Equational Abstraction Refinement for Certified Tree Regular Model Checking

Yohan Boichut, Benoit Boyer, Thomas Genet, Axel Legay

Equational Abstraction Refinement for Certified Tree Regular Model Checking

Y. Boichut[1], B. Boyer[4], T. Genet[2], and A. Legay[3]

[1] LIFO - Université Orléans, France
[2] IRISA - Université Rennes 1, France
[3] INRIA - Rennes, France
[4] VERIMAG - Université Joseph Fourier, France

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 trees and sets of states by tree automata. The central problem is to decide whether a set of bad states belongs to the set of reachable states. An obstacle is that this set is in general neither regular nor computable in finite time.
This paper proposes a new CounterExample Guided Abstraction Refinement (CEGAR) algorithm for TRMC. Our approach relies on a new equational-abstraction based completion algorithm to compute a regular overapproximation of the set of reachable states in finite time. This set is represented by $\mathcal{R}_{/E}$-automata, a new extended tree automaton formalism whose structure can be exploited to detect and remove false positives in an efficient manner. Our approach has been implemented in TimbukCEGAR, a new toolset that is capable of analyzing Java programs by exploiting an elegant translation from the Java byte code to term rewriting systems. Experiments show that TimbukCEGAR outperforms existing CEGAR-based completion algorithms. Contrary to existing TRMC toolsets, the answers provided by TimbukCEGAR are certified by Coq, which means that they are formally proved correct.

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 this paper, we rely on Tree Regular Model Checking (TRMC) [19, 31], and assume that states of the system are represented by trees and sets of states by tree automata. The transition relation of the system is represented by a set of rewriting rules. Contrary to approaches that are dedicated to specific applications, TRMC is generic and expressive enough to describe a broad class of communication protocols [4], various C programs [16] with complex data structures, multi-threaded programs [34], cryptographic protocols [26, 28, 5], and Java [13].

In TRMC, the central objective is to decide whether a set of states representing some state property belongs to the set of reachable states. An obstacle is that this set is in general neither regular nor computable in a finite time. Most existing solutions rely on computing the transitive closure of the transition relation of the systems through heuristic-based semi-algorithms [31, 4], or on the computation of some regular abstraction of the set of reachable states [19, 16]. While the first approach is precise, it is acknowledged to be ineffective on complex systems. This paper focuses on the second approach.

The first abstraction-based technique for TRMC, *Abstract Tree Regular Model Checking* (ATRMC), was proposed by Bouajjani et al [17, 15, 16]. ATRMC computes sequences of automata by successive applications of the rewriting relation to the automaton representing the initial set of states. After each computation step, techniques coming from predicate abstraction are used to over-approximate the set of reachable states. If the property holds on the abstraction, then it also holds on the concrete system. Otherwise, a counter-example is detected and the algorithm has to decide if it is a false positive or not. In case of a spurious counter-example, the algorithm refines the abstraction by backward propagation of the set of rewriting rules. The approach, which may not terminate, proceeds in a CounterExample Guided Abstraction Refinement fashion by successive abstraction/refinement until a decision can be taken. The approach has been implemented in a toolset capable, in part, to analyse C programs.

Independently, Genet et al. [24] proposed *Completion* that is another technique to compute an over-approximation of the set of reachable states. Completion exploits the structure of the term rewriting system to add new transitions in the automaton and obtain a possibly overapproximation of the set of one-step successor states. Completion leads to a direct application of rewriting rules to the automaton, while other approaches rely on possibly heavy applications of sequences of transducers to represent this step. Completion alone may not be sufficient to finitely compute the set of reachable states. A first solution to this problem is to plug one of the abstraction techniques implemented in ATRMC. However, in this paper, we prefer another solution that is to apply equational abstraction [33]. There, the merging of states is induced by a set of equations that largely exploit the structure of the system under verification and its corresponding TRS, hence leading to accurate approximations. We shall see that, initially, such equations can easily be derived from the structure of the system. Later, they are refined automatically with our procedure without manual intervention. Completion with equational abstraction has been applied to very complex case studies such as the verification of (industrial) cryptography protocols [26, 28] and Java bytecode applications [13]. CEGAR algorithms based on equational-abstraction completion exist [11, 12], but are known to be inefficient.

In this paper, we design the first efficient and certified CEGAR framework for equational-abstraction based completion algorithm. Our approach relies on $\mathcal{R}_{/E}$-automata, that is a new tree automaton formalism for representing sets of reachable states. In $\mathcal{R}_{/E}$-automata, equational abstraction does not merge states, but rather links them with rewriting rules labeled with equations. Such

technique is made easy by exploiting the nature of the completion step. During completion steps, such equations are propagated, and the information can be used to efficiently decide whether a set of terms is reachable from the set of initial states. If the procedure concludes positively, then the term is indeed reachable. Else, one has to refine the $\mathcal{R}_{/E}$-automaton and restart the process again.

Our approach has been implemented in TimbukCEGAR. (T)RMC toolsets result from the combination of several libraries, each of them being implemented with thousands of lines of code. It is thus impossible to manually prove that those tools deliver correctly answers. A particularity of TimbukCEGAR is that it is certified. In order to ensure that the whole set of reachable states has been explored, any TRMC technique needs to check whether a candidate overapproximation B is indeed a fixed point, that is if $\mathcal{L}(B) \supseteq \mathcal{R}^*(\mathcal{L}(A))$. Such check has been implemented in various TRMC toolsets, but there is no guarantee that it behaves correct. In [20], a checker for tree automata completion was designed and proved correct using the Coq [9] proof assistant. Any automaton B that passes the checker can be claimed to formally satisfy the fixed point. TimbukCEGAR implements an extension of [20] for $\mathcal{R}_{/E}$-automata, which means that the tool delivers correct answers. Our TimbukCEGAR is capable, in part, of analyzing Java programs by exploiting a elegant translation from the Java bytecode to term rewriting systems. Experiments show that TimbukCEGAR outperforms existing CEGAR-based completion algorithms by orders of magnitude.

Related work. Regular Model Checking (RMC) was first applied to compute the set of reachable states of systems whose configurations are represented by words [18, 14, 22]. The approach was then extended to trees and applied to very simple case studies [4, 19]. Other regular model checking works can be found in [2, 3], where an abstraction of the transition relation allows to exploit well-quasi ordering for finite termination. Such techniques may introduce false positives; a CEGAR approach exists for the case of finite word [1], but not for the one of trees. Learning techniques apply to RMC [38, 39] but trees have not yet been considered. We mention that our work extends equational abstractions [33, 37] with counter-example detection and refinement. We mention the existence of other automata-based works that can handle a specific class of system [34]. CEGAR principles have been implemented in various tools such as ARMC [35] or SLAM [7]. Those specific tools are more efficient than our approach. On the other hand, RMC and rewriting rules offers a more general framework in where the abstraction and the refinements can be computed in a systematic manner.

Structure of the paper. Section 2 introduces the basic definitions and concepts used in the paper. TRMC and Completion are introduced in Section 3. $\mathcal{R}_{/E}$-automata are introduced in Section 4. A new completion procedure is then defined in Section 5. Section 6 proposes a CEGAR approach for TRMC and Completion. Section 7 presents TimbukCEGAR. Section 8 concludes the paper and discusses future research. Due to space constraints proofs are reported in [10].

