,1]} \to q_6, [3, 3] \to q_{[3,3]}, cons(q_{[3,3]}, q_7) \to q''_3, q''_3 \to q_3, f(q_8) \to q_7, q_{[3,3]} + q_{[2,2]} \to q_8\}$.
Evaluation step: $eval(\Delta_2) = \Delta_2 \cup \{[3, 3] \to q_6, [5, 5] \to q_8\}$.
Abstraction step: we cannot apply the set of equations yet.

Third step of completion
One step completed automaton: we can apply rule (B) with substitution $x \mapsto q_6 : Norm(cons(q_6, f(q_6+ 2)) \to q'_5)$ has to be add, but $cons([3, 3], f([3, 3] + 2))$ already belongs to the language of the current automaton, so it does not add any new transitions. We can also apply the rewrite rule (B) with the substitution $x \mapsto q_8$. So $Norm(cons(q_8, f(q_8 + 2)) \to q'_7)$, and $q'_7 \to q_7$ will be add to $eval(\Delta_3)$.

So we have $\Delta_3 = eval(\Delta_3) \cup \{cons(q_8, q_9) \to q'_7, q'_7 \to q_7, f(q_{10}) \to q_9, q_8 + q_{[2,2]} \to q_{10}\}$.
Evaluation step: $eval(\Delta_3) = \Delta_3 \cup \{[7, 7] \to q_{10}\}$.
Abstraction step: As long as $q_8 + q_{[2,2]} \to q_{10}$, $[5, 5] \to q_8$ and $\gamma([5, 5]) > 4$, q_8 and q_{10} are merged according to the set of equations E.

Fourth step of completion
Let us see the full automaton at this step. We have $Merge(eval(\Delta_3), q_8, q_{10})) = \{[1, 2] \to q_1, f(q_1) \to q_2, cons(q_1, q_3) \to q'_2, q'_2 \to q_2, f(q_4) \to q_3, q_1 + q_{[1,1]} \to q_4, q_{[1,1]} \to [1, 1], [2, 3] \to q_4, [2, 2] \to q_{[2,2]}, cons(q_{[2,2]}, q_5) \to q'_3, q'_3 \to q_3, f(q_6) \to q_5, q_{[2,2]} + q_{[1,1]} \to q_6, [3, 3] \to q_{[3,3]}, cons(q_{[3,3]}, q_7) \to q''_3, q''_3 \to q_3, f(q_8) \to q_7, q_{[3,3]} + q_{[2,2]} \to q_8, [5, 5] \to q_8, cons(q_8, q_9) \to q'_7, q'_7 \to q_7, f(q_8) \to q_9, q_8+q_{[2,2]} \to q_8, [7, 7] \to q_8\}$. Since the transitions have been modified thanks to the equations, we have to perform an evaluation step. We can nottice that evaluation of the transition $q_8 + q_{[2,2]} \to q_8$ is infinite. In fact, it will add $[7, 7] \to q_8$, $[9, 9] \to q_8$, $[11, 11] \to q_8$, $\ldots$, and so on. So we have to perform widening, that is to say, replace all the transitions $\lambda \to q_8$ by $[5, +\infty[\to q_8$.
One step completed automaton: Thanks to the widening performed at the previous evaluation step, no more rule has to be add in the current automaton. We have a fixed-point which is an over-approximation of the set of reachable states, and the completion stops.

6 On Improving the Verification of *Java* Programs by TRMC

We now show how our formalism can simplify the analysis of *Java* programs. In [11], the authors developed a tool called Copster [8], to compile a *Java* `.class` file into a Term

Rewriting System (TRS). The obtained TRS models exactly a subset of the semantics[1] of the *Java* Virtual Machine (JVM) by rewriting a term representing the state of the JVM [11]. States are of the form `IO(st,in,out)` where `st` is a program state, `in` is an input stream and `out` and output stream. A program state is a term of the form `state(f,fs,h,k)` where `f` is the current frame, `fs` is the stack of calling frames, `h` a heap and `k` a static heap. A frame is a term of the form `frame(m,pc,s,l)` where `m` is a fully qualified method name, `pc` a program counter, `s` an operand stack and `t` an array of local variables. The frame stack is the call stack of the frame currently being executed: `f`. For a given progam point `pc` in a given method `m`, Copster builds an `xframe` term very similar to the original `frame` term but with the current instruction explicitly stated, in order to compute intermediate steps.

