A Completion Algorithm for Lattice Tree Automata

Thomas Genet[1], Tristan Le Gall[2], Axel Legay[1], and Valérie Murat[1]

[1] INRIA/IRISA, Rennes, France
[2] CEA, LIST, Centre de recherche de Saclay, France

Abstract. When dealing with infinite-state systems, Regular Tree Model Checking approaches may have some difficulties to represent infinite sets of data. We propose Lattice Tree Automata, an extended version of tree automata to represent complex data domains and their related operations in an efficient manner. Moreover, we introduce a new completion-based algorithm for computing the possibly infinite set of reachable states in a finite amount of time. This algorithm is independent of the lattice making it possible to seamlessly plug abstract domains into a Regular Tree Model Checking algorithm. As a first instance, we implemented a completion with an interval abstract domain. We provide some experiments showing that this implementation permits to scale up regular tree model-checking of Java programs dealing with integer arithmetics.

1 Introduction

In verification, infinite-state models are often used to avoid assumptions on data structures and architectures, e.g. an artificial bound on the size of a stack or on the value of a 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 set of states. In Regular Tree Model Checking (RTMC), states are represented by trees, set of states by tree automata, and behavior of the system by tree transducers [1, 8] or rewriting rules [11, 16]. Any RTMC approach is equipped with an acceleration algorithm to compute possibly infinite sets of states in a finite amount of time. Among such algorithms, completion by equational abstraction [16] computes successive automata obtained by application of the rewriting rules, and merges intermediary states according to an equivalence relation to enforce the termination of the process.

In [6], the authors proposed an exact translation of the semantics of the Java Virtual Machine to tree automata and rewriting rules. This translation permits to analyze Java programs with Regular Tree 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 number of 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 [16], their combinations may require the use of artificially large structures.

We address this issue by defining *Lattice Tree Automata* (LTA). LTA have special transitions to abstract possibly infinite sets of values by a single element of a lattice. For example, we may abstract a set of integer values by a single interval instead of using an unary or binary encoding of those integers and recognizing the corresponding terms [6]. LTA recognize terms built over such intervals, and the completion algorithm built on LTA will perform each basic arithmetic operation in a single completion step, thanks to abstract interpretation techniques [9].

In this paper, we first define the LTA structure, then we propose a completion algorithm (by equational abstraction) that returns an approximation of the set of reachable states of an infinite-state systems whose behavior is modeled by rewriting rules. Finally, we provide some experimental results on the verification of Java programs using a RTMC environment. More details can be found in [15].

Related Work. [20] defined *lattice automata* to represent sets of words over an infinite alphabet. LTA are an extension of lattice automata to trees. Other models like modal automata [4] or data trees [12, 13] consider tree structure with infinite alphabets but do not exploit the lattice structure as we do. Lattice (-valued) automata [19]map words over a finite alphabet to a lattice value, while LTA map trees over an infinite alphabet to $\{0, 1\}$. Similar automata may define fuzzy tree languages [10]. Verification of particular classes of properties of Java programs with interpreted terms can be found in [23].

Many techniques aim at the verification of programs with integer arithmetics. Among them, abstract interpretation [9] computes over-approximations of reachability sets, but requires a complete evaluation of arithmetic expressions. LTA can handle expressions that are only partially evaluated, thus may be useful in interprocedural analysis. There are other ways to deal with arithmetic efficiently in a regular model-checking framework such as [21]. However, we think that LTA provide a way to abstract many different types of data (integers, strings, etc.) by simply plugging the adapted abstract domain (and using its best available implementation) in a RTMC framework. In particular, LTA could be used by other RTMC techniques like [1, 8] where such an ability does not exist.

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). $\mathcal{V}ar(t)$ denotes the set of variables of a term t, and $\mathcal{F}^n$, the set of functional symbols of arity n. We denote by $Pos(t)$ the set of positions of a term t, *i.e.* the set of positions of all its subterms, where a position is a world over $\mathbb{N}$ and ε denotes the top-most position. 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 of $\mathcal{R}$ is left-linear.

We now define Tree Automata 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 $\mathcal{T}(\mathcal{F} \cup \mathcal{Q})$. 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 $\mathcal{A} = \langle \mathcal{F}, \mathcal{Q}, \mathcal{Q}_F, \Delta \rangle$, 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* $\bot$ 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.* $\bot \sqsubset x \wedge \forall y \in \Lambda : \bot \sqsubset y \sqsubseteq x \Rightarrow y = x$. The set of atoms of Λ is denoted by $Atoms(\Lambda)$. A lattice $(\Lambda, \sqsubseteq)$ is atomic if any element $x \in \Lambda$ where $x \neq \bot$ 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), there is a *Galois connection* between the two 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 $(I, \sqsubseteq)$ 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 [18]. Then we propose a new type of tree automata recognizing terms with elements of an abstract lattice.

