Rewriting Approximations For Properties Verification Over CCS Specifications

Roméo Courbis

To cite this version:

Roméo Courbis. Rewriting Approximations For Properties Verification Over CCS Specifications. 2010. hal-00530351v2

HAL Id: hal-00530351

https://hal.archives-ouvertes.fr/hal-00530351v2

Preprint submitted on 10 Mar 2011

Rewriting Approximations For Properties Verification Over CCS Specifications

Roméo Courbis

INRIA/CASSIS
LIFC/University of Franche-Comté
16 route de Gray
F-25030 Besançon Cedex
rcourbis@lifc.univ-fcomte.fr

Abstract. This paper presents a way to verify CCS (without renaming) specifications using tree regular model checking. From a term rewriting system and a tree automaton representing the semantics of CCS and equations of a CCS specification to analyse, an over-approximation of the set of reachable terms is computed from an initial configuration. This set, in the framework of CCS, represents an over-approximation of all states (modulo bisimulation) and action sequences the CCS specification can reach. The approach described in this paper can be fully automated. It is illustrated with the Alternating Bit Protocol and with hardware components specifications.

1 Introduction

Model-checking techniques [20, 21] are commonplace in computer aided verification. Model checking refers to the following problem: given a desired property, expressed as a temporal logic formula φ, and a structure M with initial state s, decide if $M, s \models \varphi$. The use of model-checking techniques and tools is however limited to systems whose state space can be finitely and concisely represented.

Recently, reachability analysis turned out to be a very efficient verification technique for proving properties on infinite systems modeled by term rewriting systems (TRSs for short). In the rewriting theory, the reachability problem is the following: given a TRS $\mathcal{R}$ and two terms s and t, can we decide whether $\mathcal{R}^*(\{s\}) \cap \{t\} = \emptyset$ or not? This problem, which can easily be solved on strongly terminating TRSs, is undecidable on non terminating TRSs. However, on the one hand, there exist several syntactic classes of TRSs for which this problem becomes decidable [13, 18, 26]. On the other hand, in addition to classical proof tools of rewriting, given a set $\mathsf{E} \subseteq \mathcal{T}(\mathcal{F})$ of initial terms, provided that $s \in \mathsf{E}$, one can prove $\mathcal{R}^*(\{s\}) \cap \{t\} = \emptyset$ by using over-approximations of $\mathcal{R}^*(\mathsf{E})$ [13, 19] and proving that t does not belong to these approximations.

Motivations. Recently, some of the most successful experiments using reachability analysis were done on cryptographic protocols, [6, 16], and on Java byte code programs [5]. For example, Java MIDLet applications security properties are verified through $\mathcal{R}^*(\mathsf{E})$ over-approximations. To this end, following works on

CEGAR [8], an over-approximations refinement depending on a security property to be verified is developed in [4]. As TRSs and tree automata are powerful tools to express specifications, it is possible to perform reachability analysis on those. This paper fits in line with this context by adapting reachability analysis to verification of CCS (without renaming) specifications. Note that the reachability problem for this fragment of CCS is undecidable [7].

Contributions. This paper address the following problem : *Is it easy to adapt approximation rewriting to the verification of infinite state systems specified in CCS ?* The solution presented in this paper consists in a translation of a CCS specification into a TRS and a tree automaton. Then it is possible to verify properties using reachability analysis. This solution is illustrated with the Alternating Bit Protocol and with specifications of hardware components.

Structure of the paper. This paper is organised as follows. Section 2 introduces basic definitions of terms, TRSs, tree automata completion and CCS. Then Section 3 explains how to translate a CCS specification into a TRS and a tree automaton, and then how to verify properties on sequences of actions. Section 4 and Section 5 show applications of the technique presented in Section 3. Finally, Section 6 presents related works and the conclusion.

2 Preliminaries

Comprehensive surveys can be found in [1, 12] for TRSs, in [10, 17] for tree automata and tree language theory, and in [22] for CCS.

2.1 Terms and TRSs

Let $\mathcal{F}$ be a finite set of symbols, associated with an arity function $ar : \mathcal{F} \to \mathbb{N}$, 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 extended uniquely 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 ϵ denotes the top-most position. The set $\mathcal{P}os(t)$ of positions of a term t is inductively defined by $\mathcal{P}os(t) = \{\epsilon\}$ if $t \in \mathcal{X}$ and by $\mathcal{P}os(f(t_1, \dots, t_n)) = \{\epsilon\} \cup \{i.p \mid 1 \leq i \leq n \text{ and } p \in \mathcal{P}os(t_i)\}$ 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. We also denote by $t(p)$ the symbol occurring in t at position p. Given a term $t \in \mathcal{T}(\mathcal{F}, \mathcal{X})$, we denote $\mathcal{P}os_A(t) \subseteq \mathcal{P}os(t)$ the set of positions of t such that $\mathcal{P}os_A(t) = \{p \in \mathcal{P}os(t) \mid t(p) \in A\}$. Thus $\mathcal{P}os_{\mathcal{F}}(t)$ is the set of functional positions of t. A TRS $\mathcal{R}$ is a set of *rewrite rules* $l \to r$, where $l, r \in \mathcal{T}(\mathcal{F}, \mathcal{X})$ and $l \notin \mathcal{X}$. A rewrite rule $l \to r$ is *left-linear* (resp. right-linear) if each variable of l (resp. r) occurs only once within l (resp. r). A TRS $\mathcal{R}$ is left-linear (resp. right-linear) if every rewrite rule $l \to r$ of $\mathcal{R}$ is left-linear (resp. right-linear). A TRS $\mathcal{R}$ is linear if it is right and left-linear. The TRS $\mathcal{R}$ induces a rewriting relation $\to_{\mathcal{R}}$ on terms whose reflexive transitive closure is written $\to^{\star}_{\mathcal{R}}$. The set of $\mathcal{R}$-descendants of a set of ground terms E is $\mathcal{R}^*(\mathsf{E}) = \{t \in \mathcal{T}(\mathcal{F}) \mid \exists s \in \mathsf{E} \text{ s.t. } s \to^{\star}_{\mathcal{R}} t\}$.

2.2 Tree Automata Completion

Note that $\mathcal{R}^*(\mathsf{E})$ is possibly infinite: $\mathcal{R}$ may not terminate and/or E may be infinite. The set $\mathcal{R}^*(\mathsf{E})$ is generally not computable [17]. However, it is possible to over-approximate it [13] using tree automata, i.e. a finite representation of infinite (regular) sets of terms. We next define tree automata.

Let $\mathcal{Q}$ be a finite set of symbols, of arity 0, called *states* such that $\mathcal{Q} \cap \mathcal{F} = \emptyset$. $\mathcal{T}(\mathcal{F} \cup \mathcal{Q})$ is called the set of *configurations*.

Definition 1 (Transition and normalised transition). *A* transition *is a rewrite rule $c \to q$, where $c \in \mathcal{T}(\mathcal{F} \cup \mathcal{Q})$ is a configuration and $q \in \mathcal{Q}$. A* normalised transition *is a transition $c \to q$ where $c = f(q_1, \ldots, q_n)$, $f \in \mathcal{F}$, $ar(f) = n$, and $q_1, \ldots, q_n \in \mathcal{Q}$.*

Definition 2 (Bottom-up non-deterministic finite tree automaton). *A bottom-up non-deterministic finite tree automaton (tree automaton for short) is a quadruple $\mathcal{A} = (\mathcal{F}, \mathcal{Q}, \mathcal{Q}_f, \Delta)$, $\mathcal{Q}_f \subseteq \mathcal{Q}$ and Δ is a finite set of normalised transitions.*

The *rewriting relation* on $\mathcal{T}(\mathcal{F} \cup \mathcal{Q})$ induced by the transition set Δ of $\mathcal{A}$ is denoted $\to_\Delta$. When Δ is clear from the context, $\to_\Delta$ is also written $\to_\mathcal{A}$.

