Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science 204 (2008) 83–98
Electronic Notes in Theoretical Computer Science
www.elsevier.com/locate/entcs

Regular Strategies as Proof Tactics for CIRC

Dorel Lucanu[1],[3]

Faculty of Computer Science
Alexandru Ioan Cuza University
Iaşi, Romania

Grigore Roşu[2],[4]

Department of Computer Science
University of Illinois
Urbana-Champaign, USA

Gheorghe Grigoraş[1],[5]

Faculty of Computer Science
Alexandru Ioan Cuza University
Iaşi, Romania

Abstract

CIRC is an automated circular coinductive prover implemented as an extension of Maude. The main engine of CIRC consists of a set of rewriting rules implementing the circularity principle. The power of the prover can be increased by adding new capabilities implemented also by rewriting rules. In this paper we prove the correctness of the coinductive prover and show how rewriting strategies, expressed as regular expressions, can be used for specifying proof tactics for CIRC. We illustrate the strength of the method by defining a proof tactic combining the circular coinduction with a particular form of simplification for proving the equivalence of context-free processes.

Keywords: Behavioral equivalence, circular coinduction, regular strategies, proof tactics

1 Introduction

A behavioral algebraic specification is an algebraic specification where the sorts are split into *visible* (or *observational*) for data and *hidden* for states, and the

[1] Partially supported by CEEX grant 47/2005 and CNCSIS grant 1162/2007.

[2] Partially supported by NSF grants CCF-0448501 and CNS-0509321.

[3] Email: dlucanu@info.uaic.ro

[4] Email: grosu@cs.uiuc.edu

[5] Email: grigoras@info.uaic.ro

doi:10.1016/j.entcs.2008.03.055

equality is behavioral. Two states are *behaviorally equivalent* if and only if they *appear* to be the same under any visible *experiment*. The experiments are derived operations of visible result sort, and defined only with *behavioral operations* (*derivatives*). A canonical example is that of (infinite) streams. The sort *Stream* of streams is hidden, the sort of elements, e.g. *Int*, is visible, and the behavioral operations are the head `hd(*:Stream)` and the tail `tl(*:Stream)`. The special variable `*:Stream` marks the place of the state parameter. Examples of experiments are `hd(*:Stream)`, `hd(tl(*:Stream))`, `hd(tl(tl(*:Stream)))`, and so on. The behavioral equivalence (equality) over streams is given by $S \equiv S'$ iff $\mathtt{hd}(S) = \mathtt{hd}(S')$ and $\mathtt{tl}(S) \equiv \mathtt{tl}(S')$. If this is the case, then all the above experiments return the same visible data value for S and S', respectively. Note that the behavioral equivalence coincides with the equality over the visible data. Therefore the behavioral equivalence over streams requires as the values returned by the head operation to be equal.

The main issue in behavioral specification theory is how to prove that two states are behavioral equivalent. The coalgebraic *bisimulation* (see, e.g., Jacobs and Rutten [10]) as well as Hennicker's *context induction* [7] are both sound proof techniques for behavioral equivalence. Unfortunately, both of them need human intervention: coinduction to pick a "good" bisimulation relation, and context induction to device and prove auxiliary lemmas. *Circular coinduction* [4,14] is an automatic proof technique for behavioral equivalence, supported in BOBJ. By circular coinduction one can prove, for instance, the equality $\mathit{zip}(\mathit{zeros}, \mathit{ones}) = \mathit{blink}$ on streams as follows (zeros is the stream 0^ω, ones is 1^ω, blink is $(01)^\omega$, zip merges two streams):

(i) check that the two streams have the same head, 0;

(ii) take the tail of the two streams and generate the new goal $\mathit{zip}(\mathit{ones}, \mathit{zeros}) = \mathit{1\ blink}$; this becomes the next task;

(iii) check that the two new streams have the same head, 1;

(iv) take the tail of the two new streams; after simplification one gets the new goal $\mathit{zip}(\mathit{zeros}, \mathit{ones}) = \mathit{blink}$, which is nothing but the original proof task;

(v) conclude that $\mathit{zip}(\mathit{zeros}, \mathit{ones}) = \mathit{blink}$ holds.

The intuition for the above "proof" is that the two streams have been exhaustively tried to be distinguished by iteratively checking their heads and taking their tails. Ending up in circles (we obtained the same new proof task as the original one) means that the two streams are indistinguishable, i.e., they are equal.

Circular coinduction can be explained and proved to be correct by reducing it to either bisimulation or context induction: it iteratively constructs a bisimulation, but it also discovers all lemmas needed by a context induction proof. Since the behavioral equivalence problem is Π_2^0-complete [14] (it is so, even in the context of just streams [15]), there is *no* algorithm or proof system that is complete for behavioral equality in general, as well as *no* algorithm or proof system that is complete for inequality of streams. Therefore, the best we can do is to focus our efforts on exploring heuristics or deduction rules to prove or disprove equalities of streams that *work well on examples of interest* rather than in general.

BOBJ [4,14] was the first system supporting circular coinduction. Hausmann, Mossakowski, and Schröder [6] also developed circular coinductive techniques and tactics in the context of CoCASL. CIRC [12,11] is an automated circular coinductive prover implemented in Full Maude [3] as a behavioral extension of the Maude system [2], making heavy use of meta-level and reflection capabilities of rewriting logic. Maude is by now a very mature system, with many uses, a high-performance rewrite engine and a broad spectrum of analysis tools. Maude's current meta-level capabilities where not available when we developed the BOBJ system; consequently, BOBJ was a heavy system, with rather poor parsing and performance. By allowing the entire Maude system visible to the user, CIRC inherits all Maude's uses, performance and analysis tools. CIRC implements the *circularity principle*, which generalizes circular coinductive deduction [5] and can be expressed as follows. Assume that each equation of interest (to be proved) e admits a *frozen form* $\mathit{fr}(e)$ and a set of derived equations, its *derivatives*, $\mathit{Der}(e)$. The circularity principle says that if from hypotheses $\mathcal{H}$ together with $\mathit{fr}(e)$ we can deduce $\mathit{Der}(e)$, then e is a consequence of $\mathcal{H}$. When $\mathit{fr}(e)$ freezes the equation at the top, as in [5], the circularity principle becomes circular coinduction. Interestingly, when the equation is frozen at the bottom on a variable, then it becomes a structural induction (on that variable) derivation rule. This way, CIRC supports both coinduction and induction as projections of a more general principle. This paper makes two main contributions. First, we prove the correctness of CIRC's coinductive capabilities. Second, we define and implement a strategy language for CIRC.