3.1 Interpreted Symbols and Evaluation

In what follows, elements of a possibly infinite concrete domain $\mathcal{D}$ will be represented by a set of *interpreted* symbols $\mathcal{F}_\bullet$. The set of symbols is now $\mathcal{F} = \mathcal{F}_\circ \cup \mathcal{F}_\bullet$, where $\mathcal{F}_\circ$ is the set of *passive* (uninterpreted) symbols. The set of *interpreted* symbols $\mathcal{F}_\bullet$ is composed of elements of $\mathcal{D}$ (notice that $\mathcal{D} \subseteq \mathcal{F}^0_\bullet$), and is also composed of some predefined operations $op : \mathcal{D}^n \to \mathcal{D}$, where $op \in \mathcal{F}^n_\bullet$ and $n > 0$. We denote by OP the set of predefined operations, thus we have $\mathcal{F}_\bullet = \mathcal{D} \cup OP$. 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 $\mathcal{F}_\bullet$ (called interpreted terms) can be evaluated by using an eval function $eval : \mathcal{T}(\mathcal{F}_\bullet) \to \mathcal{D}$. The purpose of *eval* is to simplify a term using

the built-in operations of the domain $\mathcal{D}$. *eval* naturally extends to $\mathcal{T}(\mathcal{F})$: (1) $eval(f(t_1,\ldots,t_n)) = f(eval(t_1),\ldots,eval(t_n))$ if $f \in \mathcal{F}_\circ$ or $\exists i = 1\ldots n : t_i \notin \mathcal{T}(\mathcal{F}_\bullet)$, or (2) the evaluation returns an element of $\mathcal{D}$ if $f(t_1,\ldots,t_n) \in \mathcal{T}(\mathcal{F}_\bullet)$.

We want to define tree automata to recognize sets of interpreted terms. To recognize $\{f(1), f(2), f(3), f(4)\}$, we would like to have tree automata with special transitions to handle sets of integers for instance: $\{1,\ldots,4\} \to q, f(q) \to q_f$. We propose 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 $(2^{\mathbb{Z}}, \subseteq)$). By considering generic lattices, we can also improve the efficiency of the approach. Since RTMC only requires an over-approximation of the set of reachable states, we have special transitions to recognize elements of a simple, *abstract lattice* $(\Lambda, \sqsubseteq)$ such as the lattice of intervals. Moreover, we assume that this abstract lattice is atomic (cf. Section 2).

Each built-in operation $op \in OP$ defined on $\mathcal{D}$, is also abstracted by $op^\# \in OP^\#$. Since we have that $\mathcal{F}_\bullet = \mathcal{D} \cup OP$, the set of *abstract* symbols is $\mathcal{F}_\bullet^\# = \Lambda \cup OP^\#$. The arity of $op^\#$ is the same as the one of op. Assuming there is a Galois connection between the concrete domain and the abstract one (cf. Section 2), then $op^\# = \alpha \circ op \circ \gamma$ and $eval^\# : \mathcal{T}(\mathcal{F}_\bullet^\#) \mapsto \Lambda$ is the best approximation of $eval$.

Example 1. There is a Galois connection between $(2^{\mathbb{Z}}, \subseteq)$ and the lattice of intervals $(I, \sqsubseteq)$. $eval^\#([2,3] +^\# [-1,2]) = [1,5])$.

3.2 Definition and Semantics

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

The set of *lambda transitions*, which recognize elements of the lattice, 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 formally defined by $\Delta_G = \{f(q_1,\ldots,q_n) \to q \mid f \in \mathcal{F} \wedge f(q_1,\ldots,q_n) \to q \in \Delta \wedge q, q_1, \ldots, q_n \in Q\}$. *Epsilon transitions* are transitions of the form $q \to q'$ where $q, q' \in \mathcal{Q}$. 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) $eval(s) \sqsubseteq eval(t)$ (if both s and t belong to $\mathcal{T}(\mathcal{F}_\bullet^\#)$), or (2) $s = f(s_1,\ldots,s_n)$, $t = f(t_1,\ldots,t_n)$, $f \in \mathcal{F}_\circ^n$ and $s_1 \sqsubseteq t_1 \wedge \ldots \wedge s_n \sqsubseteq t_n$.

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 may omit # on abstract operations when it is clear from the context. We now define the transition relation and recognized language of an LTA. A term t is recognized by an LTA $\mathcal{A}$ if $eval(t)$ can be reduced in $\mathcal{A}$.