Definition 3 (Recognised language). *The tree language recognised by $\mathcal{A}$ in a state q is $L(\mathcal{A}, q) = \{t \in \mathcal{T}(\mathcal{F}) \mid t \to_\mathcal{A}^\star q\}$. The language recognised by $\mathcal{A}$ is $L(\mathcal{A}) = \bigcup_{q \in \mathcal{Q}_f} L(\mathcal{A}, q)$. A tree language is regular if and only if it is recognised by a tree automaton.*

Let us now recall how tree automata and TRSs can be used for term reachability analysis. Given a tree automaton $\mathcal{A}$ and a TRS $\mathcal{R}$, the tree automata completion algorithm proposed in [13] computes a tree automaton $\mathcal{A}_\mathcal{R}^k$ such that $L(\mathcal{A}_\mathcal{R}^k) = \mathcal{R}^*(L(\mathcal{A}))$ when it is possible (for the classes of TRSs where an exact computation is possible, see [13]), and such that $L(\mathcal{A}_\mathcal{R}^k) \supseteq \mathcal{R}^*(L(\mathcal{A}))$ otherwise.

The tree automata completion works as follows. From $\mathcal{A} = \mathcal{A}_\mathcal{R}^0$ the completion builds a sequence $\mathcal{A}_\mathcal{R}^0, \mathcal{A}_\mathcal{R}^1 \ldots \mathcal{A}_\mathcal{R}^k$ of automata such that if $s \in L(\mathcal{A}_\mathcal{R}^i)$ and $s \to_\mathcal{R} t$ then $t \in L(\mathcal{A}_\mathcal{R}^{i+1})$. If there is a fix-point automaton $\mathcal{A}_\mathcal{R}^k$ such that $\mathcal{R}^*(L(\mathcal{A}_\mathcal{R}^k)) = L(\mathcal{A}_\mathcal{R}^k)$, then $L(\mathcal{A}_\mathcal{R}^k) = \mathcal{R}^*(L(\mathcal{A}_\mathcal{R}^0))$ (or $L(\mathcal{A}_\mathcal{R}^k) \supseteq \mathcal{R}^*(L(\mathcal{A}))$ if $\mathcal{R}$ is in no class of [13]). To build $\mathcal{A}_\mathcal{R}^{i+1}$ from $\mathcal{A}_\mathcal{R}^i$, a *completion step* is achieved. It consists of finding *critical pairs* between $\to_\mathcal{R}$ and $\to_{\mathcal{A}_\mathcal{R}^i}$. To define the notion of critical pair, the substitution definition is extended to terms in $\mathcal{T}(\mathcal{F} \cup \mathcal{Q})$. For a substitution $\sigma : \mathcal{X} \mapsto \mathcal{Q}$ and a rule $l \to r \in \mathcal{R}$ such that $\mathcal{V}ar(r) \subseteq \mathcal{V}ar(l)$, if there exists $q \in \mathcal{Q}$ satisfying $l\sigma \to_{\mathcal{A}_\mathcal{R}^i}^* q$ then $l\sigma \to_{\mathcal{A}_\mathcal{R}^i}^* q$ and $l\sigma \to_\mathcal{R} r\sigma$ is a critical pair. Note that since $\mathcal{R}$ and $\mathcal{A}_\mathcal{R}^i$ are finite, there is only a finite number of critical pairs. Thus, for every critical pair detected between $\mathcal{R}$ and $\mathcal{A}_\mathcal{R}^i$ such that $r\sigma \not\to_{\mathcal{A}_\mathcal{R}^i}^* q$, the tree automaton $\mathcal{A}_\mathcal{R}^{i+1}$ is constructed by adding a new transition $r\sigma \to q$ to $\mathcal{A}_\mathcal{R}^i$. Consequently, $\mathcal{A}_\mathcal{R}^{i+1}$ recognises $r\sigma$ in q, i.e. $r\sigma \to_{\mathcal{A}_\mathcal{R}^{i+1}} q$.

$$\begin{tikzcd}
l\sigma \arrow[r, "\mathcal{R}"'] \arrow[d, "\mathcal{A}_\mathcal{R}^i"', "*"] & r\sigma \arrow[dl, dashed, "*"', "\mathcal{A}_\mathcal{R}^{i+1}"] \\
q &
\end{tikzcd}$$

However, the transition $r\sigma \to q$ is not necessarily a normalized transition of the form $f(q_1, \ldots, q_n) \to q$ and so it has to be normalized first. For example, to normalize a transition of the form $f(g(a), h(q')) \to q$, we need to find some states q_1, q_2, q_3 and replace the previous transition by a set of normalized transitions: $\{a \to q_1, g(q_1) \to q_2, h(q') \to q_3, f(q_2, q_3) \to q\}$.

If q_1, q_2, q_3 are new states, then adding the transition itself or its normalized form does not make any difference. On the opposite, if we identify q_1 with q_2, the normalized form becomes $\{a \to q_1, g(q_1) \to q_1, h(q') \to q_3, f(q_1, q_3) \to q\}$. This set of normalized transitions represents the regular set of non-normalized transitions of the form $f(g^*(a), h(q')) \to q$ which contains the transition we want to add but also many others. Hence, this is an over-approximation. We could have made an even more drastic approximation by identifying q_1, q_2, q_3 with q, for instance.

When always using a new states to normalize the transitions, completion is as precise as possible. However, without approximation, completion is likely not to terminate (because of general undecidability results [17]). To enforce termination, and produce an over-approximation, the completion algorithm is parametrized by a set N of *approximation rules*. When the set N is used during completion to normalize transitions, the obtained tree automata are denoted by $\mathcal{A}^1_{N,\mathcal{R}}, \ldots, \mathcal{A}^k_{N,\mathcal{R}}$. Each such rule describes a context in which a list of rules can be used to normalize a term. For all $s, l_1, \ldots, l_n \in \mathcal{T}(\mathcal{F} \cup \mathcal{Q}, \mathcal{X})$ and for all $x, x_1, \ldots, x_n \in \mathcal{Q} \cup \mathcal{X}$, the general form for an approximation rule is:
$[s \to x] \to [l_1 \to x_1, \ldots, l_n \to x_n]$. The expression $[s \to x]$ is a pattern to be matched with the new transition $t \to q'$ obtained by completion. The expression $[l_1 \to x_1, \ldots, l_n \to x_n]$ is a set of rules used to normalize t. to normalize a transition of the form $t \to q'$, we match s with t and x with q', obtain a substitution σ from the matching and then we normalize t with the rewrite system $\{l_1\sigma \to x_1\sigma, \ldots, l_n\sigma \to x_n\sigma\}$. Furthermore, if $\forall i = 1 \ldots n : x_i \in \mathcal{Q}$ or $x_i \in \mathcal{V}ar(l_i) \cup \mathcal{V}ar(s) \cup \{x\}$ then $x_1\sigma, \ldots, x_n\sigma$ are necessarily states. If a transition cannot be fully normalized using approximation rules N, normalization is finished using some new states.

The main property of the tree automata completion algorithm is that, whatever the state labels used to normalize the new transitions, if completion terminates then it produces an over-approximation of reachable terms [13]. In other words, approximation safety does not depend on the set of approximation rules used. Since the role of approximation rules is only to select particular states for normalizing transitions, the safety theorem of [13] can be reformulated in the following way.

Theorem 1. *Let $\mathcal{A}$ be a tree automaton, N be a set of approximation rules and $\mathcal{R}$ be a left-linear TRS such that for every $l \to r \in \mathcal{R}$, $\mathcal{V}ar(r) \subseteq \mathcal{V}ar(l)$. If completion terminates on $\mathcal{A}^k_{N,\mathcal{R}}$ then*

$$L(\mathcal{A}^k_{N,\mathcal{R}}) \supseteq \mathcal{R}^*(L(\mathcal{A}))$$

Here is a simple example illustrating completion and the use of approximation rules when the language $\mathcal{R}^*(\mathsf{E})$ is not regular.