Version 1.2 of CIRC [11] provides automatic proving support for both coinduction and induction, but not for combinations of them. A combination of the two techniques is possible only in an assisted way. In practice, there are many cases when the two must be combined with other techniques. CIRC's proof capabilities are implemented using rewriting rules. A proof tactic constraints the application of rules according to a given aim. To express the proof tactics, we use a strategy language based on regular expressions over rule labels. Regular expressions can be behaviorally specified; CIRC includes a copy of this specification to handle the proof tactics. This new feature is included CIRC v1.3. We illustrate it by defining a proof tactic able to prove the equivalence of context-free processes.

The paper is structured as follows. Section 2 introduces behavioral specification and presents the coinductive capabilities of the CIRC, and shows the correctness of implementation of the circular coinduction as rewriting rules. We use the regular expressions as an example of behavioral specification and we show that CIRC together with this specification supplies a fully automatic decision procedure for the equivalence of regular expressions. Section 3 introduces the regular strategies and defines proof tactics in terms of regular strategies. A proof tactic that combines coinduction with simplification of goals is presented. Section 4 shows how context-free processes can be behaviorally specified and presents a proof technique, based on coinduction and simplification, for proving the equivalence of context-free processes. The paper ends with concluding remarks.

2 Behavioral Algebraic Specifications

Let Σ be an algebraic signature consisting of a set $\mathit{Sorts}(\Sigma)$ of *sorts* and an $S^* \times S$-indexed set $\mathit{Op}(\Sigma) = (\mathit{Op}(\Sigma)_{w,s} \mid w \in S^*, s \in S)$ of *operations*. We assume $\mathit{Sorts}(\Sigma) = V \cup H$, where V is a subset of *visible sorts*, H is a subset of *hidden sorts*, and $V \cap H = \emptyset$. For each $f \in \mathit{Op}(\Sigma)_{w,s}$, we use the following notations: $\mathit{arity}(f) = w$, $\mathit{sort}(f) = s$, and $\mathit{type}(f) = (w, s)$ (also written $w \to s$). Let $\mathcal{X}$ be a fixed S-indexed set of *variables*. $T_\Sigma(\mathcal{X})$ is the Σ-algebra of terms with variables in $\mathcal{X}$. Let $\mathit{Var}(t)$ denote the set of variables occurring in term t. A Σ-*behavioral operation* for hidden sort $h \in H$, also called a *derivative*, is a term $\delta \in T_\Sigma(\mathcal{X} \cup \{*{:}h\})$, where $*{:}h$ is a special variable of sort h. The sort of δ is called δ's *result sort*. A Σ-*equation* is a sentence $(\forall X)\ t = t'$ `if` c, where t and t' are Σ-terms over variables $X \subseteq \mathcal{X}$ having the same result sort, and c is the *condition* of the equation consisting of a finite set of pairs (t_i, t'_i) of terms over variables X and with *visible* result sorts; we only consider equations with finitely many visible conditions in this paper. A condition c is also written as $t_1 = t'_1 \wedge \cdots \wedge t_n = t'_n$. If the sorts of t and t' are hidden, then the equation is called *behavioral*. If the condition c is empty, then we get an unconditional Σ-equation and write it as $(\forall X)\ t = t'$.

A *behavioral specification* is a triple $\mathcal{B} = (\Sigma, \Delta, E)$, where Σ is an algebraic signature, Δ is a set of behavioral operations, and E is a set of Σ-equations. Let $=_E$ denote the standard equational derivability congruence over $T_\Sigma(\mathcal{X})$ with the equations E (the E-equality), and let $T_{\Sigma,E}(\mathcal{X})$ denote the quotient $T_\Sigma(\mathcal{X})/{=_E}$. A Δ-*experiment* for the hidden sort $h \in H$ is inductively defined as follows: each behavioral operation for the hidden sort $h \in H$ with visible result sort is a Δ-experiment for h; if γ is a Δ-experiment for h' and δ a behavioral operation for h with result sort h', then $\gamma[\delta/*{:}h']$ is a Δ-experiment for h.

The notion of *behavioral equivalence* is an inherently semantic one: there is a behavioral equivalence relation on each model which can be defined as "indistinguishability under experiments". For technical simplicity, we here prefer to avoid introducing models, so we give an alternative, proof theoretic definition. The Δ-*behavioral equivalence* $\equiv_\Delta$ over $T_\Sigma(\mathcal{X})$ is the E-equality (closure under equational deduction with E) generated by the following: for each visible sort $v \in V$, $\equiv_{\Delta,v}$ is $=_{E,v}$; if $h \in H$ and $t, t' \in T_\Sigma(\mathcal{X})_h$ then $t \equiv_\Delta t'$ iff $\gamma[t/*{:}h] =_E \gamma[t'/*{:}h]$ for each Δ-experiment γ for h. (We here therefore assume that operations are behaviorally congruent, i.e., $f(t_1, \ldots, t_n) \equiv_\Delta f(t'_1, \ldots, t'_n)$ whenever $t_i \equiv_\Delta t'_i$ for $i = 1, \ldots, n$.) We often write $\gamma[t]$ for $\gamma[t/*{:}h]$. Note that the relation $\equiv_\Delta$ is not algorithmic, because one needs an infinite number of experiments to decide it; the basis for the Π^0_2 result in [14] is the observation that *for each* experiment *there is* some proof (thanks to the completeness of equational deduction). Moreover, the E-equality is undecidable for the general case. $\mathcal{B}$ *behaviorally entails* the behavioral Σ-equation e, written $\mathcal{B} \models\!\!\equiv e$, iff

(i) either e is of the form $(\forall X)\ t = t'$ (the condition is empty) and $t \equiv_\Delta t'$,

(ii) or e is of the form $(\forall X)\ t = t'$ `if` c with c consisting of $t_1 = t'_1 \wedge \cdots \wedge t_n = t'_n$ and $\mathcal{B}(c) \models\!\!\equiv (\forall X)\ t = t'$, where $\mathcal{B}(c)$ is the behavioral specification $(\Sigma \cup X, \Delta, E \cup$

$\{(\forall\emptyset)t_i = t'_i \mid i = 1,\ldots,n\}$) (the variables in X are added as constants, then the condition is added as hypothesis).

The *equational entailment* is defined in a similar way: $\mathcal{B} \models e$, iff

(i) either e is of the form $(\forall X)\ t = t'$ and $t =_E t'$,

(ii) or e is of the form $(\forall X)\ t = t'$ `if` c and $\mathcal{B}(c) \models (\forall\emptyset)\ t = t'$.

Example: Regular Expressions.