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

- *if* $t_1|_p \in \mathcal{T}(\mathcal{F}_\bullet^\#)$*, there is a transition* $\lambda \to q \in \Delta$ *such that* $eval(t_1|_p) \sqsubseteq \lambda$ *and* $t_2 = t_1[q]_p$
- *if* $t_1|_p = q$ *where* $q \in \mathcal{Q}$*, there is an epsilon-transition* $q \to q' \in \Delta$*, where* $q' \in \mathcal{Q}$ *such that* $t_2 = t_1[q']_p$
- *if* $t_1|_p = f(s_1, \ldots, s_n)$ *where* $f \in \mathcal{F}^n$ *and* $s_1, \ldots s_n \in \mathcal{T}(\mathcal{F} \cup \mathcal{Q})$*,* $\exists s_i' \in \mathcal{T}(\mathcal{F} \cup \mathcal{Q})$ *such that* $s_i \to_\mathcal{A} s_i'$ *and* $t_2 = t_1[f(s_1, \ldots, s_{i-1}, s_i', s_{i+1}, \ldots, s_n)]_p$
- *if* $t_1|_p = f(q_1, \ldots, q_n)$ *where* $f \in \mathcal{F}^n$ *and* $q_1, \ldots q_n \in \mathcal{Q}$*, there is a transition* $f(q_1, \ldots, q_n) \to q \in \Delta$ *such that* $t_2 = t_1[q]_p$*.*

$\to_\mathcal{A}^*$ is the reflexive transitive closure of $\to_\mathcal{A}$. There is a run from t_1 to t_2 if $t_1 \to_\mathcal{A}^* t_2$. If a LTA has a transition $[0, 2] \to q$ then $[0, 0] \to_\mathcal{A}^* q$, $[1, 2] \to_\mathcal{A}^* q$, $\ldots$, *i.e.* all possible unions of atoms $[0, 0]$,$[1, 1]$,$[2, 2]$. The language recognized by a LTA is thus defined over $\mathcal{T}(\mathcal{F}, Atoms(\Lambda))$, where $\mathcal{T}(\mathcal{F}, Atoms(\Lambda))$ is the set of ground terms built over $(\mathcal{F} \setminus \Lambda) \cup Atoms(\Lambda)$.

Definition 4 (Recognized language). *The tree language recognized by* $\mathcal{A}$ *in a state* q *is* $\mathcal{L}(\mathcal{A}, q) = \{t \in \mathcal{T}(\mathcal{F}, Atoms(\Lambda)) \mid \exists\ t' \text{ such that } t \sqsubseteq t' \text{ and } t' \to_\mathcal{A}^* q\}$*. 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 $\mathcal{Q}_f = \{q_2\}$. We have: $f([1, 4]) \to_\mathcal{A}^* q_2$ and $f([0, 1] + [0, 1]) \to_\mathcal{A}^* q_2$, and the recognized language of $\mathcal{A}$ is given by $\mathcal{L}(\mathcal{A}, q_2) = \{f([0, 0]), f([1, 1]), \ldots, f([4, 4])\}$.

4 Completion Algorithm

We only present here the completion algorithm on LTA, other operations are detailed in [15]. We are interested in computing the set of reachable states of an infinite state system. We propose to represent states by (built-in) terms and possibly infinite set of states by an LTA. In this section, we assume that the behavior of the system can be represented by conditional term rewriting systems, *i.e. TRS* equipped with conjunction of conditions used to restrain the applicability of the rule. Our conditional *TRS*, which extends the classical definition of [2], 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 in *TRS*.

Definition 5 (Predicates). *Let* $\mathcal{P}$ *be the set of predicates over* $\mathcal{D}$*. Let* ρ *be a* n*-ary predicate of* $\mathcal{P}$ *such that* $\rho : \mathcal{D}^n \mapsto \{true, false\}$*. We extend the domain of* ρ *to* $\mathcal{T}(\mathcal{F})^n$ *in the following way:*

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

Observe that 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 6 (Conditional Term Rewriting System (*CTRS*) on $\mathcal{T}(\mathcal{F}_\circ \cup \mathcal{F}_\bullet, \mathcal{X})$). *In our setting, a Conditional Term Rewriting System $\mathcal{R}$ is a set of* rewrite rules $l \to r \Leftarrow c_1 \wedge \ldots \wedge c_n$, *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* $\forall i = 1 \ldots n : c_i = \rho_i(t_1, \ldots, t_m)$ *where* ρ_i *is a* m*-ary predicate of* $\mathcal{P}$ *and* $\forall j = 1 \ldots m : t_j \in \mathcal{T}(\mathcal{F}_\bullet, \mathcal{X}) \wedge \mathcal{V}ar(t_j) \subseteq \mathcal{V}ar(l)$.