Example 1. Let $\mathcal{R} = \{g(x,y) \to g(f(x), f(y))\}$ and let $\mathcal{A}$ be a tree automaton such that $\mathcal{Q}_f = \{q_f\}$ and $\Delta = \{a \to q_a, g(q_a, q_a) \to q_f\}$. Hence $L(\mathcal{A}) = \{g(a,a)\}$ and $R^*(L(\mathcal{A})) = \{g(f^n(a), f^n(a)) \mid n \geq 0\}$. Let $N = [g(f(x), f(y)) \to z] \to [f(x) \to q_1 \ f(y) \to q_1]$. During the first completion step, we find a critical pair $g(q_a, q_a) \to_{\mathcal{R}} g(f(q_a), f(q_a))$ and $g(q_a, q_a) \to^*_{\mathcal{A}} q_f$. We thus have to add the transition $g(f(q_a), f(q_a)) \to q_f$ to $\mathcal{A}$. To normalize this transition, we match $g(f(x), f(y))$ with $g(f(q_a), f(q_a))$ and match z with q_f and obtain $\sigma = \{x \mapsto q_a, y \mapsto q_a, z \mapsto q_f\}$. Applying σ to $[f(x) \to q_1 f(y) \to q_1]$ gives $[f(q_a) \to q_1 f(q_a) \to q_1]$. This last system is used to normalize the transition $g(f(q_a), f(q_a)) \to q_f$ into the set $\{g(q_1, q_1) \to q_f, f(q_a) \to q_1\}$ which is added to $\mathcal{A}$ to obtain $\mathcal{A}^1_{N,\mathcal{R}}$. The completion process continues for another step and ends on $\mathcal{A}^2_{N,\mathcal{R}}$ whose set of transition is $\{a \to q_a, g(q_a, q_a) \to q_f, g(q_1, q_1) \to q_f, f(q_a) \to q_1, f(q_1) \to q_1\}$. We have $L(\mathcal{A}^2_{N,\mathcal{R}}) = \{g(f^n(a), f^m(a)) \mid n, m \geq 0\}$ which is an over-approximation of $\mathcal{R}^*(L(\mathcal{A}))$.

2.3 The Calculus of Communicating Systems

Syntax. Let $A = \{a, b, c, \ldots\}$ be the set of names and $\bar{A} = \{\bar{a}, \bar{b}, \bar{c}, \ldots\}$ be the set of co-names. Let $\text{Ł} = A \cup \bar{A}$ be a set of labels, and let τ be the invisible action such that $\tau \notin \text{Ł}$. Let $Act = \text{Ł} \cup \{\tau\}$ be the set of actions. Let $\mathcal{P}$ be a set of process names, and let $\mathbf{0} \in \mathcal{P}$ be the inactive process. Let $\mathcal{E}$ be the set of restricted CCS expressions defined according to the following syntax:
$E, E_1, E_2 := \alpha.E \mid E_1 + E_2 \mid E_1 \parallel E_2 \mid E \setminus \ell \mid \mathbf{0} \mid P$
where $\alpha, \ell \in Act$, $E, E_1, E_2 \in \mathcal{E}$ and $P \in \mathcal{P}$. Process names $P \in \mathcal{P}$ are defined such that for all P and $E \in \mathcal{E}$, one has : $P \overset{def}{=} E$. The set $Action(E)$ of actions is inductively defined by $Action(\alpha.E) = \{\alpha\} \cup Action(E)$, $Action(\mathbf{0}) = \emptyset$, $Action(P) = Action(E)$ (with $P \overset{def}{=} E$) and $Action(E_1 + E_2) = Action(E_1 \parallel E_2) = Action(E_1) \cup Action(E_2)$. The set of actions $ResAction(E)$ is inductively defined by: $ResAction(E \setminus \ell) = ResAction(E) \cup \{\ell\}$ and $ResAction(\alpha.E) = ResAction(E)$, $ResAction(\mathbf{0}) = \emptyset$, $ResAction(P) = ResAction(E)$ (with $P \overset{def}{=} E$) and $ResAction(E_1 + E_2) = ResAction(E_1 \parallel E_2) = ResAction(E_1)$ $\cup ResAction(E_2)$. The set $Subterm(E)$ of CCS expressions is inductively defined by $Subterm(\alpha.E) = \{\alpha.E\} \cup Subterm(E)$, $Subterm(\mathbf{0}) = \emptyset$, $Subterm(P) = Subterm(E)$ (with $P \overset{def}{=} E$), $Subterm(E_1 + E_2) = \{E_1 + E_2\} \cup Subterm(E_1) \cup Subterm(E_2)$ and $Subterm(E_1 \parallel E_2) = \{E_1 \parallel E_2\} \cup Subterm(E_1) \cup Subterm(E_2)$. A CCS expression E' is a sub-term of E, or E contains E', if $E' \in Subterm(E)$.

CCS programs. A CCS program S is a 3-tuple $S = (\Lambda, \Gamma, P_0)$ where $\Lambda \subseteq Act$, $\Gamma \subseteq \mathcal{P} \times \mathcal{E}$ is a finite set of equations, denoted by (P, E) or by $P \overset{def}{=} E$, and $P_0 \in dom(\Gamma)$ is the head process name, which usually builds the complete system. For example if we have : $A \overset{def}{=} a.B$, $B \overset{def}{=} b.B$, $S = (\{a, b\}, \{(A, a.B), (B, b.B)\}, A)$ is a CCS program.

Semantics. A CCS program $S = (\Lambda, \Gamma, P_0)$ defines the labeled transition system (LTS) $T_{CCS} \subseteq \mathcal{E} \times \Lambda \times \mathcal{E}$, built according to inference rules in Fig. 1. A transition

$$\textbf{Act}\ \frac{}{\alpha.E \overset{\alpha}{\to} E} \qquad \textbf{Com}_\mathbf{1}\ \frac{E_1 \overset{\alpha}{\to} E_1'}{E_1 \parallel E_2 \overset{\alpha}{\to} E_1' \parallel E_2}$$

$$\textbf{Sum}_\mathbf{1}\ \frac{E_1 \overset{\alpha}{\to} E_1'}{E_1 + E_2 \overset{\alpha}{\to} E_1'} \qquad \textbf{Com}_\mathbf{2}\ \frac{E_2 \overset{\alpha}{\to} E_2'}{E_1 \parallel E_2 \overset{\alpha}{\to} E_1 \parallel E_2'}$$

$$\textbf{Sum}_\mathbf{2}\ \frac{E_2 \overset{\alpha}{\to} E_2'}{E_1 + E_2 \overset{\alpha}{\to} E_2'} \qquad \textbf{Com}_\mathbf{3}\ \frac{E_1 \overset{a}{\to} E_1' \quad E_2 \overset{\bar{a}}{\to} E_2'}{E_1 \parallel E_2 \overset{\tau}{\to} E_1' \parallel E_2'}$$

$$\textbf{Res}\ \frac{E \overset{\alpha}{\to} E'}{E \setminus \ell \overset{\alpha}{\to} E' \setminus \ell} \text{ if } \alpha, \bar{\alpha} \neq \ell \in Ł$$

$$\textbf{Con}\ \frac{E \overset{\alpha}{\to} E'}{P \overset{\alpha}{\to} E'} \text{ if } (P, E) \in \Gamma$$

Fig. 1. Inference rules of CCS

$E \overset{\alpha}{\to} E'$ will denote the 3-uplet $(E, \alpha, E') \in T_E$. In this context, CCS expressions E and E' can be called states.

As a CCS program $S = (\Lambda, \Gamma, P_0)$ has a head process, the initial state of the corresponding LTS is the state (or process) P_0.

A CCS expression E can perform an action α and becomes a CCS expression E' if the transition $E \overset{\alpha}{\to} E'$ can be inferred by the rules of Fig. 1. For example, the transition $(a.b.\mathbf{0} + c.\mathbf{0}) \parallel d.\mathbf{0} \overset{a}{\to} b.\mathbf{0} \parallel d.\mathbf{0}$, can be inferred by rules $\mathbf{Com_1}$, $\mathbf{Sum_1}$ and **Act**.

Derivatives. If $E \overset{\alpha}{\to} E'$, the pair (α, E') is called the *immediate derivative* of E. If $E \overset{\alpha_0}{\to} \dots \overset{\alpha_n}{\to} E'$, the pair $(\alpha_0 \dots \alpha_n, E')$ is called a *derivative* of E, where $\alpha_0 \dots \alpha_n$ is an action sequence.

Let $deriv(E)$ be the set of all derivatives of E such that
$deriv(E) = \{(\alpha_0 \dots \alpha_n, E') \mid E' \in \mathcal{E},\ E \overset{\alpha_0}{\to} \dots \overset{\alpha_n}{\to} E'\}$.

3 Rewriting Approximations for CCS

Section 3 shows how to encode a CCS program into a TRS $\mathcal{R}$ and an initial automaton $\mathcal{A}$. The aim is to compute an over-approximation of $\mathcal{R}^*(L(\mathcal{A}))$ representing an over-approximation of all derivatives of a CCS program and, then, to verify properties such as absence of specific succession of actions.