Regular expressions (RE) are finite presentations for possibly infinite languages. The languages denoted by REs are inductively defined using the operations union, concatenation, and Kleene closure. Formally, let *Alph* be an alphabet; the set of *regular expressions* over *Alph* is given by the grammar

$$R ::= \varepsilon \mid \emptyset \mid a \mid R_1 + R_2 \mid R_1 \# R_2 \mid R^*$$

where a ranges over *Alph*. The *language* denoted by a RE R is defined as follows: $\mathcal{L}(\varepsilon) = \{\varepsilon\}$, $\mathcal{L}(\emptyset) = \emptyset$, $\mathcal{L}(a) = \{a\}$, $\mathcal{L}(R_1 + R_2) = \mathcal{L}(R_1) \cup \mathcal{L}(R_2)$, $\mathcal{L}(R_1 \# R_2) = \{ww' \mid w \in \mathcal{L}(R_1), w' \in \mathcal{L}(R_2)\}$, and $\mathcal{L}(R^*) = (\mathcal{L}(R))^*$. Two REs are *equivalent* iff they denote the same language. In [16,17] a behavioral specification for (Extended) REs is given, where the behavioral equivalence coincides with the RE equivalence. The behavioral operations (derivatives) defining behavioral equivalence are `epsIn_` (testing the membership of ε to a RE) and $\_\{\_\}$, which takes a RE R and a letter a and returns an expression $R\{a\}$ characterized by $L(R\{a\}) = \{w \mid aw \in L(R)\}$. $R\{a\}$ is semantically equivalent to an RE because regular languages are closed under left-quotient by an arbitrary language. Here is a Maude description of REs:

```
(th RE is including BOOL .
  sort Ere .                               --- extended regular expressions
  sort Alph .                              --- the alphabet
  ops a b : -> Alph .                      --- the letters of the alphabet

  vars R R1 R2 : Ere .
  vars A B : Alph .

  op  _‘{_‘} : Ere Alph -> Ere .           --- the letters derivatives
  op epsIn_ : Ere -> Bool .                --- epsilon membership derivative

  subsort Alph < Ere .                     --- a letter
  eq epsIn A = false .
  eq B { A } = if A == B then epsilon else empty fi .

  op epsilon : -> Ere .                    --- the empty word
  eq epsilon { A } = empty .
  eq epsIn epsilon = true .

  op empty : -> Ere .                      --- the empty language
  eq empty { A } = empty .
  eq epsIn empty = false .

  op _#_ : Ere Ere -> Ere [assoc] .        --- concatenation
 ceq ( R1 # R2 ){ A } = ((R1 { A }) # R2) + (R2 { A })
     if epsIn R1 = true .
 ceq ( R1 # R2 ){ A } = (R1 { A }) # R2
     if epsIn R1 = false .
  eq epsIn ( R1 # R2 ) =  epsIn R1 and  epsIn R2 .

  op _+_ : Ere Ere -> Ere [assoc comm] .   --- union
  eq ( R1 + R2 ){ A } = (R1 { A }) + (R2 { A }) .
  eq epsIn ( R1 + R2 ) = epsIn R1 or  epsIn R2 .

  op _* : Ere -> Ere .                     --- star operator
  eq R * { A } = (R { A }) # (R *) .
  eq epsIn R * = true .

  --- simplifying equations
  eq empty + R = R .
  eq R + R = R .
  eq empty # R = empty .
  eq epsilon # R = R .
endth)
```

2.1 Circular Coinduction

As mentioned in the introduction, CIRC implements the principle of circularity, which generalizes both structural induction and circular coinduction; we will discuss this principle in depth elsewhere. We here focus on its coinductive instance. Circular coinduction [4,5,14], is a sound proof calculus for $\models\!\!\!\equiv$, which can be defined as an instance of the circularity principle as follows: the "frozen" form of equation "$(\forall X)\ t = t'\ \texttt{if}\ c$" is "$(\forall X)\mathit{fr}(t) = \mathit{fr}(t')\ \texttt{if}\ c$", where $\mathit{fr} : \mathit{sort}(t) \to \texttt{new}$ is a new operation and `new` is a new sort. The set $Der_{\Delta}(e)$ is

$$\{(\forall X)\ \mathit{fr}(\delta[t/*{:}h]) = \mathit{fr}(\delta[t'/*{:}h])\ \texttt{if}\ c \mid \delta \text{ behavioral for } h = \mathit{sort}(t)\}.$$

The frozen operator ensures the sound use of the coinduction hypotheses. We take the liberty to also call $\mathit{fr}(e)$ *visible* when e is visible, i.e., the left-hand side and right-hand side are terms of visible sorts.

In CIRC we use the standard rewriting-based semi-decision procedure to derive equations "$(\forall X)\ t = t'\ \texttt{if}\ c$": add the variables X as constants, then add the conditions in c to the set of equations, and then reduce t, t' to normal forms orienting all the equations into rewrite rules. In what follows we let $\mathcal{E} \vdash e$ denote the fact that e can be deduced from $\mathcal{E}$ using this standard approach ($\mathcal{E}$ is any set of equations). The inference relation $\vdash$ is sound for $\models$, i.e., $\mathcal{E} \vdash e$ implies $\mathcal{E} \models e$. We currently do not interfere with the rewriting procedure: if rewriting does not terminate during a proof session then CIRC does not terminate either. If we write $\mathcal{E} \not\vdash e$ then we mean "knowingly incapable of proving it", that is, that the rewrite engine reduced the two terms to normal forms, but those are not equal. Obviously, this does not necessarily mean that the equation is not true.

CIRC implements circular coinduction as a nondeterministic procedure aiming at reducing a pair $(E, \mathit{fr}(e))$ to a pair $(\mathcal{E}, \emptyset)$, where E is the original set of equations in $\mathcal{B}$ and e is the equation to prove (the goal). If that is the case, then $\mathcal{B} \models\!\!\!\equiv e$. While trying to do so, the procedure can also fail, in which case we conclude that it could not prove $\mathcal{B} \models\!\!\equiv e$, or it can run forever. Here are the reduction rules:

$$\begin{array}{l}
\text{[EqRed]} : \\
\quad (\mathcal{E}, \mathcal{G} \cup \{\mathit{fr}(e)\}) \Rightarrow (\mathcal{E}, \mathcal{G}) \quad \text{if } \mathcal{E} \vdash \mathit{fr}(e) \\
\text{[CoindFail]} : \\
\quad (\mathcal{E}, \mathcal{G} \cup \{\mathit{fr}(e)\}) \Rightarrow \mathit{failure} \qquad \text{if } \mathcal{E} \not\vdash \mathit{fr}(e) \text{ and } e \text{ is visible} \\
\text{[CCStep]} : \\
\quad (\mathcal{E}, \mathcal{G} \cup \{\mathit{fr}(e)\}) \Rightarrow (\mathcal{E} \cup \{\mathit{nf}(\mathit{fr}(e))\}, \mathcal{G} \cup \mathit{Der}_\Delta(e)) \text{ if } \mathcal{E} \not\vdash \mathit{fr}(e) \text{ and } e \text{ is hidden.}
\end{array}$$

Let $\mathit{nf}(e)$ denote the equation e where the left-hand and right-hand sides are reduced to normal forms. [EqRed] removes a goal if it can be proved using ordinary equational reduction. [CoindFail] says that the procedure fails whenever it finds a visible goal which cannot be proved using ordinary equational reduction. Finally, [CCStep] implements the circularity principle: when a behavioral equation cannot be proved using ordinary equational reduction, its frozen form (or an equivalent variant of its frozen form, such as its normal form) is added to the specification and its derivatives are added to the set of goals.