Example 4. Using conditional rewriting rules, the factorial can be encoded by the CTRS: $fact(x) \to 1 \Leftarrow x \geq 0 \wedge x \leq 1$, $fact(x) \to x * fact(x-1) \Leftarrow x \geq 2$.

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 $t = f(t_1, \ldots, t_n)$ then $t\sigma = f(t_1\sigma, \ldots, t_n\sigma)$, where $t, t_1, \ldots, t_n \in \mathcal{T}(\mathcal{F}, \mathcal{X}), f \in \mathcal{F}^n$, (2) if $t = x \in \mathcal{X}$ then $t\sigma = \sigma(x)$. Recall that $\mathcal{F} = \mathcal{F}_\circ \cup \mathcal{F}_\bullet$. The *CTRS* $\mathcal{R}$ and the *eval* function induces a rewriting relation $\to_\mathcal{R}$ on $\mathcal{T}(\mathcal{F})$: in the following way: for all $s, t \in \mathcal{T}(\mathcal{F})$, we have $s \to_\mathcal{R} t$ if there exist: (1) a rewrite rule $l \to r \Leftarrow c_1 \wedge \ldots \wedge c_n \in \mathcal{R}$, (2) a position $p \in \mathcal{P}os(s)$, and (3) a substitution $\sigma : \mathcal{X} \mapsto \mathcal{T}(\mathcal{F})$ s.t. $s|_p = l\sigma$, $t = eval(s[r\sigma]_p)$ and $\forall i = 1 \ldots n : c_i\sigma = true$. The reflexive transitive closure of $\to_\mathcal{R}$ is denoted by $\to^*_\mathcal{R}$.

Let $\mathcal{A}$ be an LTA representing the set of initial states, and $\mathcal{R}$ be a *CTRS*. Our objective is to compute another LTA representing (an over-approximation of) the set $\mathcal{R}^*(\mathcal{L}(\mathcal{A})) = \{t \mid \exists t_0 \in \mathcal{L}(\mathcal{A}), t_0 \to^*_\mathcal{R} t\}$. We adopt the completion approach of [16, 11], 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 *CTRS* $\mathcal{R}$. The algorithm proceeds by computing the sequence of automata $\mathcal{A}^0_\mathcal{R}, \mathcal{A}^1_\mathcal{R}, \mathcal{A}^2_\mathcal{R}, \ldots$ 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 that will be described in section 4.3. We now give details on the above constructions, which will be illustrated step by step by a running example.

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 $l \to r \Leftarrow c_1 \wedge \ldots \wedge c_n \in \mathcal{R}$, 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 $l\sigma \to^*_{\mathcal{A}^i_\mathcal{R}} q$, $\sigma \sqsupseteq \sigma'$ and $c_1\sigma' \wedge \ldots \wedge c_n\sigma'$. 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 $r\sigma' \not\to^*_{\mathcal{A}^i_\mathcal{R}} q$, the algorithm adds two new transitions $r\sigma' \to q'$ and $q' \to q$ to $\mathcal{A}^i_\mathcal{R}$, in order to enrich the language of the previous automaton. To find all critical pairs, in what follows, we use the standard *matching algorithm* introduced and described in [11]. This algorithm $Matching(l, \mathcal{A}, q)$ matches a linear term l with a state q in the automaton $\mathcal{A}$. The solution returned by *Matching* is a set of substitutions $\{\sigma_1, \ldots, \sigma_n\}$ so that $l\sigma_i \to^*_\mathcal{A} q$. 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 $l \to r \Leftarrow c_1 \wedge \ldots \wedge c_n$ satisfies $c_1 \wedge \ldots \wedge c_n$.

Example 5. Let $\mathbb{Z}$ be the concrete domain, and intervals on $\mathbb{Z}$ be the lattice, $\mathcal{R} = \{f(x) \to cons(x, f(x+1)) \Leftarrow x \geq 1\}$ be the $CTRS$, $\mathcal{A}_0$ the LTA representing the set of initial configurations, with transitions: $\Delta_0 = \{[0,2] \to q_1, f(q_1) \to q_2\}$. To build $\mathcal{A}^1_{\mathcal{R}}$ from $\mathcal{A}_0$, we have to find all possible substitutions. The matching algorithm tells that the rewrite rule applies with the substitution $\{x \mapsto q_1\}$. To satisfy the constraint $x \geq 1$, the substitution $\{x \mapsto q_1\}$ with $[0,2] \to q_1$ will be restricted to $\{x \mapsto [1,2]\}$.