3.1 Representation of a CCS program and semantics with terms and TRS

Terms for CCS expressions. A term corresponding to a CCS expression in $\mathcal{E}$ is built by induction on the structure of the CCS expression. Let $\mathcal{F}_{CCS}$ be an alphabet such that $\mathcal{F}_{CCS} = \mathcal{F}_0 \cup \mathcal{F}_1 \cup \mathcal{F}_2 \cup \mathcal{F}_3$, where $\mathcal{F}_0 = \{\mathbf{0}\}$, $\mathcal{F}_1 = \{bar\} \cup Act$, $\mathcal{F}_2 = \{Pre, Sum, Com, Res, Sys\}$. Let $\Phi : \mathcal{E} \to \mathcal{T}(\mathcal{F}_{CCS})$ be the function such that:

$\Phi(\alpha.E) = Pre(\Phi(\alpha), \Phi(E))$
$\Phi(E_1 + E_2) = Sum(\Phi(E_1), \Phi(E_2))$
$\Phi(E_1 \parallel E_2) = Com(\Phi(E_1), \Phi(E_2))$

$\Phi(P) = P$, if $P \in \mathcal{P}$
$\Phi(E \setminus \ell) = Res(\Phi(E), \ell)$
$\Phi(\mathbf{0}) = \mathbf{0}$

$$\Phi(\alpha) = \begin{cases} \alpha & \text{if } \alpha \in A \\ bar(a) & \text{if } \alpha = \bar{a} \text{ and } \alpha \in \bar{A} \end{cases}$$

Example 2. Let $E = (a.b.\mathbf{0} + c.\mathbf{0}) \parallel d.\mathbf{0}$ be in $\mathcal{E}$. The term corresponding to this expression is:

$$\begin{aligned} \Phi(E) &= Com(\Phi(a.b.\mathbf{0} + c.\mathbf{0}), \Phi(d.\mathbf{0}) \\ &= Com(Sum(\Phi(a.b.\mathbf{0}), \Phi(c.\mathbf{0})), Pre(d, \Phi(\mathbf{0}))) \\ &= Com(Sum(Pre(a, \Phi(b.\mathbf{0})), Pre(c, \Phi(\mathbf{0}))), Pre(d, \mathbf{0})) \\ &= Com(Sum(Pre(a, Pre(b, \Phi(\mathbf{0}))), Pre(c, \mathbf{0})), Pre(d, \mathbf{0})) \\ &= Com(Sum(Pre(a, Pre(b, \mathbf{0})), Pre(c, \mathbf{0})), Pre(d, \mathbf{0})) \end{aligned}$$

Terms for derivatives. Let E be in $\mathcal{E}$. A derivative $(\alpha_0 \dots \alpha_n, E)$ is encoded into a term of the type $Sys(\alpha_0, Sys(\dots, Sys(\alpha_n, \Phi(E))))$. Formally, the encoding function $\Psi : deriv(E) \times \mathcal{T}(\mathcal{F}_{CCS})$ is defined by:

$\Psi((\alpha, E)) = Sys(\alpha, \Phi(E))$
$\Psi((\alpha_0 \dots \alpha_n, E)) = Sys(\alpha_0, \Psi((\alpha_1 \dots \alpha_n, E)))$

Rewriting rules for CCS semantics. Rewriting rules corresponding to CCS semantic are in Figure 2.

ρ_1	$Pre(x, p)$	$\to Sys(x, p)$
ρ_2	$Sum(Sys(x, p), r)$	$\to Sys(x, p)$
ρ_3	$Sum(r, Sys(x, p))$	$\to Sys(x, p)$
ρ_4	$Com(Sys(x, p), r)$	$\to Sys(x, Com(p, r))$
ρ_5	$Com(r, Sys(x, p))$	$\to Sys(x, Com(r, p))$
ρ_6	$Com(Sys(x, p), Sys(bar(x), r))$	$\to Sys(\tau, Com(p, r))$
ρ_7	$Com(Sys(bar(x), p), Sys(x, r))$	$\to Sys(\tau, Com(p, r))$
ρ_8	$Res(Sys(x, p), y)$	$\to Sys(x, Res(p, y))$

Fig. 2. Rewriting rules for CCS semantics

Let $\mathcal{R}_{sem}^{\vartheta}$ denote the TRS defined by $\mathcal{R}_{sem}^{\vartheta} = \{\rho_1, \dots, \rho_5\} \cup \{l\sigma \to r\sigma \mid \sigma = (x, \alpha),\ \alpha \in \vartheta,\ l \to r \in \{\rho_6, \rho_7\}\}$, where $\vartheta \subseteq Act$. Let $\mathcal{R}_{res}^{\Theta_1, \Theta_2}$ be the TRS defined by $\mathcal{R}_{res}^{\Theta_1, \Theta_2} = \{\rho_8\sigma \mid \sigma(x) = \alpha, \sigma(y) = \beta, \alpha \in \Theta_1, \beta \in \Theta_2, \alpha \neq \beta\}$, where $\Theta_1, \Theta_2 \subseteq Act$. The right part of the union in $\mathcal{R}_{sem}^{\vartheta}$ is made to have a left-linear TRS (as rewriting rules ρ_6 and ρ_7 are not left-linear) because completion algorithm requires a left-linear TRS to be correct. And, let $\mathcal{R}_{Con}^{\theta}$ denotes the TRS defined such that $\mathcal{R}_{Con}^{\theta} = \{\Phi(P) \to \Phi(E) \mid (P, E) \in \theta\}$, where $\theta \subseteq \mathcal{P} \times \mathcal{E}$.

Now, we can define a TRS and a tree automaton corresponding to a CCS program.

Given a CCS program $S = (\Lambda, \Gamma, P_0)$, let us denote by L_S the tree language defined such that $L_S = \{\Phi(P_0)\}$, and let us denote by $\mathcal{R}_S$ the TRS defined such that $\mathcal{R}_S = \mathcal{R}_{sem}^{\Lambda} \cup \mathcal{R}_{Con}^{\Gamma} \cup \mathcal{R}_{res}^{\Lambda, \Lambda'}$, where $\Lambda' = ResAction(E) \cup ResAction(P)$ for all $(P, E) \in \Gamma$. The TRS $\mathcal{R}_{Con}^{\Gamma}$ corresponds to the **Con** inference rule. The set of actions Λ' is the set of all actions ℓ used for the restriction in the definition of S. Thereafter, we will use the TRS $\mathcal{R}_{sr}^{\Lambda,\Lambda'} = \mathcal{R}_{sem}^{\Lambda} \cup \mathcal{R}_{res}^{\Lambda,\Lambda'}$.

The main idea is to compute the set $\mathcal{R}_S^*(L_S)$, representing all derivatives of P_0, and, then, compute the intersection between $\mathcal{R}_S^*(L_S)$ and a set of derivatives. Intuitively, the TRS $\mathcal{R}_{sem}^{\vartheta}$ rewrites a term $\Phi(E)$ into a term $Sys(\alpha, \Phi(E'))$, if it is possible, by rewriting leafs to the root. This process can be viewed as a derivation of a transition $E \xrightarrow{\alpha} E'$ by inference rules, but, in a reversed way. Moreover, the TRS $\mathcal{R}_{Con}^{\theta}$ corresponds to equation(s) of a CCS program and handles recursion in equations.

Example 3. Let $E = (a.b.\mathbf{0} + c.\mathbf{0}) \parallel d.\mathbf{0}$ be in $\mathcal{E}$. According to CCS semantics we have the transition $E \xrightarrow{a} b.\mathbf{0} \parallel d.\mathbf{0}$, justified by inference rules $\mathbf{Com_1}$, $\mathbf{Sum_1}$ and **Act**. With the help of the TRS $\mathcal{R}_{sem}^{Action(E)}$ rules, the term $\Phi(E)$ is rewritten $Sys(a, (Com(Pre(b, \mathbf{0}), Pre(d, \mathbf{0}))))$. More precisely we have :

$$\begin{aligned}
&Com(Sum(Pre(a, Pre(b, \mathbf{0})), Pre(c, \mathbf{0})), Pre(d, \mathbf{0}))\\
&\qquad \to_{\rho_1} Com(Sum(Sys(a, Pre(b, \mathbf{0})), Pre(c, \mathbf{0})), Pre(d, \mathbf{0}))\\
&\qquad \to_{\rho_2} Com(Sys(a, Pre(b, \mathbf{0})), Pre(d, \mathbf{0}))\\
&\qquad \to_{\rho_4} Sys(a, Com(Pre(b, \mathbf{0}), Pre(d, \mathbf{0})))
\end{aligned}$$

As we can see, it is possible to draw a parallel between proving that one has $E \xrightarrow{a} b.\mathbf{0} \parallel d.\mathbf{0}$ with inference rules of Fig. 1, and rewriting $\Phi(E)$ into $\Psi((a, b.\mathbf{0} \parallel d.\mathbf{0}))$: rule ρ_1 matches with the inference rule **Act**, rule ρ_2 with $\mathbf{Sum_1}$, and rule ρ_4 with $\mathbf{Com_1}$.