Theorem 2.1 *Let $\mathcal{B} = (\Sigma, \Delta, E)$ be a behavioral specification and let e be a Σ-equation such that $(E, \mathit{fr}(e)) \Rightarrow^\star (\mathcal{E}, \emptyset)$ using the procedure above. Then $\mathcal{B} \models\!\!\equiv e$.*

The proof of Theorem 2.1 is based on the following two lemmas.

Lemma 2.2 *If $(\Sigma \cup \{\mathit{fr}\}, \Delta, E) \models\!\!\equiv \mathit{fr}(e)$, then $(\Sigma, \Delta, E) \models\!\!\equiv e$.*

Proof. We assume without lose the generality that e is an unconditional equation $(\forall X)t = t'$. If γ is a Δ-experiment, then we have $\mathit{fr}(\gamma[t]) =_E \mathit{fr}(\gamma[t'])$ iff $\gamma[t] =_E \gamma[t']$ by the definition of $=_E$ and by the fact that E does not include equations involving fr. □

Lemma 2.3 *Let $\mathcal{E}'$ be a set of equations obtained during the reduction $(E, \mathit{fr}(e)) \Rightarrow^\star (\mathcal{E}, \emptyset)$ and let e' be a goal. If $(\Sigma, \Delta, \mathcal{E}' \cup \{\mathit{fr}(e')\}) \models \mathit{Der}_\Delta(e')$, then $(\Sigma, \Delta, \mathcal{E}') \models\!\!\equiv e'$.*

Proof. We assume again that e' is an unconditional equation $(\forall X)t = t'$. Let $<$ be the order over the experiments given by their depth. We show by Noetherian induction on $<$ that $\gamma[u] =_{\mathcal{E}'} \gamma[v]$, where u (resp. v) is $\theta(t)$ (resp. $\theta(t')$), where θ is any substitution (possible the identity). Let γ be an appropriate experiment for t (and t'). If $\gamma[t] = \gamma[t']$ is in $\mathit{Der}_\Delta(e')$, then we get $\gamma[t] =_{\mathcal{E}'} \gamma[t']$ by the hypothesis of the lemma ($\mathit{fr}(e')$ cannot be used in a derivation of $\gamma[t] = \gamma[t']$). Otherwise, there is a derivative $\delta \in \Delta$ and an experiment γ' such that $\gamma = \gamma'[\delta]$ and $\delta[t] = \delta[t']$ in $\mathit{Der}_\Delta(e')$. If $\mathit{fr}(\delta[t]) =_{\mathcal{E}'} \mathit{fr}(\delta[t'])$ ($\mathit{fr}(e')$ is not used in this derivation), then $\mathit{fr}(\gamma'[\delta[t]]) =_{\mathcal{E}'} \mathit{fr}(\gamma'[\delta[t']])$ and hence $\gamma[t] =_{\mathcal{E}'} \gamma[t']$. If $\mathit{fr}(e')$ is used in the derivation of $\mathit{fr}(\delta[t]) = \mathit{fr}(\delta[t'])$, then there is a substitution θ such that $\mathit{fr}(\delta[t]) =_{\mathcal{E}'} \mathit{fr}(\theta(t))$ and $\mathit{fr}(\delta[t']) =_{\mathcal{E}'} \mathit{fr}(\theta(t'))$. Since $\gamma' < \gamma$, we have $\gamma'[\theta(t)] =_{\mathcal{E}'} \gamma'[\theta(t')]$ by the inductive hypothesis. Hence $\gamma[t] =_{\mathcal{E}'} \gamma[t']$. So, we get $\gamma[t] =_{\mathcal{E}'} \gamma[t']$ in both cases, which implies $\gamma[\theta(t)] =_{\mathcal{E}'} \gamma[\theta(t')]$ for any substitution θ. □

Corollary 2.4 *If $(\Sigma, \Delta, \mathcal{E}' \cup \{\mathit{fr}(e')\}) \models \mathcal{G} \cup \mathit{Der}_\Delta(e')$, then $(\Sigma, \Delta, \mathcal{E}') \models\!\!\equiv \mathcal{G} \cup \{e'\}$.*

Proof of Theorem 2.1.

We proceed by induction on the number of applications of [CCStep]. If this rule is not applied, then $(\Sigma, \Delta, E \cup \{fr(e)\}) \models Der_\Delta(e)$ and the conclusion of the theorem follows by Lemma 2.3. We assume that

$$(E, fr(e)) \Rightarrow^\star (\mathcal{E}', \mathcal{G}' \cup \{fr(e')\}) \Rightarrow (\mathcal{E}' \cup \{nf(fr(e'))\}, \mathcal{G}' \cup Der_\Delta(e')) \Rightarrow^\star (\mathcal{E}, \emptyset)$$

where the last reductions are given using only [EqRed]. Hence $(\Sigma, \Delta, \mathcal{E}' \cup \{fr(e')\}) \models \mathcal{G}' \cup Der_\Delta(e')$. It follows that $(\Sigma, \Delta, \mathcal{E}') \models\!\!\!\equiv \mathcal{G}' \cup \{e'\}$ by the corollary of Lemma 2.3. The conclusion of Theorem 2.1 follows now applying the induction hypothesis. □

The *successful termination* of the CIRC procedure above, i.e., reaching of a configuration of the form $(\mathcal{E}', \emptyset)$, is not guaranteed. Let us consider, for instance, the addition of streams, behaviorally defined by $hd(S + S') = hd(S) + hd(S')$ and $tl(S + S') = tl(S) + tl(S')$, and the (convolution) product of streams, behaviorally defined by $hd(S \times S') = hd(S) \times hd(S')$ and $tl(S \times S') = tl(S) \times S' + [hd(S)] \times tl(S')$, where $[x]$ denotes the stream $x0^\omega$. The execution of CIRC procedure for the input goal $[0] \times [0] = [0]$ produces an infinite process. First [CCStep] is applied, which replace the initial goal $fr([0] \times [0]) = fr([0])$ with $fr(hd([0] \times [0])) = fr(hd([0]))$ and $fr(tl([0] \times [0])) = fr(tl([0]))$. The former is solved by [EqRed] and the latter is reduced by [EqRed] to $fr([0] \times [0] + [0] \times [0]) = fr([0])$. A new application of [CCStep] followed by [EqRed] (twice) will generate the goal $fr([0] \times [0] + [0] \times [0] + [0] \times [0] + [0] \times [0]) = fr([0])$. The process infinitely continues generating larger and larger goals. In Section 3 we extend CIRC with proof tactics able to handle such cases.