Restricting substitutions is done by a solver *Solve* on abstract domains. The output of $Solve(\sigma, \mathcal{A}, c_1 \wedge \ldots \wedge c_n)$ is either a set of substitutions σ' which is a restriction of σ satisfying $c_1 \wedge \ldots \wedge c_n$ or $\emptyset$ if such a restriction does not exist. On the previous example, $Solve(\{x \mapsto q_1\}, \mathcal{A}, x \geq 1) = \{\{x \mapsto [1,2]\}\}$. See [15] for details about the properties the solver needs to have. Such properties are generally fulfilled by usual abstract domains implementations.

Definition 7 (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 must add the rules $r\sigma_i \to q'$ and $q' \to q$ in the automaton, where q' is a new state. However, $r\sigma_i \to q'$ 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 it must be normalized first in order to be added to the LTA.

Definition 8 (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 CTRS, $\mathcal{F}^\#_\bullet$ the set of abstract symbols used in $\mathcal{A}$, $\mathcal{F} = \mathcal{F}^\#_\bullet \cup \mathcal{F}_\circ$, and $\alpha : \mathcal{F}^0_\bullet \to \mathcal{F}^{\#^0}_\bullet$ the abstraction function, mapping concrete constants to elements of 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 defined by:*

1. *if $s \in \mathcal{F}^0_\bullet$ then $Norm(s \to q) = \{\alpha(s) \to q\}$.*
2. *if $s \in \mathcal{F}^0_\circ \cup \mathcal{F}^{\#^0}_\bullet$ then $Norm(s \to q) = \{s \to q\}$,*
3. *if $s = f(t_1, \ldots, t_n)$ where $f \in \mathcal{F}^n_\circ \cup \mathcal{F}^n_\bullet$, 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.*

Example 6. From Ex.5, we have to add the normalized form of $cons([1,2], f([1,2]+1)) \to q'_2$ and $q'_2 \to q_2$ (where q'_2 is a new state) to the set of transitions: 1 has to be abstracted by $[1,1]$ and $f([1,2])$ has to be replaced by a state recognizing this term. So $\Delta_1 = \Delta_0 \cup Norm(cons([1,2], f([1,2]+1)) \to q'_2) \cup \{q'_2 \to q_2\} = \Delta_0 \cup \{[1,2] \to q_3, [1,1] \to q_{[1,1]}, q_3 + q_{[1,1]} \to q_4, f(q_4) \to q_5, cons(q_3, q_5) \to q'_2, q'_2 \to q_2\}$, where $q_{[1,1]}, q_3, q_4, q_5$ are new states induced by normalization.

Observe that the normalization algorithm always terminates. We conclude by the formal characterization of the one step completion.

Definition 9 (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 CTRS. 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.7, $q' \notin \mathcal{Q}$ a new state and $\mathcal{Q}'$ contains all the states of Δ'.

4.2 Evaluation of a Lattice Tree Automaton

Any set of concrete terms that contains the term $1 + 2$ should also contain the term 3. While this property can be true on the initial automaton, it may be broken when performing a completion step.

Example 7. The first completion step described in Ex.6 adds the transition $q_3 + q_{[1,1]} \to q_4$. Since we have that $[1, 2] \to q_3$ and $[1, 1] \to q_{[1,1]}$, the language recognized by q_4 should also contain the term $[2, 3]$.

The objective of the *propag* function is to evaluate the LTA and to add the transition $[2, 3] \to q_4$ in the above example.

Definition 10 (*propag*). *Let Δ be the set of transitions of a LTA. Let $f(q_1, \ldots, q_n) \to q \in \Delta$, where $f \in \mathcal{F}^n_\bullet$ is an interpreted symbol and $q, q_1, \ldots, q_n \in \mathcal{Q}$. If there exists $\lambda_1, \ldots, \lambda_n \in \Lambda$ such that $\lambda_1 \to^*_\Delta q_1, \ldots, \lambda_n \to^*_\Delta q_n$, then one step of evaluation of $f(q_1, \ldots, q_n) \to q$ is defined by:*

$$propag(\Delta, f(q_1, \ldots, q_n) \to q) = \begin{cases} \Delta \text{ if } \exists \lambda \to q \in \Delta \wedge eval(f(\lambda_1, \ldots, \lambda_n)) \sqsubseteq \lambda \\ \Delta \cup \{eval(f(\lambda_1, \ldots, \lambda_n)) \to q\}, otherwise. \end{cases}$$

One step of evaluation for Δ is defined by:

$$propag(\Delta) = \bigcup_{\forall f(q_1,\ldots,q_n) \to q \in \Delta \ s.t. \ f \in \mathcal{F}^n_\bullet} propag(\Delta, f(q_1, \ldots, q_n) \to q)$$

Since *propag* can add new transitions, it must be applied until a fix-point is reached. Then using *propag*, we can extend the *eval* function to sets of transitions and to tree automata in the following way.

