Simplification and Generalization in CIRC

Eugen-Ioan Goriac Georgiana Caltais Dorel Lucanu

{egoriac, gcaltais, dlucanu}@info.uaic.ro
Faculty of Computer Science
Alexandru Ioan Cuza University
Iaşi, Romania

***Abstract*—CIRC is an automated theorem prover based on the circular coinduction principle. The tool is used for the verification of programs, behavioral equivalence checking, and proving properties over infinite data structures. In this paper we present two extensions of CIRC that handle the case when the prover indicates an infinite execution for a certain goal. The first extension involves goal simplification rules and a procedure for checking that the new execution is indeed a proof, while the second one refers to finding and proving a generalization of the goal. Each of the extensions is presented based on a case study: Binary Process Algebra (BPA) for checking the proof correctness and Streams for using generalization.**

I. Introduction

A subject of high interest in computer science refers to system specification and analysis. Systems are generally composed of several (concurrent) processes exchanging information. One can prove properties for behaviorally specified systems using the CIRC prover [9], a metalanguage application implemented as an extension of Maude [5]. Normally, the proofs can be handled automatically by this tool, but in some cases the human intervention is needed. In this paper we present two ways to minimize the human interventions. The first approach uses the simplification rules of [12] in order to ease the proving process. The novelty here is given by a technique which allows to check if the new finite execution of CIRC supplies indeed a proof of the initial goal(s). This is done by executing exactly a command of CIRC. The second approach needs to analyze the proof session and extract more general goals which can be proved easier than the ones obtained by the prover. This approach is sound and the generalization can be automatized in CIRC.

The structure of the paper is as follows. Section II provides two motivating examples in order to illustrate the necessity for the CIRC extensions. In Section III we present some basic notions regarding BPA and Streams, used throughout the paper. Section IV provides a brief description of behavioral specifications and CIRC. Both the theoretical aspects and the implementation of the two extensions are discussed in Sections V and VI.

This work is partially supported by the PNII grant IDEI 393

II. Motivating Examples

A. Simplification and its Correctness

In terms of trace equivalence, two processes are said to be equivalent if they execute the same sequences of actions [6]. Consider the two processes U and X illustrated in Figure 1. We use "$+$" to represent the *alternative composition* (non-deterministic choice) operator and "$;$" to denote the *sequential composition* operator. It is easy to see that the languages generated by the execution of these tow processes contain only words of the form: a, $a\,a\,a$, $a\,a\,a\,a\,a$, and so on (action a occurs for an odd number of times in each string).

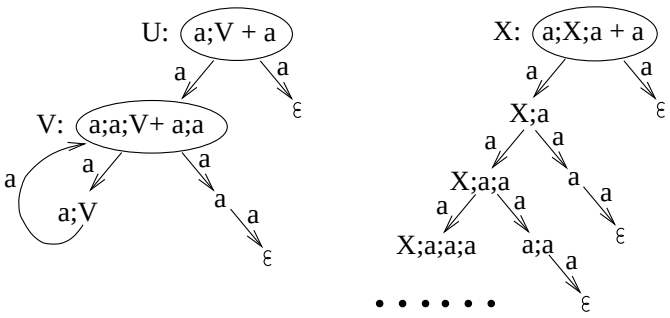

Fig. 1. Two trace-equivalent processes

Our first aim was to use CIRC in order to prove that U and X are trace-equivalent. As an extension to the work presented in [12], this paper considers infinitary trace-equivalence between processes that are not normalized. We need an extra set of simplification rules used in order to avoid infinite executions of the prover. These rules replace the current goal with a simpler one. Since we do not impose any constraints over the new goal, we need to prove that using these rules is sound. For some of these simplification rules, the soundness can be easily proved by hand, but in many cases this approach is difficult and time consuming.

For example, in the case of processes, we proved that using the following rule is sound:

$$\frac{E_1 + E_2 = E_1' + E_2'}{E_1 = E_1'} \text{ if } E_2 = E_2'$$

Note that the equalities in the above rule are interpreted as behavioral equivalences (which imply the trace equivalences).

In this paper we introduce the behavioral specification associated to basic process algebra with the infinitary trace equivalence in order to exhibit the use of the simplification rules and present our approach for automatically checking the correctness of using simplification rules. This step is compulsory in the cases when we know nothing about the soundness of the rules. For the above rule the soundness is easy to show, but this is not always the case.