One of the major difficulties of this encoding is to capture and handle the two-side infinite dimension that can arise in *Java* programs. Indeed, in such models, infinite behaviors may be due to unbounded calls to method and object creation, or simply because the program is manipulating unbounded data such as integer variables. While multiple infinite behaviors can be over-approximated with completion (just like a^nb^n can be approximated by a^*b^*), this may require to manipulate structures of large size. As an example, in [11], it was decided to encode the structure of configurations in an efficient manner, integer variables being encoded in Peano arithmetic. Not only that this choice has an impact on the size of the automata used to encode sets of configurations, but also each classical arithmetic operation may require the application of several rules.

As an example, let us consider the simple arithmetic operation "$300 + 400$". By using [11], this operation is represented by $xadd(succ^{300}(zero), succ^{400}(zero))$, which reduces to 5 rewriting rules detailled hereafter that have to be applied 300 times:
$xadd(zero, zero) \to result(zero)$
$xadd(succ(var(a)), pred(var(b))) \to xadd(var(a), var(b))$
$xadd(pred(var(a)), succ(var(b))) \to xadd(var(a), var(b))$
$xadd(succ(var(a)), succ(var(b))) \to xadd(succ(succ(var(a))), var(b))$
$xadd(pred(var(a)), pred(var(b))) \to xadd(pred(pred(var(a))), var(b))$
$xadd(succ(var(a)), zero) \to result(succ(var(a)))$
$xadd(pred(var(a)), zero) \to result(pred(var(a)))$
$xadd(zero, succ(var(b))) \to result(succ(var(b)))$
$xadd(zero, pred(var(b))) \to result(pred(var(b)))$

This means that if at the program point `pc` of method `m` there is a bytecode `add` then we switch to a `xframe` in order to compute the addition, i.e. apply $frame(m, pc, s, l) \to xframe(add, m, pc, s, l)$. To compute the result of the addition of the two first elements of the stack, we have to apply the rule $xframe(add, m, pc, stack(b(stack(a, s))), l) \to xframe(xadd(a, b), m, pc, s, l)$. Once the result is computed thanks to all the rewrite rules of $xadd$, we can compute the next operation of `m`, i.e. go to the next program point by applying $xframe(result(x), m, pc, s, l) \to frame(m, next(pc), stack(x, s), l)$.

[1] essentially basic types, arithmetic, object creation, field manipulation, virtual method invocation, as well as a subset of the String library.

The use of LTA can drastically simplify the above operations. Indeed, in our framework, we can encode natural numbers and operations directly in the alpabet of the automaton. In such context, the series of application of the rewritting rules is replaced by a one step evaluation. As an example, the rewrite rule $xframe(add, m, pc, stack(b(stack(a, s))), l) \rightarrow$ $xframe(xadd(a, b), m, pc, s, l)$ and rules $xadd$ encoding addition can be replaced by $xframe(add, m, pc, stack(b(stac$ $xframe(result(a + b), m, pc, s, l)$. The evaluation step of LTA completion will compute the result of addition of $a + b$ and add the resulting term to the language of the automaton.