Since the behavioral entailment problem is Π^0_2-complete [14,15], we know that there can be no procedure to decide behavioral equalities or inequalities in general. The reaching of the configuration *failure* means a *failing termination*.

For regular expressions, we are in the happy case when CIRC together with the specification `RE` yield a fully automatic decision procedure for their equivalence:

Proposition 2.5 *If one defines the behavioral operators to be the two derivatives and the test for epsilon membership, then CIRC becomes a fully automatic decision procedure for the equivalence of REs:*

(i) *Regular expressions R_1 and R_2 are equivalent iff CIRC successfully terminates for the initial configuration $(Eqns(\mathtt{RE}), \{R_1 = R_2\})$;*

(ii) *Regular expressions R_1 and R_2 are not equivalent iff CIRC fails for the initial configuration $(Eqns(\mathtt{RE}), \{R_1 = R_2\})$.*

Therefore, no case analysis or other assisted tactics are needed to prove the equivalence of REs. Two REs are equivalent iff CIRC returns `Proof succeeded`, and are not equivalent iff CIRC returns `failed during coinduction`.

Here we show CIRC at work presenting how it proves the equivalence of two regular expressions. CIRC extends Full-Maude [3] with a set of declarations and commands which allow the user to introduce information regarding behavioral specifications and commands to assist the prover. First we have to introduce the behavioral operations. These are included in a `cmod...endcm` module:

```
(cmod B-RE is importing RE .
    derivative epsIn(*:Ere) .
    derivative *:Ere { a } .
    derivative *:Ere { b } .
endcm)
```

After the modules RE and B-RE are loaded, we introduce the goal we want to prove:

```
Maude> (add goal (a + b)*  = ((a *)#(b *))*  .)
Goal (a + b)*  = (a * # b *)*  added.
```

The circular coinduction algorithm is triggered with the command:

```
Maude> (coinduction .)
Proof succeeded.
```

3 Regular Strategies as Proof Tactics

A successful reduction of the circular coinduction algorithm is of the form

$$(\mathcal{E}_0, \mathcal{G}_0) \xRightarrow{\text{CCStep}} (\mathcal{E}_1, \mathcal{G}_1) \ldots \xRightarrow{\text{EqRed}} (\mathcal{E}_i, \mathcal{G}_i) \ldots \xRightarrow{\text{CCStep}} (\mathcal{E}_j, \mathcal{G}_j) \ldots \xRightarrow{\text{EqRed}} (\mathcal{E}_n, \emptyset)$$

and a failing reduction is of the form

$$(\mathcal{E}_0, \mathcal{G}_0) \xRightarrow{\text{CCStep}} (\mathcal{E}_1, \mathcal{G}_1) \ldots \xRightarrow{\text{EqRed}} (\mathcal{E}_i, \mathcal{G}_i) \ldots \xRightarrow{\text{CCStep}} (\mathcal{E}_j, \mathcal{G}_j) \ldots \xRightarrow{\text{CoindFail}} \textit{failure}$$

If we consider only the labels of the rules applied in the reduction, then the former one is described by a word in the language given by the regular expression $R_{succ} = \text{CCStep}(\text{CCStep} + \text{EqRed})^*\text{EqRed}$, and the later one by a word given by the regular expression $R_{fail} = \text{CCStep}(\text{CCStep} + \text{EqRed})^*\text{CoindFail}$. We say that the regular expression $R_{succ} + R_{fail}$ describes the circular coinduction proof tactic.

CIRC can be extended with new proof capabilities implemented as rewriting rules and new proof tactics described by regular expressions. We assume that the prover is given by a set of rewriting rules of the form $\ell : C \to C'$ `if` *cond*, where ℓ is the label of the rule, C, C' are configurations, and *cond* is the condition of the rule. Let $\mathcal{L}$ denote the set of the rules labels. The same label may be shared by more rules. This is, e.g., the case of two rules whose conditions are complementary, i.e., if one condition is true, then the other one is false. The description of the proof could become simpler if we use the same label for the two rules. We say that a word $w \in \mathcal{L}^*$ describes a reduction $C \Rightarrow^* C'$ iff $w = \ell_1 \ldots \ell_n$ and the reduction is given by $C = C_0 \xRightarrow{\ell_1} C_1 \ldots \xRightarrow{\ell_n} C_n = C'$, where $C_{i-1} \xRightarrow{\ell_i} C_i$ means that the configuration C_i is obtained from C_{i-1} by applying the rule ℓ_i. We write $C \xRightarrow{w} C'$.

A *regular strategy term* is a regular expression over $\mathcal{L}$. The *semantics* of a strategy term R consists of all the reductions $C \xRightarrow{w} C'$ with $w \in L(R)$.

We extend the definition of configurations by adding a new component given by a regular strategy term. A rewriting rule

$$\ell : C \to C' \quad \texttt{if} \quad \textit{cond}$$

is replaced by

$$\ell : (C, R) \to (C', R') \quad \texttt{if} \quad \mathit{cond} \wedge \mathit{nf}(R\{\ell\}) = R' \wedge R' \neq \mathit{empty}.$$

where $\mathit{nf}(R\{\ell\}) = R' \wedge R' \neq \mathit{empty}$ means that there is a word starting with ℓ in $L(R)$. We have $(C, R) \xRightarrow{w} (C', \varepsilon)$ if and only if $w \in L(R)$.

Let $\mathcal{B} = (\Sigma, \Delta, \mathcal{E})$ be a behavioral specification. We say that a regular strategy term R is a *proof tactic* for $\mathcal{B}$ if $\mathcal{B} \models\!\!\equiv e$ whenever $(E, \{\mathit{fr}(e)\}, R) \xRightarrow{w} (\mathcal{E}, \emptyset, \varepsilon)$.