2 Background

In this section, we introduce some definitions and concepts that will be used throughout the rest of the paper (see also [6, 21, 30]). Let $\mathcal{F}$ be a finite set of symbols, each associated with an arity function, and let $\mathcal{X}$ be a countable set of *variables*. $\mathcal{T}(\mathcal{F}, \mathcal{X})$ denotes the set of *terms* and $\mathcal{T}(\mathcal{F})$ denotes the set of *ground terms* (terms without variables). The set of variables of a term t is denoted by $\mathcal{V}ar(t)$. A *substitution* is a function σ from $\mathcal{X}$ into $\mathcal{T}(\mathcal{F}, \mathcal{X})$, which can be uniquely extended to an endomorphism of $\mathcal{T}(\mathcal{F}, \mathcal{X})$. A *position* p for a term t is a word over $\mathbb{N}$. The empty sequence λ denotes the top-most position. The set $\mathcal{P}os(t)$ of positions of a term t is inductively defined by $\mathcal{P}os(t) = \{\lambda\}$ if $t \in \mathcal{X}$ and $\mathcal{P}os(f(t_1, \ldots, t_n)) = \{\lambda\} \cup \{i.p \mid 1 \leq i \leq n \text{ and } p \in \mathcal{P}os(t_i)\}$ otherwise. If $p \in \mathcal{P}os(t)$, then $t|_p$ denotes the subterm of t at position p and $t[s]_p$ denotes the term obtained by replacement of the subterm $t|_p$ at position p by the term s.

A *term rewriting system* (TRS) $\mathcal{R}$ is a set of *rewrite rules* $l \to r$, where $l, r \in \mathcal{T}(\mathcal{F}, \mathcal{X})$, $l \notin \mathcal{X}$, and $\mathcal{V}ar(l) \supseteq \mathcal{V}ar(r)$. A rewrite rule $l \to r$ is *left-linear* (resp. right-linear) if each variable of l (resp. r) occurs only once in l. A TRS $\mathcal{R}$ is left-linear if every rewrite rule $l \to r$ of $\mathcal{R}$ is left-linear. A TRS $\mathcal{R}$ is said to be linear iff $\mathcal{R}$ is left-linear and right-linear. The TRS $\mathcal{R}$ induces a rewriting relation $\to_{\mathcal{R}}$ on terms as follows. Let $s, t \in \mathcal{T}(\mathcal{F}, \mathcal{X})$ and $l \to r \in \mathcal{R}$, $s \to_{\mathcal{R}} t$ denotes that there exists a position $p \in \mathcal{P}os(s)$ and a substitution σ such that $s|_p = l\sigma$ and $t = s[r\sigma]_p$. The reflexive transitive closure of $\to_{\mathcal{R}}$ is denoted by $\to^*_{\mathcal{R}}$ and $s \to^!_{\mathcal{R}} t$ denotes that $s \to^*_{\mathcal{R}} t$ and t is irreducible by $\mathcal{R}$. The set of $\mathcal{R}$-descendants of a set of ground terms I is $\mathcal{R}^*(I) = \{t \in \mathcal{T}(\mathcal{F}) \mid \exists s \in I \text{ s.t. } s \to^*_{\mathcal{R}} t\}$. An *equation set* E is a set of *equations* $l = r$, where $l, r \in \mathcal{T}(\mathcal{F}, \mathcal{X})$. The relation $=_E$ is the smallest congruence such that for all substitution σ we have $l\sigma = r\sigma$. Given a TRS $\mathcal{R}$ and a set of equations E, a term $s \in \mathcal{T}(\mathcal{F})$ is rewritten modulo E into $t \in \mathcal{T}(\mathcal{F})$, denoted $s \to_{\mathcal{R}/E} t$, if there exist $s' \in \mathcal{T}(\mathcal{F})$ and $t' \in \mathcal{T}(\mathcal{F})$ such that $s =_E s' \to_{\mathcal{R}} t' =_E t$. Thus, the set of $\mathcal{R}$-descendants modulo E of a set of ground terms I is ${\mathcal{R}/E}^*(I) = \{t \in \mathcal{T}(\mathcal{F}) \mid \exists s \in I \text{ s.t. } s \to^*_{\mathcal{R}/E} t\}$.

Let Q be a finite set of symbols with arity 0, called *states*, such that $Q \cap \mathcal{F} = \emptyset$. $\mathcal{T}(\mathcal{F} \cup Q)$ is called the set of *configurations*. A *transition* is a rewrite rule $c \to q$, where c is a configuration and q is state. A transition is *normalized* when $c = f(q_1, \ldots, q_n)$, $f \in \mathcal{F}$ is of arity n, and $q_1, \ldots, q_n \in Q$. A *ε-transition* is a transition of the form $q \to q'$ where q and q' are states. A bottom-up nondeterministic finite tree automaton (tree automaton for short) over the alphabet $\mathcal{F}$ is a tuple $A = \langle \mathcal{F}, Q, Q_F, \Delta \rangle$, where $Q_F \subseteq Q$, Δ is a set of normalized transitions and ε-transitions. The transitive and reflexive *rewriting relation* on $\mathcal{T}(\mathcal{F} \cup Q)$ induced by all the transitions of A is denoted by $\to^*_A$. The tree language recognized by A in a state q is $\mathcal{L}(A, q) = \{t \in \mathcal{T}(\mathcal{F}) \mid t \to^*_A q\}$. We define $\mathcal{L}(A) = \bigcup_{q \in Q_F} \mathcal{L}(A, q)$.

3 Tree Regular Model Checking with Completion

We first introduce *Tree Regular Model Checking* (TRMC), a tree automata based framework to represent possibly infinite-state systems. In TRMC, a program is

represented by a tuple $(\mathcal{F}, A, \mathcal{R})$, where $\mathcal{F}$ is an alphabet on which a set of terms $\mathcal{T}(\mathcal{F})$ can be defined; A is the tree automaton representing a possibly infinite set of configurations I, and $\mathcal{R}$ is a set of term rewriting rules that represent a transition relation *Rel*. We consider the following problem.

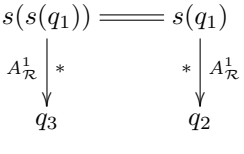
Definition 1 (Reachability Problem (RP)). *Consider a program $(\mathcal{F}, A, \mathcal{R})$ and a set of bad terms Bad. The Reachability Problem consists in checking whether there exists a term of $\mathcal{R}^*(\mathcal{L}(A))$ that belongs to Bad.*