Other operations such as "if-then-else" can also be drastically simplified by using our formalism. Indeed, with Peano numbers the evaluation of the condition of the instruction `"if"` requires several rules. As an example, the instruction `"if a=b then go to the program point x"` is encoded by the term $ifEqint(x, a, b)$, and the following rules will be applied:
$ifEqint(x, zero, zero) \rightarrow ifXx(valtrue, x)$
$ifEqint(x, succ(a), pred(b)) \rightarrow ifXx(valfalse, x)$
$ifEqint(x, pred(a), succ(b)) \rightarrow ifXx(valfalse, x)$
$ifEqint(x, succ(a), succ(b)) \rightarrow ifEqint(x, a, b)$
$ifEqint(x, pred(a), pred(b)) \rightarrow ifEqint(x, a, b)$
$ifEqint(x, succ(a), zero) \rightarrow ifXx(valfalse, x)$
$ifEqint(x, pred(a), zero) \rightarrow ifXx(valfalse, x)$
$ifEqint(x, zero, succ(b)) \rightarrow ifXx(valfalse, x)$
$ifEqint(x, zero, pred(b)) \rightarrow ifXx(valfalse, x)$

Rules of this type will disappear with LTA because an equality between two elements is directly evaluated, and so are all the predefined predicates.

In Copster, if at the program point `pc` of the method `m` we have an `"if"` where the condition is an equality between two elements, we switch to a `xframe` where the operation to evaluate is an `"if"` with a equality condition between the two first elements of the stack, and which go to a program point x if the condition is true. Then we can apply the rule $xframe(ifACmpEq(x), m, pc, stack(b, stack(a, s)), l) \rightarrow xframe(ifEqint(x, a, b), m, pc, s, l)$ which permits to compute the solution, i.e. calls the $ifEqint$ rules detailed above.

According to the result returned by these rules, we will go at program point x if the condition is true or else to the next program point. This is modelised by the two following rules:
$xframe(ifXx(valtrue, x), m, pc, s, l) \rightarrow frame(m, x, s, l)$
$xframe(ifXx(valfalse, x), m, pc, s, l) \rightarrow frame(m, next(pc), s, l)$

In LTA completion, thanks to the fact that predicates are directly evaluated and that we have conditional rules, all this rules are replaced by the two following conditional rules:
$xframe(ifACmpEq(x), m, pc, stack(b, stack(a, s)), l) \rightarrow frame(m, x, s, l) \Leftarrow a = b$ (if $a = b$ we go to program point p)
$xframe(ifACmpEq(x), m, pc, stack(b, stack(a, s)), l) \rightarrow frame(m, x, s, l) \Leftarrow a \neq b$ (if $a \neq b$ we go to next program point)

7 Experiments

We developped a version of Copster [8] that is abble to compile *Java* `.class` files either to TRS or to Conditional TRS with built-ins as defined in Section 4. A precise description of this translation can be found in [11]. We also developped TimbukLTA a version of Timbuk handling LTA completion of Section 4.1. Thus, we can prove the same properties on Java programs using either Timbuk or TimbukLTA and compare the efficiency LTA completion w.r.t. standard completion.

7.1 An introductory example

To illustrate the use of TimbukLTA, we start by a simple example that is not related to Java programs but whose specification is small enough to be read. Let *filter* be the function filtering out 0 from any list of integers. Here is the corresponding TimbukLTA specification.

```
Ops filter:1 nil:0 cons:2        Vars F X Y Z U Xs
TRS R1
  filter(nil) -> nil
  filter(cons(X,Y)) -> cons(X,filter(Y)) if >(X,0)
  filter(cons(X,Y)) -> cons(X,filter(Y)) if <(X,0)
  filter(cons(X,Y)) -> filter(Y)         if =(X,0)
Automaton A0 States qf qln qn Final States qf
Transitions filter(qln)->qf  nil->qln  cons(qn,qln)->qln  [-oo;+oo]->qn
Equations Approx Rules cons(X,Y)=Y
```

The automaton resulting of completion of `A0` by `R1` is the following:

```
States q0 q1 q2 q3 q4 q6 q7 q8  Final States q0  Transitions
[-oo,+oo]->q6  filter(q2)->q4  cons(q7,q0)->q4  cons(q8,q0)->q4   nil->q2
[-oo,+oo]->q3  filter(q2)->q0  cons(q6,q2)->q2  nil->q1
[-oo,-1]->q8   [1,+oo]->q7     cons(q8,q0)->q0  cons(q7,q0)->q0   nil->q0
```

This automaton recognizes all intermediate computations of *filter* and its result: lists of integers in $[-\infty, 1]$ or in $[1, +\infty]$ which is the expected result. To prove that there are no 0 in lists produced by filter, we could have seached for a *pattern*, *i.e.* a term of the form $cons(0, \_)$ and checked that it is not reachable.