3.1 Simplification

Since circular coinduction uses frozen forms of the goals, the coinductive hypotheses added by [CCStep] can be applied only at the top. There are cases when it is sound to apply these hypotheses under the top. For instance, we assume that we have to show the following equality over streams: $S \times [0] = [0]$. [CCStep] will add the hypothesis $\mathit{fr}(S \times [0]) = \mathit{fr}([0])$, and replace the above goal with $\mathit{fr}(\mathit{hd}(S \times [0])) = \mathit{fr}(\mathit{hd}([0]))$ and $\mathit{fr}(\mathit{tl}(S \times [0])) = \mathit{fr}(\mathit{tl}([0]))$. The former is solved by [EqRed] and the last is reduced to $\mathit{fr}(\mathit{tl}(S) \times [0] + [\mathit{hd}(S)] \times [0]) = \mathit{fr}([0])$. It is easy to see that the circular coinduction algorithm produces a infinite set of new goals in this case. For streams it is sound to simplify the goals with the following rule:

$$\frac{S_1 + S_2 = S'}{S_1 = S'} \quad \text{if } S_2 = [0]$$

Then the above goal can be simplified to $\mathit{fr}(\mathit{tl}(S) \times [0]) = \mathit{fr}([0])$ because we get $\mathit{fr}([\mathit{hd}(S)] \times [0]) = \mathit{fr}([0])$ applying the coinductive hypothesis. The remaining goal is proved in the same way. Note that the equality sign $=$ in the above rule denotes the behavioural equivalence.

The following rule handles the general case:

[Simpl] :
$$(\mathcal{E}, \mathcal{G} \cup \{\mathit{fr}(e)\}, R) \Rightarrow (\mathcal{E}, \mathcal{G}', R') \text{ if } \mathit{nf}(R\{\text{Simpl}\}) \neq \mathit{empty} \wedge \mathit{scond}$$
where
$$\mathcal{G}' = \begin{cases} \mathcal{G} \cup \{t_i = t'_i \texttt{ if } c \mid i = 1, \ldots, n\} & \text{if } e := f(t_1, \ldots, t_n) = f(t'_1, \ldots, t'_n) \texttt{ if } c \\ \mathcal{G} \cup \{\mathit{fr}(e)\} & \text{otherwise} \end{cases}$$
$$R' = \mathit{nf}(R\{\text{Simpl}\})$$

where *scond* is a condition which constraints the application of the rule (in the above example, $S_2 = [0]$ is such a condition). [Simpl] replaces a goal with the corresponding set of subgoals if the left hand side and right hand side of the equation e have in top the operator f, and let the set of goals unchanged in the other cases.

The rule [Simpl] must be used only in a controlled way, otherwise it could trigger infinite reductions, due to the “otherwise” case.

Theorem 3.1 *Let $\mathcal{B} = (\Sigma, \Delta, \mathcal{E})$ be a behavioral specification, and let $f \in \Sigma$ be an operation such that the proving of the goal $f(t_1, \ldots, t_n) = f(t'_1, \ldots, t'_n)$ `if` c can be simplified to the proving of the subgoals $\{t_i = t'_i \texttt{ if } c \mid i = 1, \ldots, n\}$ provided that the simplification condition scond holds. Then R_{cosi} = CCStep(Simpl^n # CCStep + Simpl^n # EqRed)*EqRed is a proof tactic for $\mathcal{B}$.*

Proof. Let e be an equation such that $(E, \{fr(e)\}, R_{cosi}) \overset{w}{\Longrightarrow} (\mathcal{E}, \emptyset, \varepsilon)$ with $w \in L(R_{cosi})$. We proceed by induction on the number representing how many times [Simpl] is applied. If [Simpl] is not applied, then $w \in L(R_{succ})$ and we apply Theorem 2.1. We point out one application of [Simpl] which replaces a goal e' of the form $f(t_1, \ldots, t_n) = f(t'_1, \ldots, t'_n)$ `if` c with $\{t_i = t'_i \texttt{ if } c \mid i = 1, \ldots, n\}$. We have $\mathcal{B} \models\!\equiv t_i = t'_i$ `if` c by the inductive hypothesis, for $i = 1, \ldots n$. Hence $\mathcal{B} \models\!\equiv e'$ by the assumption on $\mathcal{B}$ and f. □

[Simpl] is applied successively more than once when a goal of the form $f(\ldots f(\ldots)\ldots) = f(\ldots f(\ldots)\ldots)$ `if` c is reached. The choice of n in the definition of R_{cosi} could be a difficult task. A bigger value for n leads to a larger class of goals which can be proved by this tactic but to a less computationally efficient tactic; a smaller value leads to a more computationally efficient tactic but with a more restricted applicability. We may have both if we implement [Simpl] as follows:

[Simpl] :

$$(\mathcal{E}, \mathcal{G} \cup \{fr(e)\}, R) \Rightarrow (\mathcal{E}, \mathcal{G}', R') \texttt{ if } nf(R\{\text{Simpl}\}) \neq \textit{empty} \wedge \textit{scond}$$

where

$$(\mathcal{G}', R') = \begin{cases} (\mathcal{G} \cup \{t_i = t'_i \texttt{ if } c \mid i = 1, \ldots, n\}, R) & \text{if } e \text{ is } f(t_1, \ldots, t_n) = f(t'_1, \ldots, t'_n) \texttt{ if } c \\ (\mathcal{G}, nf(R\{\text{Simpl}\})) & \text{otherwise} \end{cases}$$

The above rule applies [Simpl] as long as there is a goal having the operator f ate the top. We may consider now $n = 1$ in Theorem 3.1. In terms of strategy languages [13,18], the new rule [Simpl] is equivalent to [Simpl'] `orelse id`, where [Simpl'] is similar to [Simpl] but it is applied only if it simplifies a goal, and `id` is the identity strategy.

[Simpl] is implemented in CIRC (version 1.3) by extending the configuration with attribute *proofStatus* having as values regular strategies R. A copy of the specification `RE` is included and used to handle the regular strategies.

4 Equivalence of Context Free Processes

In this section we illustrate the usefulness of the regular strategies showing how the result of Proposition 2.5 can be extended to context-free grammars. We have to notice that we cannot obtain a fully automatic decision procedure for context-free grammars because the equivalence problem for these grammars is undecidable [8]. Baeten, Bergstra, and Klop [1] have shown that the equivalence problem is decidable for normalized context free processes presented in Greibach normal forms. These processes are equivalent to simple (iredundant) context-free grammars. However the proof of decidability and the presentation of the algorithms are not easy and many researchers looked for simpler solutions, see , e.g., [9]. In this section we show that an adequate behavioral specification together with two simplifying rules and the proof tactic given in Theorem 3.1 supply a fully automatic semi-decision procedure for context free-processes. We prefer the formalism of context-free processes to that of context-free grammars because their equivalence is expressed as a bisimulation relation, which is more suitable for the behavioral approach.