For finite-state systems, computing the set of reachable terms $(\mathcal{R}^*(\mathcal{L}(A)))$ reduces to enumerating the terms that can be reached from the initial set of configurations. For infinite-state systems, acceleration-based methods are needed to perform this possibly infinite enumeration in a finite time. In general, such accelerations are not precise and the best one can obtain is an R-closed approximation $A^*_{\mathcal{R}}$. A tree automaton $A^*_{\mathcal{R}}$ is $\mathcal{R}$-closed if for all terms $s, t \in \mathcal{T}(\mathcal{F})$ such that $s \to_{\mathcal{R}} t$ and s is recognized by $A^*_{\mathcal{R}}$ into state q then so is t. It is easy to see that if $A^*_{\mathcal{R}}$ is $\mathcal{R}$-closed and $\mathcal{L}(A^*_{\mathcal{R}}) \supseteq \mathcal{L}(A)$, then $\mathcal{L}(A^*_{\mathcal{R}}) \supseteq \mathcal{R}^*(\mathcal{L}(A))$. A wide range of acceleration techniques have been developed, most of them have been discussed in Section 1. Here, we focus on Completion [24], whose objective is to computes successive automata $A^0_{\mathcal{R}} = A, A^1_{\mathcal{R}}, A^2_{\mathcal{R}}, \ldots$ that represent the effect of applying the set of rewriting rules to the initial automaton. To compute infinite sets in a finite time, each completion step is followed by a widening operator. More precisely, each application of $\mathcal{R}$, which is called a *completion step*, consists in searching for *critical pairs* $\langle t, q\rangle$ with $s \to_{\mathcal{R}} t$, $s \to^*_A q$ and $t \not\to^*_A q$. The idea being that the algorithm solves the critical pair by building from $A^i_{\mathcal{R}}$, a new tree automaton $A^{i+1}_{\mathcal{R}}$ with the additional transitions that represent the effect of applying $\mathcal{R}$. As the language recognized by A may be infinite, it is not possible to find all the critical pairs by enumerating the terms that it recognizes. The solution that was promoted in [24] consists in applying sets of substitutions $\sigma : \mathcal{X} \mapsto Q$ mapping variables of rewrite rules to states that represent infinite sets of (recognized) terms. Given a tree automaton $A^i_{\mathcal{R}}$ and a rewrite rule $l \to r \in \mathcal{R}$, to find all the critical pairs of $l \to r$ on $A^i_{\mathcal{R}}$, completion uses a *matching algorithm* [23] that produces the set of substitutions $\sigma : \mathcal{X} \mapsto Q$ and states $q \in Q$ such that $l\sigma \to^*_{A^i_{\mathcal{R}}} q$ and $r\sigma \not\to^*_{A^i_{\mathcal{R}}} q$. Solving critical pairs thus consists in adding new transitions: $r\sigma \to q'$ and $q' \to q$. Those new transitions may have to be *normalized* in order to satisfy the definition of transitions of tree automata (see [23] for details). As it was shown in [24], this operation may add not only new transitions but also new states to the automaton. In the rest of the paper, the completion-step operation will be represented by $\mathtt{C}$, i.e., the automaton obtained by applying the completion step to $A^i_{\mathcal{R}}$ is denoted $\mathtt{C}(A^i_{\mathcal{R}})$. Observe that when considering right-linear rewriting rules, we have that $\mathtt{C}$ is precise, i.e. it does not introduce in $A^{i+1}_{\mathcal{R}}$ terms that cannot be obtain from $A^i_{\mathcal{R}}$ by applying the set of rewriting rules. Observe also that if the system is non left-linear, then completion step may not produce all the reachable terms. Non left-linear rules will not be considered in the present paper.

The problem is that, except for specific classes of systems [23, 25], the automaton representing the set of reachable terms cannot be obtained by applying

a finite number of completion steps. The computation process thus needs to be accelerated. For doing so, we apply a *widening operator* $\mathtt{W}$ that uses a set E of equations[5] to merge states and produce an $\mathcal{R}$-closed automaton that is an over-approximation of the set of reachable terms, i.e., an automaton $A^*_{\mathcal{R},E}$ such that $\mathcal{L}(A^*_{\mathcal{R},E}) \supseteq \mathcal{R}^*(\mathcal{L}(A))$. An equation $u = v$ is applied to a tree automaton A as follows: for all substitution $\sigma : \mathcal{X} \mapsto Q$ and distinct states q_1 and q_2 such that $u\sigma \to^*_A q_1$ and $v\sigma \to^*_A q_2$, states q_1 and q_2 are merged. Completion and widening steps are applied, i.e., $A^{i+1}_{\mathcal{R},E} = \mathtt{W}(\mathtt{C}(A^i_{\mathcal{R},E}))$, until a $\mathcal{R}$-closed fixpoint $A^*_{\mathcal{R},E}$ is found. Our approximation framework and methodology are close to the equational abstractions of [33]. In [27], it has been shown that, under some assumptions, the widening operator may be exact, i.e., does not add terms that are not reachable.

Example 1. Let $\mathcal{R} = \{f(x) \to f(s(s(x)))\}$ be a rewriting system, $E = \{s(s(x)) = s(x)\}$ be an equation, and $A = \langle \mathcal{F}, Q, Q_F, \Delta \rangle$ be a tree automaton with $Q_F = \{q_0\}$ and $\Delta = \{a \to q_1, f(q_1) \to q_0\}$, i.e. $\mathcal{L}(A) = \{f(a)\}$. The first completion step finds the following critical pair: $f(q_1) \to^*_A q_0$ and $f(s(s(q_1))) \not\to^*_A q_0$. Hence, the completion algorithm produces $A^1_{\mathcal{R}} = \mathtt{C}(A)$ having all transitions of A plus $\{s(q_1) \to q_2, s(q_2) \to q_3, f(q_3) \to q_4, q_4 \to q_0\}$ where q_2, q_3, q_4 are new states produced by normalization of $f(s(s(q_1))) \to q_0$. Applying $\mathtt{W}$ with the equation $s(s(x)) = s(x)$ on $A^1_{\mathcal{R}}$ is equivalent to rename q_3 into q_2. The set of transitions of $A^1_{\mathcal{R},E}$ is thus $\Delta \cup \{s(q_1) \to q_2, s(q_2) \to q_2, f(q_2) \to q_4, q_4 \to q_0\}$. Completion stops on $A^1_{\mathcal{R},E}$ that is $\mathcal{R}$-closed, and thus $A^*_{\mathcal{R},E} = A^1_{\mathcal{R},E}$.



Observe that if the intersection between $A^*_{\mathcal{R},E}$ and Bad is not empty, then it does not necessarily mean that the system does not satisfy the property. Consider a set $Bad = \{f(s(a)), f(s(s(a)))\}$, the first term of this set is not reachable from A, but the second is. There is thus the need to successively refine the $\mathcal{R}$-closed automaton. The latter can be done by using a CounterExample Guided Abstraction Refinement algorithm (CEGAR). Developing such an algorithm for completion and equational abstraction is the objective of this paper.

4 $\mathcal{R}_{/E}$-Automata

Existing CEGAR approaches [17, 15, 16, 11] check for spurious counter examples by performing a sequence of applications of the rewriting rules to $A^*_{\mathcal{R},E}$. To avoid this potentially costly step, we suggest to replace the merging of states by the addition of new rewriting rules giving information on the merging through equations. Formally:

Definition 2 ($\mathcal{R}_{/E}$-automaton). *Given a TRS $\mathcal{R}$ and a set E of equations, an $\mathcal{R}_{/E}$-automaton A is a tuple $\langle \mathcal{F}, Q, Q_F, \Delta \cup \varepsilon_{\mathcal{R}} \cup \varepsilon_E \rangle$. Δ is a set of normalized*

[5] Those equations have to be provided by the user. In many cases, they can be produced when formalizing the problem in the TRMC framework [37]. The situation is similar for the predicates used in [17, 15, 16].

transitions. ε_E *is a set of* ε*-transitions.* $\varepsilon_{\mathcal{R}}$ *is a set of* ε*-transitions labeled by* $\top$ *or conjunctions over predicates of the form* $Eq(q, q')$ *where* $q, q' \in Q$*, and* $q \to q' \in \varepsilon_E$*.*

Set $\varepsilon_{\mathcal{R}}$ is used to distinguish a term from its successors that has been obtained by applying one or several rewriting rules. Instead of merging states according to the set of equations, A links them with epsilon transitions in ε_E. During the completion step, when exploiting critical pairs, the combination of transitions in ε_E generates transition in $\varepsilon_{\mathcal{R}}$ that are labeled with a conjunction of equations representing those transitions in ε_E. In what follows, we use $\to^*_{\Delta}$ to denote the transitive and reflexive closure of Δ. Given a set Δ of normalized transitions, the set of representatives of a state q is defined by $Rep(q) = \{t \in \mathcal{T}(\mathcal{F}) | t \to^*_{\Delta} q\}$.