7.2 An example without arithmetic : "Threads"

The first example deals with a bounded number of object creations, arithmetic operations but deals with several threads and their synchronisation.

```
class T1 extends java.lang.Thread{
    private int l;
    public T1(int l){this.l=l;}
    public void run(){
        while (true){
            synchronized(Top.lock){
                System.out.println(Top.f);
                Top.f=1;
                System.out.println(Top.f);
                Top.f=0;
            }
        }
    }
}

class Top{
    public static Object lock;
    public static int f;
    public static void main(String[] argv){
        int i=1;
        lock = new Object();
        Top.f=0;
        while (i<=2){
            T1 t1 = new T1(i++);
            t1.start();
        }
    }
}
```

From the bytecode of this Java program, Copster produces a TRS of 863 rewrite rules. The objective of the analysis is to prove that whatever the scheduling of threads, in the sequence of printed integers, there cannot be two consecutive 0. In other words, we aim at proving that the critical portion of the code which affects the `Top.f` shared variable is *really* protected by the Java `synchronize` instructions. Initially `Top.f` is 0. Each thread prints `Top.f`, set `Top.f` to the integer identifying the thread, prints it and then sets back `Top.f` to 0. This is repeated forever. If synchronization fails then several threads may execute this critical code at the same time and we are likely to print several consecutive 0 values. Since threads loop for ever, an approximation equation is necessary. On this simple example, the only term that may grow infinitely is the term representing the output stream. These terms are of the form $outstack(x, outstack(y, \ldots))$ representing an output stream whose last printed element is x and y was printed immediately before. Hence, a simple approximation equation of the form $outstack(x, outstack(y, z)) = z$ is enough for the standard completion to terminate in 56s after 306 steps. Then, on $\mathcal{A}^{306}_{\mathcal{R},E}$ we can easily check that the pattern $outstack(zero, outstack(zero, \_))$ is not found, meaning that we have no consecutive 0 printed in the output stream. On the same example, the conditional TRS has 788 rules and LTA completion with TimbukLTA terminates in 280s after 328 steps of LTA completion.

We can similarily check on $\mathcal{A}^{328}_{\mathcal{R},E}$ that no pattern of the form $outstack(0, outstack(0, \_))$ is found proving that synchronization was well performed. Thus LTA completion is abble to prove the same property. However, we can also remark that, on this example where arithmetic is not crucial for the analysis, using lattices in completion may reduce its efficiency. This is no longer the case when arithmetic is crucial for the analysis, as shown in the next example.

7.3 An example with arithmetic and recursive method calls : "Euclide"'

The second example only deals with arithmetic and recursive method calls. This example shows that LTA-completion is more efficient that regular completion as soon as some subterms deal with numbers and arithmetic operations. From the bytecode of this Java program, Copster produces a CTRS of 793 rewrite rules.

```
public class ExEuclide {
    static int my_div( int a, int b) {//returns a/b
if (a<b)
    return 0;
else
    return 1+my_div(a-b,b);
}

    static int my_mod (int a, int b){
if (a<b)
    return a;
else
    return my_mod(a-b,b);
}

public static void main (String[] args){
    int i,j,k;
    for (i=1; i<=5; i++){
     j=my_div(10*i,7);
     k=my_mod(10*i,7);
     System.out.println(k);
    }
  }
}
```

On this example, no approximation is needed and we can prove that the output stream only contains value in the interval $[1; 6]$ which is true for the remainders of the divisions of the integers 10 to 50 by the integer 7. We can prove the same result with standard and LTA completion. However, LTA completion only needs 727 steps and 14s where standard completion needs 2019 steps and 59s.