Definition 11 (*eval* on transitions and automata). *$\mu X.f(X)$ denotes the least fix-point of a generic function f. We define: $eval(\Delta) = \mu X.propag(X) \cup \Delta$ and $eval(\langle \mathcal{F}, \mathcal{Q}, \mathcal{Q}_f, \Delta \rangle) = \langle \mathcal{F}, \mathcal{Q}, \mathcal{Q}_f, eval(\Delta) \rangle$.*

Example 8. In our example, $eval(\Delta_1) = \Delta_1 \cup \{[2, 3] \to q_4\}$.

Theorem 1. *$\mathcal{L}(\mathcal{A}) \subseteq \mathcal{L}(eval(\mathcal{A}))$.*

4.3 Equational Abstraction

If we perform another completion step on our example, we see that we can apply the rewrite rule with a new substitution mapping x to q_4.

Example 9. Then $Norm(cons(q_4, f(q_4+1))) \to q'_5$ and $q'_5 \to q_5$ will be added to $eval(\Delta_1)$ to build $\mathcal{A}^2_{\mathcal{R}}$ from $\mathcal{A}^1_{\mathcal{R}}$. We have $\Delta_2 = eval(\Delta_1) \cup \{q_4 + q_{[1,1]} \to q_6, f(q_6) \to q_7, cons(q_4, q_7) \to q'_5, q'_5 \to q_5\}$. If we perform the evaluation step, we have $eval(\Delta_2) = \Delta_2 \cup \{[3,4] \to q_6\}$. We can see that this process is infinite, because it will compute the infinite term $cons([1,2], cons([2,3], cons([3,4], \ldots)))$.

Termination of completion can be enforced using a set E of *approximation equations* as in [22, 16]. Depending on the objective, E can either be defined by hand (e.g. [22]), by hand and automatically refined [5], or automatically generated from a static analysis of the *TRS* (e.g. [7]). In our example, the infinite behavior is due to transitions of the form $q_i + q_{[1,1]} \to q_j$. An equation such as $x = x + 1$ is needed to ensure termination of completion. Equations of E will be of the form $u = v \Leftarrow c_1 \wedge \ldots \wedge c_n$, where $u, v \in \mathcal{T}(\mathcal{F}_\circ \cup \mathcal{F}_\bullet, \mathcal{X})$. Let $\sigma : \mathcal{X} \mapsto \mathcal{Q}$ be a substitution s.t. $u\sigma \to_{\mathcal{A}^{i+1}_{\mathcal{R}}} q$, $v\sigma \to_{\mathcal{A}^{i+1}_{\mathcal{R}}} q'$ and $q \neq q'$. An over-approximation of $\mathcal{A}^{i+1}_{\mathcal{R}}$ (denoted by $\mathcal{A}^{i+1}_{\mathcal{R},E}$) can be obtained by merging states q and q', *i.e.* replacing each occurrence of q' by q in $\mathcal{A}^{i+1}_{\mathcal{R}}$. 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., $Solve(\sigma, \mathcal{A}, c_1 \wedge \cdots \wedge c_n) \neq \emptyset$.

For instance, $E = \{x = x + 1 \Leftarrow x > 2\}$ can be used on Ex.9. We have two possible substitutions: $\sigma_1 = \{x \mapsto q_3\}$ and $\sigma_2 = \{x \mapsto q_4\}$. σ_1 is due to the transition $q_3 + q_{[1,1]} \to q_4$. However, since $[1,2] \to q_3$ we have $Solve(\{x \mapsto q_3\}, \mathcal{A}_2, x > 2) = \emptyset$ and thus σ_1 does not satisfy the condition. Substitution σ_2, due to the transition $q_4 + q_{[1,1]} \to q_6$, satisfies the condition because $[2,3] \to q_4$ and $Solve(\{x \mapsto q_4\}, \mathcal{A}_2, x > 2) = \{x \mapsto [3,3]\} \neq \emptyset$. Hence, the equation is applied for σ_2 and results in the merging of q_4 and q_6 according to E.

Theorem 2. *Let $\mathcal{A}$ be an LTA and E a set of equations. We denote by $\rightsquigarrow^!_E$ the transformation of $\mathcal{A}$ by merging all equivalent states according to E. If $\mathcal{A} \rightsquigarrow^!_E \mathcal{A}'$ then $\mathcal{L}(\mathcal{A}) \subseteq \mathcal{L}(\mathcal{A}')$.*

Widening Step. Any set containing the term $1+2$ should also contain the term 3. However, this can be broken by merging. Merging of states changes transitions of the LTA. So we have to perform an evaluation step after merging by equations.