Definition 3 (Run of a $\mathcal{R}_{/E}$-automaton A).

- $t|_p = f(q_1, \ldots, q_n)$ *and* $f(q_1, \ldots, q_n) \to q \in \Delta$ *then* $t \xrightarrow{\top}_A t[q]_p$
- $t|_p = q$ *and* $q \to q' \in \varepsilon_E$ *then* $t \xrightarrow{Eq(q,q')}_A t[q']_p$
- $t|_p = q$ *and* $q \xrightarrow{\alpha} q' \in \varepsilon_{\mathcal{R}}$ *then* $t \xrightarrow{\alpha}_A t[q']_p$
- $u \xrightarrow{\alpha}_A v$ *and* $v \xrightarrow{\alpha'}_A w$ *then* $u \xrightarrow{\alpha \wedge \alpha'}_A w$

Theorem 1. $\forall t \in \mathcal{T}(\mathcal{F} \cup Q),\ q \in Q,\ t \xrightarrow{\alpha}_A q \iff t \to^*_A q$

A run $\xrightarrow{\alpha}$ abstracts a rewriting path of $\to_{\mathcal{R}/E}$. If $t \xrightarrow{\alpha} q$, then there exists a term $s \in Rep(q)$ such that $s \to^*_{\mathcal{R}/E} t$. The formula α denotes the subset of transitions of ε_E needed to recognize t into q.

Example 2. Let $I = f(a)$ be an initial set of terms, $\mathcal{R} = \{f(c) \to g(c),\ a \to b\}$ be a set of rewriting rules, and $E = \{b = c\}$ be a set of equations. We build A an overapproximation automaton for $\mathcal{R}^*(I)$, using E.

Thanks to ε-transitions, the automaton A represented in Fig. 1 contains some information about the path used to reach terms using $\mathcal{R}$ and E. Each state has a representative term from which others are obtained. The equality $b = c$ is represented by the two transitions $q_c \to q_b$ and $q_b \to q_c$ of ε_E, taking into account that b and c are the representatives terms for states q_b and q_c, respectively. Consider now State q_c, Transition $q_b \to q_c$ indicates that the term b is obtained from Term c by using the equality. Conversely, Transition $q_c \to q_b$ leads to the conclusion that Term c is obtained from Term b.

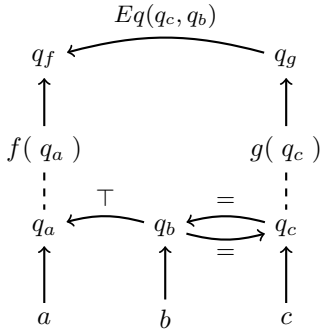


Fig. 1: Automaton A

The transition $q_b \to q_a$ denotes that the term b is a descendant of a by rewriting. Using Definition 3, the runs $f(c) \xrightarrow{Eq(q_c,q_b)} q_f$ indicates that to obtain $f(c)$ from $f(a)$ – the representative term of q_f – we used the equality $b = c$, which is obtained from $q_c \to q_b$. We indeed observe $f(a) \to_{\mathcal{R}} f(b) =_E f(c)$. If we now consider the transition $q_g \to q_f$ we labeled the transition with the formula $Eq(q_c, q_b)$. To reach $g(c)$ from $f(a)$, we rewrite $f(c)$. We have seen this term is reachable thanks to the equivalence relation induced by $b = c$. By transitivity, this equivalence is also used to reach the term $g(c)$. We thus label the transition of $\varepsilon_{\mathcal{R}}$ to save this information. We obtain the run $g(c) \xrightarrow{Eq(q_c,q_b)} q_f$. We observe that the transition $q_b \to q_a$ is labeled by the formula $\top$ since b is reachable from a without any equivalence. By congruence, so is $f(b)$ from $f(a)$. The run $f(b) \xrightarrow{\top} q_f$ denotes it.

We now introduce a property that will be used in the refinement procedure to distinguish between counter-examples and false positives.

Definition 4 (A well-defined $\mathcal{R}_{/E}$-automaton). *A is a* well-defined $\mathcal{R}_{/E}$*-automaton, if :*

- *For all states q of A, and all terms v such that $v \xrightarrow{\top}_A q$, there exists u a term representative of q such that $u \to^*_{\mathcal{R}} v$*
- *If $q \xrightarrow{\phi} q'$ is a transition of $\varepsilon_{\mathcal{R}}$, then there exist terms $s, t \in \mathcal{T}(\mathcal{F})$ such that $s \xrightarrow{\phi}_A q$, $t \xrightarrow{\top}_A q'$ and $t \to_{\mathcal{R}} s$.*

The first item in Definition 4 guarantees that every term recognized by using transitions labeled with the formula $\top$ is indeed reachable from the initial set. The second item is used to refine the automaton. A rewriting step of $\to_{\mathcal{R}/E}$ denoted by $q \xrightarrow{\phi} q'$ holds thanks to some transitions of ε_E that occurs in ϕ. If we remove transitions in ε_E in such a way that ϕ does not hold, then the transition $q \xrightarrow{\phi} q'$ should also be removed.

According to the above construction, a term t that is recognized by using at least a transition labeled with a formula different from $\top$ can be removed from the language of the $\mathcal{R}_{/E}$-automaton by removing some transitions in ε_E. This "pruning" operation will be detailed in Section 6.

5 Solving the Reachability Problem with $\mathcal{R}_{/E}$-automaton

In this section, we extend the completion and widening principles introduced in Section 3 to take advantage of the structure of $\mathcal{R}_{/E}-$automata. We consider an initial set I that can be represented by a tree automaton $A^0_{\mathcal{R},E} = \langle \mathcal{F}, Q^0, Q_F, \Delta^0 \rangle$, and transition relation represented by a set of linear rewriting rules $\mathcal{R}$. In the next section, we will see that the right-linearity condition may be relaxed using additionnal hypotheses. We compute successive approximations $A^i_{\mathcal{R},E} = \langle \mathcal{F}, Q^i, Q_f, \Delta^i \cup \varepsilon^i_{\mathcal{R}} \cup \varepsilon^i_E \rangle$ from $A^0_{\mathcal{R},E}$ using $A^{i+1}_{\mathcal{R},E} = \mathtt{W}(\mathtt{C}(A^i_{\mathcal{R},E}))$. Observe that $A^0_{\mathcal{R},E}$ is well-defined as the sets $\varepsilon^0_{\mathcal{R}}$ and ε^0_E are empty.

5.1 The Completion step C

Extending completion to $\mathcal{R}_{/E}$-automaton requires to modify the concept of critical pair and so the algorithm to compute them. A critical pair for a $\mathcal{R}_{/E}$-automaton is a triple $\langle r\sigma, \alpha, q\rangle$ such that $l\sigma \to r\sigma$, $l\sigma \xrightarrow{\alpha}_{A^i_{\mathcal{R},E}} q$ and there is no formula α' such that $r\sigma \xrightarrow{\alpha'}_{A^i_{\mathcal{R},E}} q$. The resolution of such a critical pair consists of adding to $\mathtt{C}(A^i_{\mathcal{R},E})$ the transitions to obtain $r\sigma \xrightarrow{\alpha}_{\mathtt{C}(A^i_{\mathcal{R},E})} q$. This is followed by a normalization step $\mathtt{Norm}$ whose definition is similar to the one for classical tree automata.