7.4 An example with arithmetic and object allocation: "FactoList"

The third example deals with object creations and some arithmetic. This example confirms that LTA-completion is more efficient that regular completion when arithmetic is concerned. From the bytecode of this Java program, Copster produces a CTRS of 805 rewrite rules.

```
class List{
    List next;
    int val;
    public List(int elt, List l){
        next =l;
        val= elt;
    }

    public void printList(){
        List l=this;
        while (l!=null){
                System.out.println(l.val);
                l=l.next;
        }
    }
}


public class FactoList {
        public static int factorial(int i){
                int res=1;
                for (int j=2; j<=i; j++){
                        res=res*j;
                }
                return res;
        }

        public static void main(String arg[]){
                 List ls= null;
                 int x=0;
                 while (x>=0) {
                         try {x=System.in.read();} catch(java.io.IOException e){};
                         if (x>=0) ls=new List(factorial(x),ls);
                }
                ls.printList();
        }
}
```

In this program, integers are read on the input channel and their factorial value is stored into a singly linked list. In the end, the content of the list is printed using the `printList` method. Completion can be used to show, for instance, that no integer lesser than 1 is printed out on the output channel whatever the integer read on the input channel may be. This time we need equations to over-approximate the integers, the heap (that may contain infinitely many objects) and the output channel that can contain infinitely many integers. This is done using three simple equations. The equation $outstack(x, y) = y$ merges infinite out stack terms together in a single equivalence class. The equation $stack1(\_, \_, \_) = stack1(\_, \_, \_)$ merges together all objects of class `List` (that are stored into terms rooted by the $stack1$ symbol in the generated TRS). Finally, the equations $succ(succ(x)) = succ(x)$ and $pred(pred(x)) = pred(x)$ merge all integers into three distinct equivalence classes, *i.e.* 0, 1 or more and -1 or less. Using those equations, in 20 seconds and 467 completion steps, standard completion produces an automaton $\mathcal{A}^{467}_{\mathcal{R},E}$ that does not contain any term of the form $outstack(zero, \_)$ or $outstack(pred(\_), \_)$ meaning that no integer lesser than 1 has been printed on the output stream. For LTA completion, the necessary equations are similar to approximate the heap and the output channel. Equations are slightly different to approximate the integers. Indeed, infinitely increasing integers values are automatically approximated using the widening described in section 4.3. However, terms built on built-in symbols may be infinite. Using the previous Java program, the infinite built-in terms that are likely to be built are of the form $+(+(+(\_, 1), 1), 1)$ because of the `j++` instruction and of the form $*(\_, *(\_, *(\_, \_)))$ because of the factorial operation. These two kinds of terms can be approximated using the following equations: $+(x, 1) = x$ and $*(x, y) = y$. In 40 seconds and 349 completion steps we can obtain a fixpoint by LTA completion. The same property, *i.e.* that no integer lesser than 1 is printed, can be proven on $\mathcal{A}^{349}_{\mathcal{R},E}$. Since the approximation of integers is strong and simple: only three equivalence classes, standard completion with peano integers still behave well w.r.t. LTA completion. However, as soon as more precision is needed on integers (requiring more equivalence classes on peano integers) then LTA completion outperforms standard completion.

If we restrict the values of the integers on the input stream then we can prove a more precise property on the values of the integers on the output stream. If we restrict input integers to values in the interval $[2; +\infty]$ (or -1 for the loop to stop), then we can prove that integers on the output stream are all greater to 1. For this, we also need to refine approximation equations on integers and approximate integers into 4 different equivalence classes -1 or less, 0, 1 and 2 or more. This is done by hand but could be automatized using a CEGAR-completion as in [10]. Then, the property can be proved using standard completion in 21 seconds and 468 completion steps. LTA completion needs 14 seconds and 430 completion steps. If we restrict the interval of input values to $[3; +\infty]$ and prove that output values are greater to 5 then standard completion needs 953 completion steps and 320 seconds where LTA completion is more stable and needs 467 completion steps and 15 seconds. With a set of input values restricted to $[4; +\infty]$ then standard completion exhausts memory and had to be stopped after 1500 completion steps and 2 hours. However, LTA completion answers in 32 seconds and 641 completion steps and proves that no integer lesser than 24 are printed on the output stream.