Example 10. After merging q_4 and q_6, we have $Merge(\Delta_2, q_4, q_6) = eval(\Delta_1) \cup q_3 + q_{[1,1]} \to q_4, f(q_4) \to q_5, cons(q_3, q_5) \to q'_2, q'_2 \to q_2, [2,3] \to q_4, q_4 + q_{[1,1]} \to q_4, f(q_4) \to q_7, cons(q_4, q_7) \to q'_5, q'_5 \to q_5, [3,4] \to q_4\}$. We have to evaluate the transition $q_4 + q_{[1,1]} \to q_4$. The first iteration will evaluate the term $[3,4] + [1,1]$ which adds the transition $[4,5] \to q_4$. Since a new element is in the state q_4, the second iteration will evaluate the term $[4,5] + [1,1]$ recognized by the transition $q_4 + q_{[1,1]} \to q_4$. Since there will always be a new element of the lattice that will be associated to q_4, the computation of the evaluation will not terminate.

Since *eval* is defined as a fix-point of *propag*, this computation may not terminate without the application of a widening operator $\nabla_\Lambda : \Lambda \times \Lambda \mapsto \Lambda$. It is a classical way to compute over-approximation of fix-points within the abstract interpretation framework [9].

Example 11. If we apply such a widening operator on our example after 3 iterations (for instance) of the *propag* function, then the transitions: $[2,3] \to q_4$, $[3,4] \to q_4$, $[4,5] \to q_4$ will be replaced by $[2,+\infty[\to q_4$.

4.4 LTA Completion and its Soundness

Definition 12 (Automaton completion for LTA). *Let $\mathcal{A}$ be a tree automaton, $\mathcal{R}$ a CTRS and E a set of equations.*

- $\mathcal{A}^0_{\mathcal{R},E} = \mathcal{A}$,
- *Repeat* $\mathcal{A}^{n+1}_{\mathcal{R},E} = \mathcal{A}'$ *with* $eval(\mathcal{C}_\mathcal{R}(\mathcal{A}^n_{\mathcal{R},E})) \rightsquigarrow^!_E \mathcal{A}''$ *and* $eval(\mathcal{A}'') = \mathcal{A}'$,
- *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 reached.*

Theorem 3 (Soundness). *Let $\mathcal{R}$ be a left-linear CTRS, $\mathcal{A}$ be a tree automaton and E be a set of linear equations. If completion terminates on $\mathcal{A}^*_{\mathcal{R},E}$ then* $\mathcal{L}(\mathcal{A}^*_{\mathcal{R},E}) \supseteq \mathcal{R}^*(\mathcal{L}(\mathcal{A}))$

Example 12. In our example, thanks to the widening performed at the previous evaluation step, completion adds no more rule to the current automaton and stops. We have a fixed-point which is an over-approximation of the set of reachable states.

5 Experiments

LTA completion has been developed and integrated into Timbuk [14]. For those experiments, we choose to instantiate the generic LTA-completion algorithm with the lattice of integer intervals: TimbukLTA. Experiments are detailed in [15].

We compare the efficiency of LTA completion w.r.t. standard completion on *TRS* produced by Copster [3]. Copster compiles *Java* `.class` files into a *TRS* modeling exactly the semantics of the Java program[3]. We extend Copster to produce either *TRSs* or conditional *TRS* (*CTRS*) as in Section 4. *CTRSs* do not use Peano integers or arithmetic but assume that all integer arithmetic is built-in. On the Java program examples, we prove the same properties using either Timbuk or TimbukLTA and compare their efficiency. We made several experiments on three different Java programs that are detailed in [15]. On the first one, called "Threads", we prove that whatever the scheduling of Java threads may be, the access to a critical section is protected using the `synchronized` Java mechanism. The second one "Euclid" consists of an implementation of integer

[3] Copster covers basic types, arithmetic, object creation, heap management, field manipulation, virtual method invocation, threads, as well as a subset of the System and String library. Details about this compilation can be found in [6].

division in a recursive way using addition and subtraction. In the third one, called "FactoList", there is an unbounded number of integers which are read on the input channel and their factorial values are stored into a singly linked list. In the end, the content of the list is printed to the output stream. Depending on the possible values for integers read on the input stream, we can prove different properties on the integers printed on the output stream.

Examples	Standard completion		LTA completion	
	Compl. steps	Compl. time	Compl. steps	Compl. time
Threads	306	56s	328	280s
Euclid	2019	59s	727	14s
FactoList, input stream=$(3, 1, 2, 0)$	799	17s	538	33s
FactoList, input stream=$(7, 5, 6, 4, 1)$	>9465	>2h	1251	250s
FactoList, any input stream of $[-\infty; +\infty]$	467	20s	349	40s
FactoList, any input stream of $[2; +\infty]$	468	21s	430	14s
FactoList, any input stream of $[3; +\infty]$	953	320s	467	15s
FactoList, any input stream of $[4; +\infty]$	>1500	> 2h	641	32s

Table 1. Performances of standard completion against LTA completion

Table 1 shows that integration of LTA in completion may reduce its efficiency when the *TRS* to verify does not rely on arithmetic ("Threads" example). On the opposite, unlike standard completion, LTA completion scales up when arithmetic is used in the analysis ("Euclid" and "FactoList" example). TimbukLTA and the adapted Copster can be downloaded from their respective pages [14, 3].