Definition 5 (Normalization). *The normalization is done in two mutually inductive steps parametrized by the configuration c to recognize, and by the set of transitions Δ to extend. Let Q^{Δ}_{new} be a set of (new) states not occurring in Δ.*

$$\begin{cases} \mathtt{Norm}(c,\Delta) = \mathtt{Slice}(d,\Delta), & \text{for one } d \text{ s.t. } c \to^*_{\Delta} d, \text{ with } c,\ d \in \mathcal{T}(\mathcal{F}\cup Q) \\ \mathtt{Slice}(q,\Delta) = \Delta, & q \in Q \\ \mathtt{Slice}(f(q_1,\ldots,q_n),\Delta) = \Delta \cup \{f(q_1,\ldots,q_n) \to q\}, & q_i \in Q \text{ and one } q \in Q^{\Delta}_{new} \\ \mathtt{Slice}(f(t_1,\ldots,t_n),\Delta) = \mathtt{Norm}(f(t_1,\ldots,t_n),\mathtt{Slice}(t_i,\Delta)), & \exists t_i \in \mathcal{T}(\mathcal{F}\cup Q)\setminus Q \end{cases}$$

Definition 6 (Resolution of a critical pair). *Given a $\mathcal{R}_{/E}$-automaton $A = \langle \mathcal{F}, Q, Q_f, \Delta \cup \varepsilon_{\mathcal{R}} \cup \varepsilon_E\rangle$ and a critical pair $p = \langle r\sigma, \alpha, q\rangle$, the* resolution *of p on A is the $\mathcal{R}_{/E}$-automaton $A' = \langle \mathcal{F}, Q', Q_f, \Delta' \cup \varepsilon'_{\mathcal{R}} \cup \varepsilon_E\rangle$ where*

- $\Delta' = \Delta \cup \mathtt{Norm}(r\sigma, \Delta \setminus \Delta^0)$*;*
- $\varepsilon'_{\mathcal{R}} = \varepsilon_{\mathcal{R}} \cup \{q' \xrightarrow{\alpha} q\}$ *where q' is the state such that $r\sigma \to_{\Delta'\setminus\Delta_0} q'$;*
- Q' *is the union of Q with the set of states added when creating Δ'.*

Note that Δ_0, the set of transitions of $A^0_{\mathcal{R}}$, is not used in the normalization process. This is to guarantee that A' is well-defined. The $\mathcal{R}_{/E}$-automaton $\mathtt{C}(A^i_{\mathcal{R},E})$ is obtained by recursively applying the above resolution principle to all critical pairs p of the set of critical pairs between $\mathcal{R}$ and $A^i_{\mathcal{R},E}$.

The set of all critical pairs is obtained by solving the *matching problems* $l \trianglelefteq q$ for all rewrite rules $l \to r \in \mathcal{R}$ and all states $q \in A^i_{\mathcal{R},E}$. Solving $l \trianglelefteq q$ is performed in two steps. First, one computes S, that is the set of all pairs (α, σ) such that α is a formula, σ is a substitution of $\mathcal{X} \mapsto Q^i$, and $l\sigma \xrightarrow{\alpha} q$. The formula α is a conjunction of Predicates $Eq(q', q'')$ that denotes the used transitions of ε_E to rewrite $l\sigma$ in q, in accordance with Definition 3. Due to space constraints the algorithm, which always terminates, can be found in [10].

Second, after having computed S for $l \trianglelefteq q$, we identify elements of the set that correspond to critical pairs. By definition of S, we know that there exists a transition $l\sigma \xrightarrow{\alpha}_{A^i_{\mathcal{R},E}} q$ for $(\alpha, \sigma) \in S$. If there exists a transition $r\sigma \xrightarrow{\alpha'}_{A^i_{\mathcal{R},E}} q$, then $r\sigma$ has already been added to $A^i_{\mathcal{R},E}$. If there does not exist a transition of the form $r\sigma \xrightarrow{\alpha'}_{A^i_{\mathcal{R},E}} q$, then $\langle r\sigma, \alpha', q\rangle$ is a critical pair to solve on $A^i_{\mathcal{R},E}$. The following theorem shows that our methodology is complete.

Theorem 2. *If $A^i_{\mathcal{R},E}$ is well-defined then so is $\mathtt{C}(A^i_{\mathcal{R},E})$, and $\forall q \in Q^i$, $\forall t \in \mathcal{L}(A^i_{\mathcal{R},E}, q)$, $\forall t' \in \mathcal{T}(\mathcal{F})$, $t \to_{\mathcal{R}} t' \implies t' \in \mathcal{L}(\mathtt{C}(A^i_{\mathcal{R},E}), q)$.*

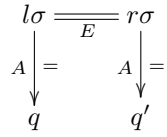
Example 3. Let $\mathcal{R} = \{f(x) \to f(s(s(x)))\}$ be a set of rewriting rules and $A^0_{\mathcal{R},E} = \langle \mathcal{F}, Q, Q_F, \Delta^0 \rangle$ be a tree automaton such that $Q_F = \{q_0\}$ and $\Delta^0 = \{a \to q_1, f(q_1) \to q_0\}$. The solution of the matching problem $f(x) \trianglelefteq q_0$ is $S = \{(\sigma, \phi)\}$, with $\sigma = \{x \to q_1\}$ and $\phi = \top$. Hence, since $f(s(s(q_1))) \not\xrightarrow{\top}_{A^0_{\mathcal{R},E}} q_0$, $\langle f(s(s(q_1))), \top, q_0 \rangle$ is the only critical pair to be solved. So, we have $\mathtt{C}(A^0_{\mathcal{R},E}) = \langle \mathcal{F}, Q^1, Q_F, \Delta^1 \cup \varepsilon^1_{\mathcal{R}} \cup \varepsilon^0_E \rangle$, with:

$$\Delta^1 = \mathtt{Norm}(f(s(s(q_1))), \emptyset) \cup \Delta^0 = \{s(q_1) \to q_2, s(q_2) \to q_3, f(q_3) \to q_4\} \cup \Delta^0,$$
$$\varepsilon^1_{\mathcal{R}} = \{q_4 \xrightarrow{\top} q_0\}, \text{since } f(s(s(q_1))) \to_{\Delta^1 \setminus \Delta^0} q_4, \varepsilon^0_E = \emptyset \text{ and } Q^1 = \{q_0, q_1, q_2, q_3, q_4\}.$$

Observe that if $\mathtt{C}(A^i_{\mathcal{R},E}) = A^i_{\mathcal{R},E}$, then we have reached a fixpoint.

5.2 The Widening Step W

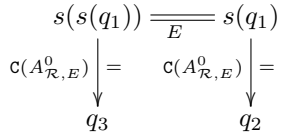
Consider a $\mathcal{R}_{/E}$-automaton $A = \langle \mathcal{F}, Q, Q_f, \Delta \cup \varepsilon_{\mathcal{R}} \cup \varepsilon_E \rangle$, the widening consists in computing a $\mathcal{R}_{/E}$-automaton $\mathtt{W}(A)$ that is obtained from A by using E. For each equation $l = r$ in E, we consider all pair (q, q') of distinct states of Q^i such that there exists a substitution σ to obtain the following diagram. Observe that $\xrightarrow{=}_A$, the transitive and reflexive rewriting relation induced by $\Delta \cup \varepsilon_E$, defines particular runs which exclude transitions of $\varepsilon_{\mathcal{R}}$. This allows us to build a more accurate approximation. The improvement in accurary is detailed in [27].



Intuitively, if we have $u \xrightarrow{=}_A q$, then we know that there exists a term t of $Rep(q)$ such that $t =_E u$. The automaton $\mathtt{W}(A)$ is given by the tuple $\langle \mathcal{F}, Q, Q_f, \Delta \cup \varepsilon_{\mathcal{R}} \cup \varepsilon'_E \rangle$, where ε'_E is obtained by adding the transitions $q \to q'$ and $q' \to q$ to ε_E (for each pair (q, q')).