Lemma 1. *Let α be in Act and E_s, E be in $\mathcal{E}$. If $\alpha.E_s \in Subterm(E)$ then $Pre(\alpha, \Phi(E_s))$ is a sub-term of $\Phi(E)$.*

Proof. We will show that there exists a position $p \in \mathcal{P}os(\Phi(E))$ such that $\Phi(E)|_p = Pre(\alpha, \Phi(E_s))$ by structural induction on E:

Case 1: $E = \beta.E'$

Case 1.1: $\beta = \alpha$ and $E' = E_s$

According to definition of Φ we have $\Phi(E) = \Phi(\alpha.E_s) = Pre(\alpha, \Phi(E_s))$ and $p = \epsilon$.

Case 1.2: $\beta \neq \alpha$ or $E' \neq E_s$

One has $\Phi(\beta.E') = Pre(\beta, \Phi(E'))$ with $\alpha.E_s$ is a sub-term of E' and, by induction hypothesis, there exists a position $p = 2.p'$ such that $\Phi(E')|_{p'} = Pre(\alpha, \Phi(E_s)$. Then, the proof is by induction on E', thus $p = 2.p'$ satisfies the requirement.

Case 2: $E = E_1 + E_2$

Case 2.1: $\alpha.E_s$ is a sub-term of E_1

According to the definition of Φ we have :

$\Phi(E) = \Phi(E_1 + E_2) = Sum(\Phi(E_1), \Phi(E_2))$ with $\alpha.E_s$ a sub-term of E_1. By induction hypothesis, one has $p = 1.p'$ such that $\Phi(E_1)|_{p'} = Pre(\alpha, \Phi(E_s))$. Then, the proof is by induction on E_1.

Case 2.2: $\alpha.E_s$ is a sub-term of E_2

Similar to case 2.1.

Case 3: $E = E_1 \parallel E_2$

Similar to case 2.

Case 4: $E = E' \setminus \ell$

According to definition of Φ we have $\Phi(E) = Res(\Phi(E'), \ell)$ with $\alpha.E_s$ a sub-term of E', and one has $p = 1.p'$ such that $\Phi(E')|_{p'} = Pre(\alpha, \Phi(E_s))\}$. Then, the proof is by induction on E'.

There are no cases $E = \mathbf{0}$ or $E = P$ with $P \in \mathcal{P}$, because the condition of lemma 1 $\alpha.E_s \in Subterm(E)$ is not verified. □

Proposition 1. *Let E and E' be two CCS expressions, let $\alpha \in Act$, let $A_E = Action(E)$ and $A'_E = ResAction(E)$. If $E \xrightarrow{\alpha} E'$ then*

$$Sys(\alpha, \Phi(E')) \in \mathcal{R}_{sr}^{A_E A'_E *}(\Phi(E))$$

Proof. Assuming that $E \xrightarrow{\alpha} E'$, we will show there exists a sequence of rewriting rules $r_0, \ldots, r_n \in \mathcal{R}_{sr}^{A_E A'_E *}$ and a sequence of terms $t_0, \ldots, t_n \in \mathcal{T}(\mathcal{F}_{CCS})$ such that $\Phi(E) = t_0 \rightarrow_{r_0} \ldots \rightarrow_{r_n} t_n = Sys(\alpha, \Phi(E'))$. (1)

We begin by proving that $r_0 = \rho_1$. In fact, as $\mathcal{P}os_{\{Sys\}}(t_0) = \emptyset$, only rule ρ_1 can be applied to t_0. As $E \xrightarrow{\alpha} E'$, E contains a sub-term of the form $\alpha.E_s$, then, according to Lemma 1, there exists a position $p \in \mathcal{P}os(\Phi(E))$ such that $\Phi(E)|_p = Pre(\alpha, \Phi(E_s))$. We can conclude that there exists a substitution $\sigma : \mathcal{X} \rightarrow \mathcal{T}(\mathcal{F}_{CCS})$ such that $t_0 \rightarrow_{\rho_1} t_0[r_{\rho_1}\sigma]_p$ (with $\rho_1 = l_{\rho_1} \rightarrow r_{\rho_1}$). (2)

Now we have to show (1) using (2) by transition induction on the depth of the inference by which the action $E \xrightarrow{\alpha} E'$ is inferred. We argue by cases on the form of E and its sub-terms:

Case 1: $E = \beta.E_1$

As $E \xrightarrow{\alpha} E'$, one has $\beta = \alpha$ and $E_1 = E'$. Then, using (2), we have $\Phi(E) = Pre(\alpha, \Phi(E'))$ and $\Phi(E) \rightarrow_{\rho_1} Sys(\alpha, \Phi(E'))$. One can conclude that $Sys(\alpha, \Phi(E')) \in \mathcal{R}_{sr}^{A_E A'_E *}(\Phi(E))$.

Case 2: $E_3 = E_1 + E_2$, where E_3 is a sub-term of E

Case 2.1: (α, E'_1) is a derivative of E_1

According to the definition of Φ, one has $\Phi(E_3) = Sum(\Phi(E_1), \Phi(E_2))$. As $\Phi(E_1) = Sys(\alpha, \Phi(E'_1))$, and by induction hypothesis (1), there exists a substitution $\sigma_1 : \mathcal{X} \rightarrow \mathcal{T}(\mathcal{F}_{CCS})$ such that $l_{\rho_2}\sigma_1 \rightarrow_{\rho_2} r_{\rho_2}\sigma_1$. We obtain $\Phi(E_3) \rightarrow_{\rho_2} Sys(\alpha, \Phi(E'_1))$. If $E_3 = E$ then Proposition 1 is proved, else the proof continues by induction on a sub-term of E containing E_3.

Case 2.2: (α, E'_2) is a derivative of E_2

Similar to case 2.1.

Case 3: $E_3 = E_1 \parallel E_2$

Similar to case 2.

Case 4: $E_2 = E_1 \setminus \ell$, where E_2 is a sub-term of E such that (α, E'_1) if a derivative of E_1.

According to the Φ definition, one has $\Phi(E_2) = Res(\Phi(E_1), \Phi(\ell))$. As $\Phi(E_1) = Sys(\alpha, \Phi(E'_1))$, one has $\Phi(E_2) = Res(Sys(\alpha, \Phi(E'_1)), \Phi(\ell))$.

One obtains $\Phi(E_2) \rightarrow_{\rho_8} Sys(\alpha, Res(\Phi(E'_1), \Phi(\ell)))$. If $E_2 = E$ then Proposition 1 is proved, else the proof continues by induction on a sub-term of E containing E_2.

□

Directly from Proposition 1, we can deduce that for all $D \in Deriv(E)$ one has $\Psi(D) \in \mathcal{R}_{sr}^{\Lambda_E \Lambda'_E *}(\Phi(E))$. Moreover, for CCS programs (and not only CCS expressions as in Proposition 1) we have the following proposition:

Proposition 2. *Let $S = (\Lambda, \Gamma, P_0)$ be a CCS program. If $d \in Deriv(P_0)$ then $\Psi(d) \in \mathcal{R}_S^*(L_S)$.*

Proof. We will show that $\Psi(P_0) \rightarrow^*_{\mathcal{R}_S} \Psi(d)$. As $d \in Deriv(P_0)$, one has $d = (\alpha_0 \ldots \alpha_n, E_n)$ and by definition one has $P_0 \overset{\alpha_0}{\rightarrow} E_1 \ldots \overset{\alpha_n}{\rightarrow} E_n$. As $P_0 \in \mathcal{P}$ and $P_0 \overset{\alpha_0}{\rightarrow} E_1$, there exists $(P_0, E_0) \in \Gamma$ such that $E_0 \overset{\alpha_0}{\rightarrow} E_1$. Let $\Lambda' = ResAction(E) \cup ResAction(P)$ for all $(P, E) \in \Gamma$. According to Proposition 1, one has $\Phi(E_0) \rightarrow^*_{\mathcal{R}_{sr}^{\Lambda\Lambda'}} Sys(\alpha_0, \Phi(E_1))$. In addition, one has $\Phi(E_0) \rightarrow^*_{\mathcal{R}_{sr}^{\Lambda\Lambda'}} Sys(\alpha_0, \Phi(E_1)) \rightarrow^*_{\mathcal{R}_{sr}^{\Lambda\Lambda'}} \ldots \rightarrow^*_{\mathcal{R}_{sr}^{\Lambda\Lambda'}} Sys(\alpha_0, Sys(\ldots, Sys(\alpha_{n-1}, \Phi(E_{n-1}))\ldots)) \rightarrow^*_{\mathcal{R}_{sr}^{\Lambda\Lambda'}} Sys(\alpha_0, Sys(\ldots, Sys(\alpha_n, \Phi(E_n))\ldots))$.