	Standard completion		LTA completion	
Example Name	# Completion steps	Completion time	# Completion steps	Completion time
Threads	306	56s	328	280s
Euclide	2019	59s	727	14s
FactoList inputs in $[-\infty; +\infty]$	467	20s	349	40s
FactoList inputs in $[2; +\infty]$	468	21s	430	14s
FactoList inputs in $[3; +\infty]$	953	320s	467	15s
FactoList inputs in $[4; +\infty]$	>1500	> 7200s	641	32s

This table shows that integration of LTA in completion may reduce its efficiency when the TRS to verify does not rely on arithmetic. On the opposite, unlike standard completion, LTA completion scales up when arithmetic is used in the analysis. This is illustrated on the "FactoList" example. The complete analysis of the TRS encoding the semantics of this program only deals with a few arithmetic operation. However, even for this limited number of arithmetic operation, when precision on the numerical values is expected, using lattices in completion is necessary to succeed.

8 Conclusion and Future work

We have proposed LTA, a new extension of tree automata for tree regular model checking of infinite-state systems whose configurations can be represented with interpreted terms. One of our main contributions is the development of a new completion algorithm for such automata. We also shown that our encoding can drastically improve the verification of TRS relying on arithmetic. This has been illustrated on the verification of *Java* programs translated into conditional TRS using TimbukLTA an implementation of the LTA completion. As LTA completion is not dedicated to the specific abstract domain of intervals of integers, we would like to plug several other abstract domains in TimbukLTA such as abstract domains for strings and reals. This is ongoing work.

References

1. P. A. Abdulla, B. Jonsson, P. Mahata, and J. d'Orso. Regular tree model checking. In *CAV*, volume 2404 of *LNCS*. Springer, 2002.
2. P. A. Abdulla, B. Jonsson, M. Nilsson, and J. d'Orso. Algorithmic improvements in regular model checking. In *CAV*, volume 2725 of *LNCS*. Springer, 2003.