Theorem 3. *Assuming that A is well-defined, we have A syntactically included in $\mathtt{W}(A)$, and $\mathtt{W}(A)$ is well-defined.*

Example 4. Consider the $\mathcal{R}_{/E}$-automaton $\mathtt{C}(A^0_{\mathcal{R},E})$ given in Example 3. Using Equation $s(s(x)) = s(x)$, we compute $A^1_{\mathcal{R},E} = \mathtt{W}(\mathtt{C}(A^0_{\mathcal{R},E}))$. We have $\sigma = \{x \mapsto q_1\}$ and the following diagram. We then obtain $A^1_{\mathcal{R},E} = \langle \mathcal{F}, Q^1, Q_f, \Delta^1 \cup \varepsilon^1_{\mathcal{R}} \cup \varepsilon^1_E \rangle$, where $\varepsilon^1_E = \varepsilon^0_E \cup \{q_3 \to q_2,\ q_2 \to q_3\}$ and $\varepsilon^0_E = \emptyset$. Observe that $A^1_{\mathcal{R},E}$ is a fixpoint, i.e., $\mathtt{C}(A^1_{\mathcal{R},E}) = A^1_{\mathcal{R},E}$.



6 A CEGAR procedure for $\mathcal{R}_{/E}$-automata

Let $\mathcal{R}$ be a TRS, I be a set of initial terms characterized by the $\mathcal{R}_{/E}-$automaton $A^0_{\mathcal{R},E}$ and Bad the set of forbidden terms represented by A_{Bad}. We now complete our CEGAR approach by proposing a technique that checks whether a term is

indeed reachable from the initial set of terms. If the term is a spurious counter-example i.e. a counter-example of the approximation, then it has to be removed from the approximation automatically, else one can deduce that the involved term is actually reachable.

Let $A^k_{\mathcal{R},E} = \langle \mathcal{F}, Q^k, Q_f, \Delta^k \cup \varepsilon^k_{\mathcal{R}} \cup \varepsilon^k_E \rangle$ be a $\mathcal{R}_{/E}$-automaton obtained after k steps of completion and widening from $A^0_{\mathcal{R},E}$ and assume that $\mathcal{L}(A^k_{\mathcal{R},E}) \cap Bad \neq \emptyset$. Let $S_{A^k_{\mathcal{R},E} \cap A_{Bad}}$ be a set of triples $\langle q, q', \phi \rangle$ where q is a final state of $A^k_{\mathcal{R},E}$, q' is a final state of A_{Bad} and ϕ is a formula on transitions of ε^k_E and such that for each triple (q, q', ϕ), the formula ϕ holds if and only if there exists $t \in \mathcal{L}(A^k_{\mathcal{R},E}, q) \cap \mathcal{L}(A_{Bad}, q')$ and $t \xrightarrow{\phi}_{A^k_{\mathcal{R},E}} q$. Note that $S_{A^k_{\mathcal{R},E} \cap A_{Bad}}$ can be obtained using an intersection based algorithm defined in [10]. We consider two cases. First, as $A^k_{\mathcal{R},E}$ is well-defined, if $\phi = \top$, we deduce that t is indeed a reachable term. Otherwise, ϕ is a formula whose atoms are of the form $Eq(q_j, q'_j)$, and t is possibly a spurious counter-example, and the run $t \xrightarrow{\phi}_{A^k_{\mathcal{R},E}} q$ must be removed. Refinement consists in computing a pruned version $\mathtt{P}(A^k_{\mathcal{R},E}, S_{A^k_{\mathcal{R},E} \cap A_{Bad}})$ of $A^k_{\mathcal{R},E}$.

Definition 7. *Given an $\mathcal{R}_{/E}-$automaton $A = \langle \mathcal{F}, Q, Q_F, \Delta_0 \cup \Delta \cup \varepsilon_{\mathcal{R}} \cup \varepsilon_E \rangle$ and a set of terms specified by the automaton A_{Bad}, the prune process is defined by*

$$\mathtt{P}(A, S_{A \cap A_{Bad}}) = \begin{cases} \mathtt{P}(A', S_{A' \cap A_{Bad}}) & \text{if } S_{A' \cap A_{Bad}} \neq \emptyset \text{ and with} \\ & A' = \mathtt{Clean}(A, S_{A \cap A_{Bad}}) \\ A & \text{if } S_{A \cap A_{Bad}} = \emptyset \text{ or there exists } t \in Bad \\ & \text{s.t. } t \xrightarrow{\top}_A q_f \text{ and } q_f \in Q_F. \end{cases}$$

where $\mathtt{Clean}(A, S_{A \cap A_{Bad}})$, consists of removing transitions of ε_E until for each $\langle q_f, q'_f, \phi \rangle \in S_{A \cap A_{Bad}}$, ϕ does not hold, i.e., $\phi = \bot$ with q_f, q'_f respectively two final states of A and A_{Bad}.

To replace Predicate $Eq(q, q')$ by $\bot$ in ϕ, we have to remove the transition $q \to q'$ from ε_E. In addition, we also have to remove all transitions $q \xrightarrow{\alpha} q' \in \varepsilon_{\mathcal{R}}$, where the conjunction α contains some predicates $Eq(q_1, q_2)$ whose transition $q_1 \to q_2$ has been removed from ε_E. In general, removing Transition $q \to q'$ may be too rough. Indeed, assuming that there also exists a transition $q'' \to q$ of ε_E, removing the transition $q \to q'$ also avoids the induced reduction $q'' \to q'$ from the automaton and then, unconcerned terms of q'' are also removed. To save those terms, Transition $q'' \to q'$ is added to ε_E, but only if it has never been removed by a pruning step. This point is important to refine the automaton with accuracy. The prune step is called recursively as inferred transitions may keep the intersection non-empty.

Theorem 4. *Let $t \in Bad$ be a spurious counter-example. The pruning process always terminates, and removes all the runs of the form $t \xrightarrow{\phi} q$.*

Example 5. We consider the $\mathcal{R}_{/E}$-automaton A of Example 2. It is easy to see that A recognizes the term $g(c)$. Indeed, by Definition 3, we have $g(c) \xrightarrow{Eq(q_c,q_b)} q_f$. Consider now the rewriting path $f(a) \rightarrow_{\mathcal{R}} f(b) =_E f(c) \rightarrow_{\mathcal{R}} g(c)$. If we remove the step $f(b) =_E f(c)$ denoted by the transition $q_c \rightarrow q_b$, then $g(c)$ becomes unreachable and should also be removed. The first step in pruning A consists thus in removing this transition. In a second step, we propagate the information by removing all transition of $\varepsilon_{\mathcal{R}}$ labeled by a formula that contains $Eq(q_c, q_b)$. This is done to remove all terms obtained by rewriting with the equivalence $b =_E c$. After having pruned all the transitions, we observe that the terms recognized by A are given by the set $\{f(a), f(b)\}$.

Let us now characterize the soundness and completness of our approach.