It remains to prove that $\Phi(P_0) \rightarrow^*_{\mathcal{R}_S} \Phi(E_0)$. By definition, there exists a rewriting rule $\Phi(P_0) \rightarrow \Phi(E_0) \in \mathcal{R}^{\Gamma}_{Con}$. From this we obtain that $\Phi(P_0) \rightarrow^*_{\mathcal{R}_S} \Phi(E_0)$. Finally we can conclude $\Phi(P_0) \rightarrow^*_{\mathcal{R}_S} \Phi(E_0) \rightarrow^*_{\mathcal{R}_{sr}^{\Lambda\Lambda'}} Sys(\alpha_0, \Phi(E_1)) \rightarrow^*_{\mathcal{R}_{sr}^{\Lambda\Lambda'}} \ldots \rightarrow^*_{\mathcal{R}_{sr}^{\Lambda\Lambda'}} \Psi((\alpha_0 \ldots \alpha_{n-1}, E_{n-1})) \rightarrow^*_{\mathcal{R}_{sr}^{\Lambda\Lambda'}} \Psi(d)$ which completes the proof. □

4 The Alternating Bit Protocol Verification

This section shows that the Alternating Bit Protocol (ABP) CCS program is not able to perform a specific succession of actions represented by a set of derivatives.

Given the TRS $\mathcal{R}$ and the language L, corresponding to the ABP CCS program, the construction of the set $\mathcal{R}^*(L)$ is not possible, but an over-approximation $\mathcal{K}$ of this reachability set can be computed [14, 19]. Because of the over-approximation, we can only deduce that a language L_p is not reachable ($\mathcal{R}^*(L) \cap L_p = \emptyset$) if $\mathcal{K} \cap L_p = \emptyset$. In our case, the language $\mathcal{K}$ recognises an over-approximation of all possible derivatives of the ABP CCS program, and the language L_p recognises a set of derivatives we do not want to be in $\mathcal{K}$. Then, if the intersection between $\mathcal{K}$ and L_p is empty, we can conclude that the set of all possible derivatives of the ABP CCS Program does not contain derivatives represented by L_p.

4.1 The Alternating Bit Protocol description

The ABP is a protocol made to ensure the successful transmission of messages through a channel which may lose or duplicate data. More precisely, the ABP is composed of a Sender and a Receiver communicating via two channels (which may lose or duplicate messages) called Trans and Ack. The Sender sends a message with a bit b through the Trans channel, and sends it one or more times until the Receiver sends an acknowledgment with the bit b through the Ack channel. After the reception of this message by the Sender, it sends (once or more) another message with the bit $b-1$ (also written $\hat{b}$) until it receives an acknowledgment with the bit $\hat{b}$, and so on.

4.2 Modeling the ABP

The CCS specification of ABP used in this article can be found in [22], and is represented by the CCS program $ABP = (\Lambda, \Gamma, AB)$ where :

- the set $\Lambda = \{accept, ack(b), deliver, reply(b), send(b), trans(b)\}$;
- the set Γ is composed of rules in Figures 3 and 4, where for each transition $A \xrightarrow{\alpha} B$ we have $(A, \alpha.B) \in \Gamma$, with A, $B \in \mathcal{E}$ and $\alpha \in \Lambda$.

The corresponding TRS $\mathcal{R}_{ABP}$ and tree language L_{ABP} is defined according to definition in Section 3. But also, we have to add rewriting rules to handle sequences of bits.

$$\begin{array}{lcl}
Send(b) & \overset{def}{=} & \overline{send(b)}.Sending(b) \\
Sending(b) & \overset{def}{=} & \tau.Send(b) + ack(b).Accept(\hat{b}) + ack(\hat{b}).Sending(b) \\
Accept(b) & \overset{def}{=} & accept.Send(b) \\
Reply(b) & \overset{def}{=} & \overline{reply(b)}.Replying(b) \\
Replying(b) & \overset{def}{=} & \tau.Reply(b) + trans(\hat{b}).Deliver(\hat{b}) + trans(b).Replying(b) \\
Deliver(b) & \overset{def}{=} & \overline{deliver}.Reply(b) \\
AB & \overset{def}{=} & Accept(\hat{b}) \parallel Trans(\varepsilon) \parallel Ack(\varepsilon) \parallel Reply(b)
\end{array}$$

Fig. 3. System equations for ABP

$$\begin{array}{lcllcl}
Ack(bs) & \xrightarrow{\overline{ack(b)}} & Ack(s) & Trans(sb) & \xrightarrow{\overline{trans(b)}} & Trans(s) \\
Ack(s) & \xrightarrow{reply(b)} & Ack(sb) & Trans(s) & \xrightarrow{send(b)} & Trans(bs) \\
Ack(sbt) & \xrightarrow{\tau} & Ack(st) & Trans(tbs) & \xrightarrow{\tau} & Trans(ts) \\
Ack(sbt) & \xrightarrow{\tau} & Ack(sbbt) & Trans(tbs) & \xrightarrow{\tau} & Trans(tbbs)
\end{array}$$
$$\text{where } s, t \in \{0,1\}^* \text{ and } b \in \{0,1\}.$$

Fig. 4. System transitions for ABP

4.3 Verifying the ABP

In this section we will show how to verify, using the tool Tomedtimbuk [2], that the ABP can not send a message with the bit b after an acknowledgment with the bit b.

We proceed as follows: first, the property is modeled using patterns. Then, we have to find an abstraction function suitable for our analysis to ensure termination of the completion. Finally we use the Tomedtimuk tool to prove automatically that the ABP can not acknowledge and then send a message with the same bit.

The property modelisation is very simple, one can use the following patterns:

Sys(s,Sys(bar(send(b)),Sys(ack(b),Sys(bar(send(b)),Sys(ss,p)))))
Sys(s,Sys(bar(send(inv(b))),Sys(ack(inv(b)),Sys(bar(send(inv(b))),Sys(ss,p)))))
Sys(s,Sys(bar(send(b)),Sys(ack(b),Sys(bar(send(b)),p))))
Sys(s,Sys(bar(send(inv(b))),Sys(ack(x,y,inv(b())),Sys(bar(send(inv(b))),p))))
Sys(bar(send(b)),Sys(ack0(b),Sys(bar(send(b)),p)))
Sys(bar(send(inv(b))),Sys(ack(inv(b)),Sys(bar(send(inv(b))),p)))

where s, ss and p can be anything in $\mathcal{T}(\mathcal{F}_{CCS})$. Those six patterns represent all possible derivatives of ABP where an action $\overline{send(b)}$ succeeds to an action $ack(b)$ (with $b \in \{0, 1\}$).

Concerning the abstraction function, the main idea is to abstract each action involved in the property in one state, and all other actions into one other state. Abstraction rules for the ABP actions, process names and bits are: $[x \to y] \to [b \to q_b, inv(q_b) \to q_b, send(q_b) \to q_{send}, bar(q_{send}) \to q_{\overline{send}}, ack(q_b) \to q_{ack}, accept(q_b) \to q_{rem}, reply(q_b) \to q_{rem}, trans(q_b) \to q_{rem}, deliver(q_b) \to q_{rem}, nil \to q_{rem}, bar(q_{rem}) \to q_{rem}, Send(q_b) \to q_{rem}, Sending(q_b) \to q_{rem}, Accept(q_b) \to q_{rem}, Reply(q_b) \to q_{rem}, Replying(q_b) \to q_{rem}, Deliver(q_b) \to q_{rem}]$. The $[x \to y]$ part matches any new transition which need to be normalized. The rules $b \to q_b$ and $inv(q_b) \to q_b$ merge all bit into one state q_b. The rules $send(q_b) \to q_{send}$, $bar(q_{send}) \to q_{\overline{send}}$ and $ack(q_b) \to q_{ack}$ merge all actions $\overline{send(b)}$ and $ack(b)$ into, respectively, states $q_{\overline{send}}$ and q_{ack}. All others actions and process names are merged into one state q_{rem}, according to the fact that those last actions and process names are not referenceed by the property.