3. P. A. Abdulla, A. Legay, A. Rezine, and J. d'Orso. Simulation-based iteration of tree transducers. In *TACAS*, volume 3440 of *LNCS*. Springer, 2005.
4. Parosh Aziz Abdulla, Giorgio Delzanno, and Ahmed Rezine. Parameterized verification of infinite-state processes with global conditions. In *CAV*, 2007.
5. Parosh Aziz Abdulla, Noomene Ben Henda, Giorgio Delzanno, Frédéric Haziza, and Ahmed Rezine. Parameterized tree systems. In *FORTE*, 2008.
6. Avispa – a tool for Automated Validation of Internet Security Protocols. `http://www.avispa-project.org`.
7. F. Baader and T. Nipkow. *Term Rewriting and All That*. Cambridge University Press, 1998.
8. N. Barré, F. Besson, T. Genet, L. Hubert, and L. Le Roux. Copster homepage, 2009. `http://www.irisa.fr/celtique/genet/copster`.
9. S Bauer, U. Fahrenberg, L. Juhl, K.G. Larsen, A. Legay, and C. Thrane. Quantitative refinement for weighted modal transition systems. In *MFCS*, volume 6907 of *lncs*. springer, 2011.
10. Y. Boichut, B. Boyer, T. Genet, and A. Legay. Equational Abstraction Refinement for Certified Tree Regular Model Checking. In *ICFEM'12*, volume 7635 of *LNCS*. Springer, 2012.
11. Y. Boichut, T. Genet, T. Jensen, and L. Leroux. Rewriting Approximations for Fast Prototyping of Static Analyzers. In *RTA*, LNCS. Springer Verlag, 2007.
12. Yohan Boichut, Benoît Boyer, Thomas Genet, and Axel Legay. Fast Equational Abstraction Refinement for Regular Tree Model Checking. Technical report, INRIA, July 2011. `http://hal.inria.fr/inria-00501487`.
13. Yohan Boichut, Pierre-Cyrille Héam, and Olga Kouchnarenko. Approximation-based tree regular model-checking. *Nord. J. Comput.*, 14(3):216–241, 2008.
14. B. Boigelot, A. Legay, and P. Wolper. Iterating transducers in the large (extended abstract). In *CAV*, LNCS. Springer, 2003.
15. B. Boigelot, A. Legay, and P. Wolper. Omega-regular model checking. In *TACAS*, volume 2988 of *LNCS*. Springer, 2004.
16. A. Bouajjani, P. Habermehl, A. Rogalewicz, and T. Vojnar. Abstract rmc of complex dynamic data structures. In *SAS*, LNCS. Springer, 2006.
17. A. Bouajjani and T. Touili. Extrapolating tree transformations. In *CAV*, volume 2404 of *LNCS*. Springer, 2002.
18. Ahmed Bouajjani, Peter Habermehl, Adam Rogalewicz, and Tomáš Vojnar. Abstract regular tree model checking. *Electron. Notes Theor. Comput. Sci.*, 149:37–48, February 2006.
19. H. Comon, M. Dauchet, R. Gilleron, C. Löding, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi. Tree automata techniques and applications, 2007.
20. Pichardie David. *Interprétation abstraite en logique intuitionniste : extraction d'analyseurs Java certifiés*. PhD thesis, Université Rennes 1, 2005. In french.
21. Zoltán Ésik and Guangwu Liu. Fuzzy tree automata. *Fuzzy Sets Syst.*, 158:1450–1460, July 2007.
22. G. Feuillade, T. Genet, and V. Viet Triem Tong. Reachability Analysis over Term Rewriting Systems. *JAR*, 33 (3-4):341–383, 2004.
23. Diego Figueira, Luc Segoufin, and Luc Segoufin. Bottom-up automata on data trees and vertical xpath. In *STACS*, 2011.
24. Tristan Le Gall and Bertrand Jeannet. Lattice automata: A representation for languages on infinite alphabets, and some applications to verification. In *SAS*, 2007.
25. Blaise Genest, Anca Muscholl, Zhilin Wu, and Zhilin Wu. Verifying recursive active documents with positive data tree rewriting. In *FSTTCS*, 2010.

26. T. Genet and F. Klay. Rewriting for Cryptographic Protocol Verification. In *cade*, volume 1831 of *lnai*. sv, 2000.
27. Th. Genet and V. Rusu. Equational approximations for tree automata completion. *Journal of Symbolic Computation*, 45(5):574–597, May 2010.
28. Stéphane Kaplan and Christine Choppy. Abstract rewriting with concrete operations. In *RTA*, pages 178–186, 1989.
29. Y. Kesten, O. Maler, M. Marcus, A. Pnueli, and E. Shahar. Symbolic model checking with rich assertional languages. In *CAV*, LNCS. Springer, 1997.
30. Orna Kupferman and Yoad Lustig. Lattice automata. In *VMCAI*, 2007.
31. J. Meseguer, M. Palomino, and N. Martï¿½-Oliet. Equational Abstractions. In *Proc. 19th CADE Conf., Miami Beach (Fl., USA)*, volume 2741 of *LNCS*, pages 2–16. Springer, 2003.
32. C. Otto, M. Brockschmidt, C. von Essen, and J. Giesl. Automated termination analysis of java bytecode by term rewriting. In *RTA*, LIPIcs. Dagstuhl, 2010.
33. P. Wolper and B. Boigelot. Verifying systems with infinite but regular state spaces. In *CAV*, volume 1427 of *LNCS*. Springer-Verlag, 1998.

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-0803