Theorem 5 (Soundness on left-linear TRS). *Consider a left-linear TRS $\mathcal{R}$, a set of terms Bad, a set of equations E and a well-defined $\mathcal{R}_{/E}-$automaton A_0. Let $A^*_{\mathcal{R},E}$ be a fixpoint $\mathcal{R}_{/E}$-automaton of $\mathtt{P}(A', S_{A' \cap A_{Bad}})$ and $A' = \mathtt{W}(\mathtt{C}(A_i))$ for $i \geq 0$. If $\mathcal{L}(A^*_{\mathcal{R},E}) \cap Bad = \emptyset$, then $Bad \cap \mathcal{R}^*(\mathcal{L}(A_0)) = \emptyset$.*

Theorem 6 (Completeness on Linear TRS). *Given a linear TRS $\mathcal{R}$, a set of terms Bad defined by automata A_{Bad}, a set of equations E and a well-defined $\mathcal{R}_{/E}-$automaton A_0. For any $i > 0$, let A_i be the $\mathcal{R}_{/E}-$automaton obtained from A_{i-1} in such a way: $A_i = \mathtt{P}(A', S_{A' \cap A_{Bad}})$ and $A' = \mathtt{W}(\mathtt{C}(A_{i-1}))$. If $Bad \cap \mathcal{R}^*(\mathcal{L}(A_0)) \neq \emptyset$ then there exists $t \in Bad$ and $j > 0$ such that $t \xrightarrow{\top}_{A_j} q_f$ and q_f is a final state of A_j.*

This result also extends to left-linear TRS with a finite set of initial terms (cardinality of $Rep(q)$ is 1 for all state q of A_0).

Theorem 7 (Completeness on Left-Linear TRS). *Theorem 6 extends to left-linear TRS if for any state q of A_0, the cardinality of $Rep(q)$ is 1.*

7 Implementation, Application and Certification

Our approach has been implemented in TimbukCEGAR that is an extension of the Timbuk 3.1 toolset [29]. Timbuk is a well-acknowledged tree automata library that implements several variants of the completion approach. TimbukCEGAR consists of around 11000 lines of OCaml, 75% of which are common with Timbuk 3.1. TimbukCEGAR exploits a BDD-based representation of equation formulas through the Buddy BDD library [32].

A particularity of TimbukCEGAR is that it is certified. At the heart of any abstraction algorithm there is the need to check whether a candidate over-approximation B is indeed a fixed point, that is if $\mathcal{L}(B) \supseteq \mathcal{R}^*(\mathcal{L}(A))$. Such check has been implemented in various TRMC toolsets, but there is no guarantee that it behaves correctly, i.e., that the TRMC toolset gives a correct answer. In [20], a checker for tree automata completion was designed and proved correct using the Coq [9] proof assistant. As such, any TRMC toolset that produces an automaton B that passes the checker can be claimed to work properly. TimbukCEGAR

implements a straightforward extension of [20] for $\mathcal{R}_{/E}$-automata, which means that the tool delivers provably correct answers. In what follows, we describe how Java programs can be analyzed using our approach. Both Timbuk and TimbukCEGAR are available at `http://www.irisa.fr/celtique/genet/timbuk/`.

In a french initiative called RAVAJ [36], we have defined a generic certified verification chain based on TRMC. This chain is composed of three main links. The two first links rely on an encoding of the operational semantics of the programming language as a term rewriting system and a set of rewrite rules. The third link is a TRMC toolset, here TimbukCEGAR. With regards to classical static analysis, the objective is to use TRMC and particularly tree automata completion as a foundation mechanism for ensuring, by construction, safety of static analyzers. For Java, using approximation rules instead of abstract domains makes the analysis easier to fine-tune. Moreover, our approach relies on a checker that certifies the answer to be correct.

We now give more details and report some experimental results. We used Copster [8], to compile a Java `.class` file into a TRS. The obtained TRS models exactly a subset of the semantics[6] of the Java Virtual Machine (JVM) by rewriting a term representing the state of the JVM [13]. States are of the form `IO(st,in,out)` where `st` is a program state, `in` is an input stream and `out` an output stream. A program state is a term of the form `state(f,fs,h,k)` where `f` is 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`. We consider the following program:

```
class List{
    List next;
    int val;
    public List(int elt, List l){
        next= l;
        if (elt<0) val= -elt;
        else val= elt;
    }
    public void printSum(){
        List l= this;
        int sum= 0;
        while (l != null){
            sum= sum+l.val;
            l= l.next;
        }
        System.out.println(sum);
    }}
```

```
class TestList{
    public static void main(String[] argv){
        List ls= null;
        int x= 0;
        while (x!=-1) {
            try {x= System.in.read();}
            catch(java.io.IOException e){};
            ls= new List(x,ls);
        }
        ls.printSum();
    }
}
```

Let us now check that the sum output by the program can never be equal to zero, for all non-empty input stream of integers. The TRS generated by Copster has 879 rules encoding both the JVM semantics and the bytecode of the above Java program. Note that, this example is bigger than those generally used by other TRMC techniques. The complete TRS is available with TimbukCEGAR distribution. Initial terms are of the form `IO(s,lin,nilout)` where `s` is the initial JVM state, `lin` is a non-empty unbounded list of integers and `nilout` is

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

the empty list of outputs. Starting from this initial set of terms, completion is likely to diverge without approximations. Indeed, the program is going to allocate infinitely many objects of class `List` in the heap and, furthermore, compute an unbounded sum in the method `printSum`. In the heap, there is one separate heap for each class. Each heap consists of a list of objects. For instance, in the heap for class `List`, objects are stored using a list constructor `stackHeapList(x,y)`. Thus, to enforce termination we can approximate the heap for objects of class `List` using the following equation `stackHeapList(x,y)=y`. The effect of this equation is to collapse all the possible lists built using `stackHeapList`, hence all the possible heaps for class `List`. The other equations are `succ(x)=x` and `pred(x)=x` for approximating infinitely growing or decreasing integers.

By using those equations, TimbukCEGAR finds a counterexample. This is due to the fact that, amongst all considered input streams, an input stream consisting of a list of 0 results into a 0 sum. The solution is to restrict the initial language to non-empty non-zero integer streams. However, refinement of equations is needed since `succ(x)=x` and `pred(x)=x` put 0 and all the other integers in the same equivalence class. Refining those equations by hand is hard, *e.g.* using equations `succ(succ(x))=succ(x)` and `pred(pred(x))=pred(x)` is not enough to eliminate spurious counterexamples. After 334 completion steps and 4 refinement steps, TimbukCEGAR is able to complete the automaton and achieve the certified proof. The resulting automaton produced by the tool has 3688 transitions which are produced in $128s$. Then, it can be Coq-certified in $17017s$. The memory usage for the whole process does not exceed $531Mb$. One of the reasons for which certifying automata produced by TimbukCEGAR takes more time than for Timbuk 3.1 is that the checker has to normalize epsilon transitions of $\mathcal{R}_{/E}$-automata. This is straightforward but may cause an explosion of the size of the tree automaton to be checked. However this can be improved a lot by defining a specific Coq-checker for $\mathcal{R}_{/E}$-automata. It is worth mentioning that the TRS produced by Copster from Java programs are left-linear but not right-linear. Hence, our soundness theorem applies but not the completeness theorem, since the TRS is not right-linear and the initial language is not finite. However, here completion steps do not introduce spurious counter examples. Some other examples of application can be found in [10]. Those experiments show, in particular, that the overhead due to the use of $\mathcal{R}_{/E}$-automata in completion can be limited thanks to BDDs. As a consequence, TimbukCEGAR performances are similar to those of Timbuk 3.1 when no refinement is performed, and TimbukCEGAR outperforms the other known CEGAR completion implementation when refinement is needed.