Finally, given the initial automaton recognizing L_{ABP}, the TRS $\mathcal{R}_{ABP}$, the property and the abstraction function, the Tomedtimbuk tool computes a fixpoint automaton $\mathcal{A}_k$ over-approximating the set of all possibles derivatives of ABP. The intersection between $L(\mathcal{A}_k)$ and the property is empty, so we can conclude the ABP can not do an action $\overline{send(b)}$ after an action $ack(b)$, according to the following Proposition 3.

Proposition 3. *Let $S = (\Lambda, \Gamma, P_0)$ be a CCS program, let L_p be the language representing a derivative $(\alpha_0 \dots \alpha_n, E)$ with $\alpha_0, \dots, \alpha_n \in \Lambda$ and $E \in \mathcal{E}$ such that $L_p = \{\Psi((\alpha_0 \dots \alpha_n, E))\}$. One has: $\mathcal{R}^*_S(L_S) \cap L_p = \emptyset$ if and only if $(\alpha_0 \dots \alpha_n, E)$ is not a derivative of P_0.*

Proof. We have to prove that $(\mathcal{R}^*_S(L_S) \cap L_p = \emptyset) \Leftrightarrow ((\alpha_0 \dots \alpha_n, E) \notin Deriv(P_0))$. The proof is divided into two parts: we will prove that $(\mathcal{R}^*_S(L_S) \cap L_p = \emptyset) \Rightarrow ((\alpha_0 \dots \alpha_n, E) \notin Deriv(P_0))$ (1), and then that $((\alpha_0 \dots \alpha_n, E) \notin Deriv(P_0)) \Rightarrow (\mathcal{R}^*_S(L_S) \cap L_p = \emptyset)$ (2).

(1) By contraposition of Proposition 2, one has $(\mathcal{R}^*_S(L_S) \cap L_p = \emptyset) \Rightarrow ((\alpha_0 \dots \alpha_n, E) \notin Deriv(P_0))$.

(2) Suppose that $((\alpha_0 \dots \alpha_n, E) \notin Deriv(P_0)) \Rightarrow (\mathcal{R}^*_S(L_S) \cap L_p = \emptyset)$ is false, we have the following hypothesis : $((\alpha_0 \dots \alpha_n, E) \notin Deriv(P_0)) \wedge (\mathcal{R}^*_S(L_S) \cap L_p \neq \emptyset)$. If $\mathcal{R}^*_S(L_S) \cap L_p \neq \emptyset$ then $\Psi(P_0) \to^*_{\mathcal{R}_S} \Psi((\alpha_0 \dots \alpha_n, E))$. We will prove that $(\alpha_0 \dots \alpha_n, E) \in Deriv(P_0)$ which is in contradiction with the hypothesis. In order to succeed we have to prove the Lemma 2.

Lemma 2. *Let E and E' be two CCS expressions, let α be an action name and let $A_E = Action(E)$ and $A'_E = ResAction(E)$. If $\Phi(E) \rightarrow^*_{\mathcal{R}_{sr}^{A_E A'_E}} \psi((\alpha, E'))$ then $E \xrightarrow{\alpha} E'$.*

Proof. We have to show that E' can be built according to the inference rules of Figure 1 from E.

As $\mathcal{P}os_{\{Sys\}}(\Phi(E)) = \emptyset$, one has $\Phi(E) \rightarrow_{\rho_1} t_1 \rightarrow^*_{\mathcal{R}_{sr}^{A_E A'_E}} \Psi((\alpha, E'))$ such that there exists $p \in \mathcal{P}os(t_1)$ where $\Phi(E)|_p = Pre(\alpha, \Phi(E_1))$ and $t_1|_p = Sys(\alpha, \Phi(E_1))$. If $p = \epsilon$ then one has $\Phi(E) \equiv \Phi(\alpha.E')$, and we can deduce that $E \xrightarrow{\alpha} E'$ according to the **Act** inference rule. Else, one has $\alpha.E_1 \xrightarrow{\alpha} E_1$.

Then, we argue by cases of the term at a position p', such that $p = p'.1$ or $p = p'.2$:

Case 1: $t_1|_p = Sum(Sys(\alpha, \Phi(E_1)), t_2)$ (resp. $t_1|_p = Sum(t_2, Sys(\alpha, \Phi(E_1))))$
According to rewriting rule ρ_2 (resp. ρ_3), it follows that $t_1|_p \rightarrow_{\rho_2} Sys(\alpha, \Phi(E_1))$ (resp. $t_1|_p \rightarrow_{\rho_3} Sys(\alpha, \Phi(E_1))$). As $\alpha.E_1 \xrightarrow{\alpha} E_1$, hence $\alpha.E_1 + E_2 \xrightarrow{\alpha} E_1$ (where $t_2 = \Phi(E_2)$), according to $\mathbf{Sum_1}$ and $\mathbf{Sum_2}$ inference rules.

Case 2: $t_1|_p = Com(Sys(\alpha, \Phi(E_1)), t_2)$ (resp. $t_1|_p = Com(t_2, Sys(\alpha, \Phi(E_1))))$
Similar to Case 1.

Case 3: $t_1|_{p'.1} = Res(Sys(\alpha, \Phi(E_1)), \ell)$, with ℓ an action name. According to rewriting rule ρ_8, it follows that $t_1|_{p'.1} \rightarrow_{\rho_8} Sys(\alpha, Res(\Phi(E_1), \ell))$. As $\alpha.E_1 \xrightarrow{\alpha} E_1$, hence $\alpha.E_1 \setminus \ell \xrightarrow{\alpha} E_1 \setminus \ell$ according to **Res** inference rule.

$\square$

Consequently to Lemma 2, one has $(\alpha_0 \ldots \alpha_n, E) \in Deriv(P_0)$ if $\Psi(P_0) \rightarrow^*_{\mathcal{R}_S} \Psi((\alpha_0 \ldots \alpha_n, E))$. This contradicts the hypothesis and proves (2).

Finally, from proofs of (1) and (2), one can conclude that $(\mathcal{R}^*_S(L_S) \cap L_p = \emptyset) \Leftrightarrow ((\alpha_0 \ldots \alpha_n, E) \notin Deriv(P_0))$. $\square$

5 Hardware components verification

In this section we are going to verify properties over two hardware components specified with CCS [25].

5.1 The *Lockable* component

The *Lockable* component is composed of two elements:

- one element with three inputs a, b and $free$, and one output z ;
- one element with two inputs $lock$ and $unlock$, and one output $free$.

We call *Lockable* the component including the parallelization of this two elements, while restricting the $free$ action. *Lockable* allows the lock and unlock effects on z output. Indeed, there is no output z when the $lock$ action is done, until the $unlock$ is done. And there is an output z only after a silent action. The CCS program corresponding to the *Lockable* component is defined in Figure 5, where LC is the initial process.

The property we want to verify is : *Is* Lockable *able to realize an action lock followed by an action $\overline{z}$?* To answer this question, we proceed in a same way that for ABP. A TRS $\mathcal{R}_{LC}$ and a tree automaton A_{LC} are constructed from the *Lockable* CCS program, the abstraction function is written following the principle used for ABP. Finally, a tree automaton A_p is build to recognize derivative of the form$(\alpha^*(lock\overline{z})\alpha^*, E)$ (where α is an action and E a CCS expression). Using Tomedtimbuk tools, one has $\mathcal{R}^*_{LC}(L(A_{LC})) \cap L(A_p) = \emptyset$, so we can answer *No* to the question.

$$\begin{aligned} LockC &\overset{def}{=} (a.b + b.a).free.\overline{z}.C \\ Lock &\overset{def}{=} \overline{free}.Lock + lock.unlock.Lock \\ LC &\overset{def}{=} (LockC \parallel Lock) \setminus \{free\} \end{aligned}$$

Fig. 5. Equations for the *Lockable* component

$$\begin{aligned} U_1 &\overset{def}{=} r_1.gS.\overline{g_1}.d_1.\overline{pS}.\overline{a_1}.U_1 \\ U_2 &\overset{def}{=} r_2.gS.\overline{g_2}.d_2.\overline{pS}.\overline{a_2}.U_2 \\ S &\overset{def}{=} (\overline{gS}.pS.S) \setminus \{gS, pS\} \end{aligned}$$

Fig. 6. Equations for the *RGDA* component

5.2 The *RGDA* component

The *RGDA* component (*Request Grant Done Acknowledgment*) is a component handling two users access to a critical section. It ensures that one user access to this section at a time. The CCS program corresponding to this component is composed by equations of Figure 6, where S is the initial process, U_1 and U_2 are users.