6 Conclusion and Future Work

We have proposed LTA, a new extension of tree automata for tree regular model checking of infinite-state systems with interpreted terms. One of our main contributions is the development of a new completion algorithm for such automata. A nice property of this adapted algorithm is that it is independent of the lattice: it only has to be atomic and equipped with a solver for the predicates of the $CTRS$ [15]. Any lattice fulfilling those requirements can be seamlessly plugged into the regular tree model checking algorithm. We developed TimbukLTA which is the implementation of completion for LTA. We presented a first instance of TimbukLTA where we plugged in an integer interval abstract domain. This simple abstract domain permitted to drastically improve the efficiency of completion for the verification of Java programs dealing with integer arithmetic. The resulting LTA homogeneously combine abstract domains to approximate numerical values with tree automata to approximate structures: thread states, stacks, heaps and objects. Future plans are to integrate in TimbukLTA more abstract domains dealing with other kinds of built-ins: strings, reals, etc. and to define syntactic constraints on equations to guarantee termination of LTA completion like in [17].

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. F. Baader and T. Nipkow. *Term Rewriting and All That*. Cambridge University Press, 1998.
3. N. Barré, F. Besson, T. Genet, L. Hubert, and L. Le Roux. Copster homepage, 2009. http://www.irisa.fr/celtique/genet/copster.
4. 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.
5. 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.
6. Y. Boichut, T. Genet, T. Jensen, and L. Leroux. Rewriting Approximations for Fast Prototyping of Static Analyzers. In *RTA*, LNCS. Springer Verlag, 2007.
7. Y. Boichut, P.-C. Héam, and O. Kouchnarenko. Approximation-based tree regular model-checking. *Nord. J. Comput.*, 14(3):216–241, 2008.
8. A. Bouajjani and T. Touili. Extrapolating tree transformations. In *CAV*, volume 2404 of *LNCS*. Springer, 2002.
9. P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL*, pages 238–252, 1977.
10. Z. Ésik and G. Liu. Fuzzy tree automata. *Fuzzy Sets Syst.*, 158:1450–1460, July 2007.
11. G. Feuillade, T. Genet, and V. Viet Triem Tong. Reachability Analysis over Term Rewriting Systems. *jar*, 33 (3-4):341–383, 2004.
12. D. Figueira and L. Segoufin. Bottom-up automata on data trees and vertical xpath. In *STACS*, 2011.
13. B. Genest, A. Muscholl, and Z. Wu. Verifying recursive active documents with positive data tree rewriting. In *FSTTCS*, 2010.
14. T. Genet. Timbuk. http://www.irisa.fr/celtique/genet/timbuk/.
15. T. Genet, T. Le Gall, A. Legay, and V. Murat. Tree regular model checking for lattice-based automata. Technical Report RT-0424, INRIA, 2012. http://hal.inria.fr/hal-00687310.
16. T. Genet and V. Rusu. Equational approximations for tree automata completion. *Journal of Symbolic Computation*, 45(5):574–597, May 2010.
17. T. Genet and Y. Salmon. Tree Automata Completion for Static Analysis of Functional Programs. Technical report, INRIA, 2013. http://hal.archives-ouvertes.fr/hal-00780124/PDF/main.pdf.
18. S. Kaplan and C. Choppy. Abstract rewriting with concrete operations. In *RTA*, pages 178–186, 1989.
19. O. Kupferman and Y. Lustig. Lattice automata. In *VMCAI*, 2007.
20. T. Le Gall and B. Jeannet. Lattice Automata: A Representation for Languages on Infinite Alphabets, and Some Applications to Verification. In *SAS*, 2007.
21. J. Leroux. Structural Presburger digit vector automata. *TCS*, 409(3), 2008.
22. J. Meseguer, M. Palomino, and N. Mart-Oliet. Equational Abstractions. In *Proc. 19th CADE Conf., Miami Beach (Fl., USA)*, volume 2741 of *LNCS*, pages 2–16. Springer, 2003.
23. C. Otto, M. Brockschmidt, C. von Essen, and J. Giesl. Automated termination analysis of java bytecode by term rewriting. In *RTA*, LIPIcs. Dagstuhl, 2010.