8 Conclusion

We have presented a new CounterExample Guided Abstraction Refinement procedure for TRMC based on equational abstraction. Our approach has been implemented in TimbukCEGAR that is the first TRMC toolset certified correct. Our approach leads, in particular, to a Java program analyzer starting from code to verification. Unlike most of existing works, our method works with the Java

semantics itself rather than with some abstract model that is statically or even manually derived from the code. One additional feature is that the complete verification is certified by an external proof assistant. We are convinced that our work open news doors in application of RMC approaches to rigorous system design. One challenge for future work is definitively to consider non left-linear TRS – a formalism that can be used to model cryptographic protocols. In [25, 26, 28, 5], it has been shown that an extension of the completion algorithm can lead to a powerful verification toolset for cryptographic protocols. However, this existing setting is still lacking an abstraction refinement procedure to be made fully automatic. On the one side, the theoretical challenge is thus to extend the CEGAR completion to non left-linear TRS. On the other side, the technical challenge is to extend the Coq checker to handle non left-linear TRS and $\mathcal{R}_{/E}$-automata, in order to improve the Coq-checking time. Tackling those two goals will allow us to propose the *first certified automatic verification tool for security protocols*, a major advance in the formal verification area.

Acknowledgements Thanks to F. Besson for his help in integrating Buddy.

References

1. P. A. Abdulla, Y.-F. Chen, G. Delzanno, F. Haziza, C.-D. Hong, and A. Rezine. Constrained monotonic abstraction: A cegar for parameterized verification. In *CONCUR*, LNCS. Springer, 2010.
2. P. A. Abdulla, G. Delzanno, and A. Rezine. Parameterized verification of infinite-state processes with global conditions. In *CAV*, LNCS. Springer, 2007.
3. P. A. Abdulla, N. B. Henda, G. Delzanno, F. Haziza, and A. Rezine. Parameterized tree systems. In *FORTE*, volume 5048 of *LNCS*, pages 69–83. Springer, 2008.
4. P. A. Abdulla, A. Legay, A. Rezine, and J. d'Orso. Simulation-based iteration of tree transducers. In *TACAS*, volume 3440 of *LNCS*, pages 30–40. Springer, 2005.
5. Avispa – a tool for Automated Validation of Internet Security Protocols. `http://www.avispa-project.org`.
6. F. Baader and T. Nipkow. *Term Rewriting and All That*. Cambridge University Press, 1998.
7. T. Ball, B. Cook, V. Levin, and S. K. Rajamani. Slam and static driver verifier: Technology transfer of formal methods inside microsoft. In *IFM*, LNCS. Springer, 2004.
8. N. Barré, F. Besson, T. Genet, L. Hubert, and L. Le Roux. Copster homepage, 2009. `http://www.irisa.fr/celtique/genet/copster`.
9. Y. Bertot and P. Castéran. *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. Springer Verlag, 2004.
10. Y. Boichut, B. Boyer, T. Genet, and A. Legay. Fast Equational Abstraction Refinement for Regular Tree Model Checking. Technical report, INRIA, 2010. `http://hal.inria.fr/inria-00501487`.
11. Y. Boichut, R. Courbis, P.-C. Heam, and O. Kouchnarenko. Finer is better: Abstraction refinement for rewriting approximations. In *RTA*, LNCS. Springer, 2008.
12. Y. Boichut, T.-B.-H. Dao, and V. Murat. Characterizing conclusive approximations by logical formulae. In *RP*, volume 6945 of *LNCS*, pages 72–84. Springer, 2011.
13. Y. Boichut, T. Genet, T. Jensen, and L. Leroux. Rewriting Approximations for Fast Prototyping of Static Analyzers. In *RTA*, LNCS 4533, pages 48–62, 2007.

14. B. Boigelot, A. Legay, and P. Wolper. Iterating transducers in the large (extended abstract). In *CAV*, LNCS, pages 223–235. Springer, 2003.
15. A. Bouajjani, P. Habermehl, A. Rogalewicz, and T. Vojnar. Abstract regular tree model checking. *ENTCS*, 149(1):37–48, 2006.
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, P. Habermehl, and T. Vojnar. Abstract regular model checking. In *CAV*, volume 3114 of *LNCS*, pages 372–386. Springer, 2004.
18. A. Bouajjani, B. Jonsson, M. Nilsson, and T. Touili. Regular model checking. In *CAV*, volume 1855 of *LNCS*, pages 403–418. Springer-Verlag, 2000.
19. A. Bouajjani and T. Touili. Extrapolating tree transformations. In *CAV*, volume 2404 of *LNCS*, pages 539–554. Springer, 2002.
20. B. Boyer, T. Genet, and T. Jensen. Certifying a Tree Automata Completion Checker. In *IJCAR'08*, volume 5195 of *LNCS*. Springer, 2008.
21. H. Comon, M. Dauchet, R. Gilleron, F. Jacquemard, D. Lugiez, C. Löding, S. Tison, and M. Tommasi. Tree automata techniques and applications. 2008.
22. D. Dams, Y. Lakhnech, and M. Steffen. Iterating transducers. *Journal of Logic and Algebraic Programming (JLAP)*, 52-53:109–127, 2002.
23. G. Feuillade, T. Genet, and V. Viet Triem Tong. Reachability Analysis over Term Rewriting Systems. *Journal of Automated Reasonning*, 33 (3-4):341–383, 2004.
24. T. Genet. Decidable approximations of sets of descendants and sets of normal forms. In *RTA*, volume 1379 of *LNCS*, pages 151–165. Springer-Verlag, 1998.
25. T. Genet. Reachability analysis of rewriting for software verification. Université de Rennes 1, 2009. Habilitation.
26. T. Genet and F. Klay. Rewriting for Cryptographic Protocol Verification. In *Proc. 17th CADE*, volume 1831 of *LNAI*. Springer-Verlag, 2000.
27. T. Genet and R. Rusu. Equational tree automata completion. *JSC*, 45, 2010.
28. T. Genet, Y.-M. Tang-Talpin, and V. Viet Triem Tong. Verification of Copy Protection Cryptographic Protocol using Approximations of Term Rewriting Systems. In *WITS'2003*, 2003.
29. T. Genet and V. Viet Triem Tong. Timbuk 2.0 – a Tree Automata Library. IRISA / Université de Rennes 1, 2001. `http://www.irisa.fr/celtique/genet/timbuk/`.
30. R. Gilleron and S. Tison. Regular tree languages and rewrite systems. *Fundamenta Informaticae*, 24:157–175, 1995.
31. Y. Kesten, O. Maler, M. Marcus, A. Pnueli, and E. Shahar. Symbolic model checking with rich assertional languages. In *CAV*, LNCS. Springer, 1997.
32. J. Lind-Nielsen. Buddy 2.4, 2002. `http://buddy.sourceforge.net`.
33. J. Meseguer, M. Palomino, and N. Martí-Oliet. Equational abstractions. *TCS*, 403:239–264, 2008.
34. G. Patin, M. Sighireanu, and T. Touili. Spade: Verification of multithreaded dynamic and recursive programs. In *CAV*, LNCS. Springer, 2007.
35. A. Podelski and A. Rybalchenko. Armc: The logical choice for software model checking with abstraction refinement. In *PADL*, LNCS, 2007.
36. Ravaj: Rewriting and Approximations for Java Applications Verification. `http://www.irisa.fr/celtique/genet/RAVAJ`.
37. T. Takai. A Verification Technique Using Term Rewriting Systems and Abstract Interpretation. In *RTA*, volume 3091 of *LNCS*, pages 119–133. Springer, 2004.
38. A. Vardhan, K. Sen, M. Viswanathan, and G. Agha. Using language inference to verify omega-regular properties. In *TACAS*, LNCS. Springer, 2005.
39. A. Vardhan and M. Viswanathan. Lever: A tool for learning based verification. In *CAV*, LNCS. Springer, 2006.