BPA (Basic Process Algebra) *expressions* are defined by the following grammar:

$$E ::= a \mid X \mid E_1 + E_2 \mid E_1E_2$$

where a ranges over an alphabet Alph, X over variables, and E, E_1, E_2 over BPA expressions. A *process* is defined by a finite set of equations of the form $X_i \overset{\text{def}}{=} E_i$. The operational semantics of BPA process is given by the following rules:

$$\frac{E_1 \xrightarrow{a} E_1'}{E_1 + E_2 \xrightarrow{a} E_1'} \qquad \frac{E_2 \xrightarrow{a} E_2'}{E_1 + E_2 \xrightarrow{a} E_2'} \qquad \frac{E_1 \xrightarrow{a} E_1'}{E_1E_2 \xrightarrow{a} E_1'E_2}$$

$$a \xrightarrow{a} \varepsilon \quad \text{if } a \in \mathit{Alph} \qquad \frac{E \xrightarrow{a} E'}{X \xrightarrow{a} E'} \quad \text{if } X \overset{\text{def}}{=} E$$

Note that ε is not a process; it is a configuration used to mark the end of a transition process. The transition relation is extended to words w in Alph^* as follows: $p \xrightarrow{aw} q$ if $p \xrightarrow{a} p'$ and $p' \xrightarrow{w} q$. The *norm* $|E|$ of an expression E is defined by structural induction over BPA expressions: $|a| = 1$ if $a \in \mathit{Alph}$, $|E_1 + E_2| = \min(|E_1|, |E_2|)$, $|E_1E_2| = |E_1| + |E_2|$, and $|X| = |E|$ if $X \overset{\text{def}}{=} E$. The norm has the following nice property: $|X| = \min\{\mathit{length}(w) \mid X \xrightarrow{w} \varepsilon\}$. A process is *normed* if $|X_i|$ is finite for each equation $X_i \overset{\text{def}}{=} E_i$ from its definition. A relation R over processes is a *bisimulation* if it satisfies: $p\,R\,q$ implies

(i) if $p \xrightarrow{a} p'$ then there is q' such that $q \xrightarrow{a} q'$ and $p'\,R\,q'$;

(ii) if $q \xrightarrow{a} q'$ then there is p' such that $p \xrightarrow{a} p'$ and $p'\,R\,q'$.

Two processes are *bisimilar* $p \sim q$ if there is a bisimulation R such that $p\,R\,q$. The language defined by a variable X is $L(X) = \{w \mid X \xrightarrow{w} \varepsilon\}$. If $X \sim Y$, then $L(X) = L(Y)$.

Here is the behavioral specification we associate to BPA:

```
(th BPA is inc BOOL + INT .
  sort Alph .          --- the alphabet
  sort Pexp .          --- process expressions
  sort Pid .           --- process ids
  sort Peq .           --- process equations
  sort Proc .          --- processes (sets of process equations)
  op  pmain : -> Proc .

  vars E E1 E2 : Pexp .        vars X Y X1 X2 : Pid .
  vars P P1 P2 Q : Proc .      vars A B : Alph .
  op _=def_ : Pid Pexp -> Peq .

  subsort Peq < Proc .
  op _‘,_ : Proc Proc -> Proc [assoc comm] .
  eq (P , P) = P .

  op  _‘{_‘}  : Pexp Alph -> Pexp .  --- the letters derivatives
  op |_| : Pexp -> Int .             --- the norm derivative

  sort PidSet .                      --- process variables
  subsort Pid < PidSet .             --- sets of process variables
  op none : -> PidSet .
  op _‘,_ : PidSet PidSet -> PidSet [assoc comm id: none] .
  vars PS PS’ : PidSet .
  eq (X, X) = X .

  op f  : PidSet Pexp -> Int? .      --- auxiliary function
  eq f((X, PS), X) = infty .
  ceq f((X, PS), Y) = f((X, Y, PS), E)
      if ( (Y =def E), P ) := pmain /\ X =/= Y .
  eq f(PS, A ) = 1 .
  eq f(PS, (E1 + E2)) = min(f(PS, E1), f(PS, E2)) .
  eq f(PS, (E1 # E2)) = f(PS, E1) + f(PS, E2) .

  subsort Alph < Pexp .              --- a letter
  subsort Pid < Pexp .               --- a process id
  eq | A | = 1 .

  op epsilon : -> Pexp .
  eq epsilon { A } = deadlock .

  op deadlock : -> Pexp .
  eq deadlock { A } = deadlock .

  eq B { A } = if A == B then epsilon else deadlock fi .

  op _+_ : Pexp Pexp -> Pexp [prec 33 assoc comm] .  --- union
  eq ( E1 + E2 ){ A } = (E1 { A }) + (E2 { A }) .
  eq | E1 + E2 | = min(| E1 |, | E2 |) .

  op _#_ : Pexp Pexp -> Pexp [prec 32 assoc] .       --- concatenation
  eq ( E1 # E2 ){ A } = (E1 { A }) # E2 .
  eq | E1 # E2 | = | E1 | + | E2 | .

  ceq X { A } = E { A } if ( (X =def E), P ) := pmain .
  ceq | X | = f(X, E) if ( (X =def E), P ) := pmain .

--- simplifying equations
  eq deadlock + E = E .
  eq deadlock # E = deadlock .
  eq epsilon # E = E .
  eq E + E = E .
  eq (E1 + E2) # E = (E1 # E) + (E2 # E) .
 endth)
```

The behavioral operators are the letter derivatives, defined in a similar way to those of regular expressions, and the norm. The sort for process variables is `Pid`. The definition of the norm requires an auxiliary function $f(X, E)$, which returns the norm of the variable X with the definition E.

The following two simplification rules are sound for normed processes [9]:

$$\frac{E_1 E_2 \sim E_1' E_2}{E_1 \sim E_1'} \qquad \frac{E_1 + E_2 \sim E_1' + E_2'}{E_1 \sim E_1',\ E_2 \sim E_2'} \text{ if } |E_1| = |E_1'| \wedge |E_2| = |E_2'|$$

We present here how CIRC[6] together with the proof tactic R_{cosi} given by Theorem 3.1 are used for proving the equivalence $X \sim A$, where $X \overset{\text{def}}{=} aXb + ab$, $A \overset{\text{def}}{=} aB + aY$, $B \overset{\text{def}}{=} Ab$, and $Y \overset{\text{def}}{=} b$. Note that processes are not required to be in Greibach normal form. Here is a Maude specification of the above process:

```
(th BPA-EX is including BPA .
  ops a b : -> Alph .   ops X Y A B : -> Pid .
  eq pmain = ( X =def ( a # X  # b ) + a # b ),
             ( A =def (a # B) + (a # Y) ),
             ( B =def A # b ),
             ( Y =def b ) .
endth)
```

The simplification rules are specified together with the derivatives in the `cmod` module:

```
(cmod B-BPA-EX is importing BPA-EX .
    derivative *:Pexp { a } .
    derivative *:Pexp { b } .
    derivative | *:Pexp | .
    simplify
      < E1:Pexp + E2:Pexp = E1':Pexp + E2':Pexp >
    by
      < E1:Pexp = E1':Pexp > /\ < E2:Pexp = E2':Pexp >
    if
      | E1:Pexp | = | E1':Pexp | /\ | E2:Pexp | = | E2':Pexp | .
    simplify
      < E1:Pexp # E2:Pexp = E1':Pexp # E2':Pexp >
    by
      < E1:Pexp = E1':Pexp >
    if
      E2:Pexp = E2':Pexp .
endcm)
```