The property we want to verify is : *Is* RGDA *able to realize the actions* $\overline{g_1}$ *and* $\overline{g_2}$ *successively ?* As the *Lockable* component, a TRS $\mathcal{R}_{RGDA}$, a tree automaton A_{RGDA}, an abstraction function and a tree automaton A_p are defined. The tree automaton A_p recognizes derivatives of the form $(\alpha^*(\overline{g_1 g_2})\alpha^*, E)$ (where α is an action and E a CCS expression). With the help of Tomedtimbuk, one can compute that $\mathcal{R}^*_{RGDA}(L(A_{RGDA})) \cap L(A_p) = \emptyset$, so we can answer *No* to the question.

6 Conclusion and Related works

The paper describes a method of encoding CCS specifications into a TRS and a tree automaton. Using the completion algorithm, one can compute an over-approximation of reachable derivatives K, modulo bisimulation. It means that the set K do not contain CCS expressions bisimilar to CCS expressions of derivatives in K. Then, it is possible to semi-decide if derivatives, encoded into a tree automaton, are reachable or not. So, bisimilar CCS expressions have to belong to those derivatives in order to get a correct answer by the semi-decision procedure.

For other existing process algebras like CSP, BPP, BPA, PA, SDL, LOTOS, ..., sharing syntax and semantics elements with CCS, it could be insteresting to adapt the over-approximation rewriting to those process algebras.

Furthermore, to build this over-approximation, a pertinent abstraction function is needed i.e. the abstraction function allows the termination of the over-approximation computation without introducing spurious counter-examples which prevent the verification to conclude. In sections 4 and 5, abstraction functions can easily be generated automatically according to a property. However, it is not always possible. Note that the automatic generation of abstraction function has already been used for the protocol verification [3].

Related Works It exists some tools made for the verification of CCS programs, as the Edinburgh Concurrency Workbench [23], the Concurrency Workbench North Carolina [9] and XMC [24], which are finite-state model-checkers while our technique deals with infinite-state systems. Also, verification of CCS programs can be done with Maude [27], where the CCS semantics is represented by conditional rewriting rules, while our method uses rewriting rules.

In [15], authors present a semi-decision procedure allowing verification of ACTL properties [11] (action based temporal properties) for infinite states systems. The method presented in this article does not handle CTL properties, but allows to verify reachability properties, based on actions and on CCS expressions. This property is represented by a tree automaton, instead of a temporal property. This can be similar to the proof by bisimulation, where behaviors of two CCS expressions are compared from the action point of view.

References

1. Baader, F., Nipkow, T.: Term Rewriting and All That. Cambridge University Press (1998)
2. Balland, E., Boichut, Y., Genet, T., Moreau, P.E.: Towards an efficient implementation of tree automata completion. In: AMAST. pp. 67–82 (2008)
3. Boichut, Y.: Approximations pour la vérification automatique de protocoles de sécurité. Thèse de doctorat, Laboratoire Informatique de l'université de Franche-Comté, Université de Franche-Comté, Besançon, France (2006), http://www.irisa.fr/lande/boichut/publications.html
4. Boichut, Y., Courbis, R., Héam, P.C., Kouchnarenko, O.: Finer is better: Abstraction refinement for rewriting approximations. In: Rewriting Techniques and Application, RTA'08. Lecture Notes in Computer Science, vol. 5117, pp. 48–62. Springer (2008)
5. Boichut, Y., Genet, T., Jensen, T., Roux, L.L.: Rewriting approximations for fast prototyping of static analyzers. In: Rewriting Techniques and Applications, RTA'07. pp. 48–62. Lecture Notes in Computer Science 4533, Springer (2007)
6. Boichut, Y., Héam, P.C., Kouchnarenko, O.: Approximation-based tree regular model-checking. Nordic Journal of Computing (2009), to appear
7. Busi, N., Gabbrielli, M., Zavattaro, G.: Replication vs. recursive definitions in channel based calculi. In: ICALP. pp. 133–144 (2003)
8. Clarke, E.M.: Counterexample-guided abstraction refinement. In: TIME-ICTL. p. 7. IEEE Computer Society (2003)
9. Cleaveland, R., Sims, S.: The ncsu concurrency workbench. In: CAV '96: Proceedings of the 8th International Conference on Computer Aided Verification. pp. 394–397. Springer-Verlag, London, UK (1996)
10. Comon, H., Dauchet, M., Gilleron, R., Jacquemard, F., Lugiez, D., Tison, S., Tommasi, M.: Tree Automata Techniques and Applications (2002), available at *http://www.grappa.univ-lille3.fr/tata/*
11. De Nicola, R., Vaandrager, F.: Action versus state based logics for transition systems. In: Proceedings of the LITP spring school on theoretical computer science on Semantics of systems of concurrent processes. pp. 407–419. Springer-Verlag New York, Inc., New York, NY, USA (1990)
12. Dershowitz, N., Jouannaud, J.P.: Handbook of Theoretical Computer Science, vol. B, chap. 6: Rewrite Systems, pp. 244–320. Elsevier Science Publishers B. V (1990)

13. Feuillade, G., Genet, T., VietTriemTong, V.: Reachability analysis over term rewriting systems. Journal on Automated Reasoning 33 (3-4) (2004)
14. Feuillade, G., Genet, T., Tong, V.V.T.: Reachability analysis over term rewriting systems. Journal of Automated Reasoning 33(3-4), 341–383 (2004)
15. Francesco, N.D., Fantechi, A., Gnesi, S., Inverardi, P.: Model checking of non-finite state processes by finite approximations. In: In Proc. Tools and Algorithms for the Construction and Analysis of Systems (TACAS'95), Lecture Notes in Computer Science 1019. pp. 195–215. Springer (1994)
16. Genet, T., Klay, F.: Rewriting for Cryptographic Protocol Verification. In: Conference on Automated Deduction, CADE'00, Lecture Notes in Computer Science, vol. 1831, pp. 271–290. Springer-Verlag (2000)
17. Gilleron, R., Tison, S.: Regular tree languages and rewrite systems. Fundamenta Informatica 24(1/2), 157–174 (1995)
18. Gyenizse, P., Vágvölgyi, S.: Linear Generalized Semi-Monadic Rewrite Systems Effectively Preserve Recognizability. Theoretical Computer Science 194(1-2), 87–122 (1998)
19. Jacquemard, F.: Decidable approximations of term rewriting systems. In: Rewriting Techniques and Applications, RTA'96. vol. 1103, pp. 362–376. Springer Verlag (1996)
20. Lamport, L.: A temporal logic of actions. ACM Transactions On Programming Languages And Systems, TOPLAS 16(3), 872–923 (May 1994)
21. Manna, Z., Pnueli, A.: The Temporal Logic of Reactive and Concurrent Systems: Specification. SV (1992)
22. Milner, R.: Communication and Concurrency. Prentice Hall (1989)
23. R., C., J., P., B., S.: The concurrency workbench: A semantics based tool for the verification of concurrent systems. ACM Transactions on Programming Languages and Systems 15 (1994)
24. Ramakrishna, Y.S., Ramakrishnan, C.R., Ramakrishnan, I.V., Smolka, S.A., Swift, T., Warren, D.S.: Efficient model checking using tabled resolution. In: Computer Aided Verification (CAV '97). Springer-Verlag (1997)
25. Stevens, K., Aldwinckle, J., Birtwistle, G., Liu, Y.: Designing parallel specifications in ccs. In: In Proceedings of Canadian Conference on Electrical and Computer Engineering. pp. 983–986 (1993)
26. Takai, T., Kaji, Y., Seki, H.: Right-linear finite-path overlapping term rewriting systems effectively preserve recognizability. In: proceedings of RTA. Lecture Notes in Computer Science, vol. 1833. Springer-Verlag (2000)
27. Verdejo, A., Martí-Oliet, N.: Two case studies of semantics execution in Maude: CCS and LOTOS. Formal Methods in System Design 27, 113–172 (2005)