We first introduce the goal and then the proof tactic R_{cosi} for this goal:

```
Maude> (add goal X  = A  .)
Goal X  =  A  added.
Maude> (coinduction and simplification .)
Proof succeeded.
```

We can see the proof steps applied by the prover using "`show history .`" command:

```
Maude> (show history .)
Introduced beh spec B-BPA-EX
Goal X  =  A  added.
Hypothesis X  =  A  added.
Goal X{a} =  A{a} reduced to b + X # b  =  B + Y
Goal b + X # b  =  B + Y  simplified to
1 .  X # b  =  B
2 .  b  =  Y
Hypothesis X # b  =  B  added.
Goal b  =  Y  reduced to b  =  Y
Hypothesis b  =  Y  added.
Goal X{b} =  A{b} proved by reduction.
Goal | X |  =  | A |  proved by reduction.
Goal X # b{a} =  B{a} reduced to
b # b + X # b # b  =  B # b + Y # b
Goal b # b + X # b # b  =  B # b + Y # b  simplified to
1 .  X # b # b  =  B # b
2 .  b # b  =  Y # b
Goal X # b # b  =  B # b  simplified to X # b  =  B
Goal X # b  =  B  proved by reduction.
Goal b # b  =  Y # b  simplified to b  =  Y
Goal b  =  Y  proved by reduction.
```

[6] Regular strategies are included in the version 1.3 of CIRC.

```
Goal X # b{b} =  B{b} proved by reduction.
Goal | X # b |  =  | B |  proved by reduction.
Goal b{a} =  Y{a} proved by reduction.
Goal b{b} =  Y{b} proved by reduction.
Goal | b |  =  | Y |  proved by reduction.
Proof succeeded.
```

Note the use of the simplification rules during the proving process:

- $b + Xb = B + Y$ is simplified to $Xb = B$ and $b = Y$,
- $bb + Xbb = Bb + Yb$ is simplified to $Xbb = Bb$ and $bb = Yb$;
- $Xbb = Bb$ is simplified to $Xb = B$.

Without regular strategies, which allow to combine the circular coinduction steps with the simplification rule, the class of context-free processes for which CIRC is able to automatically prove the equivalence is much smaller. Regular strategies effectively enlarge the class of problems which can be automatically proved with CIRC. The problem of finding the complete subclass of context-free processes for which the equivalence can be automatically proved with CIRC remains open.

5 Conclusion

We presented CIRC, an automated prover supporting the principle of circularity and in particular circular coinduction. CIRC is implemented as an extension of Maude using its metalevel programming capabilities. Two novel contributions have been made in this paper. First, we showed the correctness of the circular coinductive proof strategy. Second, we showed how CIRC can be extended using regular strategies. The method is exemplified by adding a simplification rule and defining a proof tactic (as a regular strategy) used for proving the equivalence of context-free processes. The use of regular strategies as proof tactics enlarge the class of problems which can be automatically solved. CIRC implements also the circularity principle for proving properties by induction. Regular strategies make the use of proof tactics that combine coinduction with induction possible. This can be done now only in an assisted way.

References

[1] Baeten, J. C. M., J. A. Bergstra and J. W. Klop, *Decidability of bisimulation equivalence for process generating context-free languages*, Journal of the ACM **40** (1993), pp. 653 – 682.

[2] Clavel et al., M., *Maude: Specification and Programming in Rewriting Logic*, J. of TCS **285** (2002), pp. 187–243.

[3] Clavel, M., F. Durán, S. Eker and J. Meseguer, *Building Equational Proving Tools by Reflection in Rewriting Logic*, in: *Cafe: An Industrial-Strength Algebraic Formal Method*, Elsevier, 2000 .

[4] Goguen, J., K. Lin and G. Roşu, *Circular coinductive rewriting*, in: *Proceedings of Automated Software Engineering 2000* (2000), pp. 123–131.

[5] Goguen, J., K. Lin and G. Roşu, *Conditional Circular Coinductive Rewriting with Case Analysis*, in: *WADT'02*, LNCS **2755** (2003), pp. 216–232.

[6] Hausmann, D., T. Mossakowski and L. Schröder, *Iterative Circular Coinduction for CoCASL in Isabelle/HOL*, in: *Fundamental Approaches to Software Engineering 2005*, LNCS **3442** (2005), pp. 341–356.

[7] Hennicker, R., *Context induction: a proof principle for behavioral abstractions*, Formal Aspects of Computing **3** (1991), pp. 326–345.

[8] Hopcroft, J. and J.D.Ullman, "Introduction to automata theory, languages, and computation," Addison–Wesley, 1979.

[9] Hüttel, H. and C. Stirling, *Actions Speak Louder Than Words: Proving Bisimilarity for Context-Free Processes.*, J. Log. Comput. **8** (1998), pp. 485–509.

[10] Jacobs, B. and J. Rutten, *A tutorial on (co)algebras and (co)induction*, Bulletin of the European Association for Theoretical Computer Science **62** (1997), pp. 222–259.

[11] Lucanu, D. and G. Roşu, *The CIRC Prover*, `http://fsl.cs.uiuc.edu/index.php/Circ`.

[12] Lucanu, D. and G. Roşu, *Circ: A Circular Coinductive Prover*, in: *2nd Conference on Algebra and Coalgebra in Computer Science (CALCO 2007)*, Bergen, Norway, 2007, to appear in Lecture Notes in Computer Science.

[13] Marti-Oliet, N., J. Meseguer and A. Verdejo, *Towards a Strategy Language for Maude*, Electronic Notes of Theoretical Computer Science **117** (2005), pp. 417–441.

[14] Roşu, G., "Hidden Logic," Ph.D. thesis, University of California at San Diego (2000).

[15] Roşu, G., *Equality of Streams is a* Π^0_2*-Complete Problem*, in: *the 11th ACM SIGPLAN Int. Conf. on Functional Programming (ICFP'06)* (2006).

[16] Roşu, G. and M. Viswanathan, *Testing Extended Regular Language Membership Incrementally by Rewriting*, in: *RTA'03*, LNCS **2706** (2003.), pp. 499–514.

[17] Sen, K. and G. Rosu, *Generating Optimal Monitors for Extended Regular Expressions.*, Electr. Notes Theor. Comput. Sci. **89** (2003).

[18] Visser, E., Z. e. A. Benaissa and A. Tolmach, *Building program optimizers with rewriting strategies*, ACM SIGPLAN Notices **34** (1999), pp. 13–26.