B. Generalization

Streams are data structures used to model infinite behaviors. A stream is regarded as an infinite list $a_1 : a_2 : \ldots$ of data elements. Consider the mathematical definitions for the operators *tl*, *not*, *zip* and *f* over infinite binary streams:

$$\begin{aligned} tl(a:s) &= s \\ not(a:s) &= \overline{a}:not(s) \\ zip(a:s,s') &= a:zip(s',s) \\ f(a:s) &= a:\overline{a}:f(s) \end{aligned}$$

where a is a variable from $\{0,1\}$, $\overline{a}$ is the bitwise negation of a, s and s' are variables of sort stream, and ":" is the stream constructor. The Thue-Morse sequence [1] is defined by: $morse = 0:zip(not(morse), tl(morse))$.

In [9], Example 2 presents how CIRC is used in order to prove that *morse* is a fixed point for *f*. The proposed approach is not fully automatic as the user needs to follow four steps:

- start the coinduction engine for the goal (equation) *f(morse) = morse*:

```
Maude> (set show details on .)
Maude> (add goal f(morse) = morse .)
Maude> (coinduction .)
```

- analyze the output after the engine exceeds the maximum number of steps allowed

```
Hypo [* f(tl(morse)) *] =
     [* zip(tl(morse),not(tl(morse))) *]
added and coexpanded to
[...]
```

- extract an intermediary lemma that needs to be proved: *f(tl(morse)) = zip(tl(morse), not(tl(morse)))*
- start a new proof with two goals:
  - *f(morse) = morse* – the initial equation
  - *f(s) = zip(s, not(s))* – the generalization of the lemma from the previous step

```
Maude> (add goal f(morse) = morse .)
Maude> (add goal f(S:Stream) =
            zip(S:Stream, not(S:Stream)) .)
Maude> (coinduction .)

Proof succeeded [...]
```

CIRC now provides a better way to prove properties that require generalizing some goals during a proof session. The new command `(set generalization on .)` allows the engine to automatically apply generalization over equations whenever possible during a proof session:

```
Maude> (add goal f(morse) = morse .)
Maude> (set generalization on .)
Maude> (coinduction .)

[...]
Goal [* f(tl(morse)) *] =
     [* zip(tl(morse),not(tl(morse))) *]
generalized to
     [* f(S:Stream) *] =
     [* zip(S:Stream,not(S:Stream)) *]
[...]
Proof succeeded [...]
```

In this paper we prove that the use of the generalization rule is sound.

III. Basic Process Algebra and Streams

In this section we briefly present the two case studies used in the paper as running examples.

Concurrent systems are encountered in the real world; consider for example, a colony of ants. Each ant represents a particular process and the interaction between ants consists of exchanging different information about some locations or events. Another example is a network protocol, such as the alternating bit protocol [2]. Usually, system behavior is modeled as a labeled transition system, where the nodes represent system states and the edges represent the actions performed in order for the system to reach new states.

Process algebra is a framework for specification and manipulation of processes by computers and it is used for analyzing process properties and behavior. In this section we briefly introduce some theoretical notions regarding the basic process algebra; more details can be found in [6].

Basic process terms are defined by the following grammar:

$$p ::= a \mid X \mid p + p \mid p\,;p$$

where a ranges over an alphabet *Alph* and X over variables, "$+$" is the alternative composition operator and "$;$" is the sequential composition operator. A process of the form $p = p_1 + p_2$ is a process that executes either p_1 or p_2. A process of the form $p = p_1\,;p_2$ is a process that executes first p_1 and then p_2. The notation $\xrightarrow{a} \varepsilon$ is used in order to represent the successful termination after executing the action a, where ε is a special configuration that marks the end of a transition process [6]. Each atomic action a always terminates successfully after executing itself: $a \xrightarrow{a} \varepsilon$.

A process specification is a finite set of (recursive) equations of the form:

$$X_1 =_{def} p_1(X_1, \ldots, X_n)$$
$$X_2 =_{def} p_2(X_1, \ldots, X_n)$$
$$\ldots$$
$$X_n =_{def} p_n(X_1, \ldots, X_n)$$

where X_i are recursion variables, and $p_i(X_1, \ldots, X_n)$ are process terms with possible occurrences of the recursion variables. We only consider the case of *guarded* process specifications [6]. Intuitively, in a guarded specification, the right-hand sides of these equations can be adapted to:

$$a_1 ; q_1(X_1, \ldots, X_n) + \ldots + a_k ; q_k(X_1, \ldots, X_n) + b_1 + \ldots + b_l$$

where $a_i, b_j \in Alph$, by replacing the recursion variables by the right-hand sides of their recursive equations.

Basic process algebra (BPA) represents the collection of all the basic process terms. The operational semantics of BPA is given by the following transition rules (where p, p', q and q' are process terms and $X =_{def} p$ is an equation in the specification):

$$\frac{}{a \xrightarrow{a} \varepsilon} \quad if\ a \in Alph \qquad \frac{p \xrightarrow{a} p'}{X \xrightarrow{a} p'} \quad if\ X =_{def} p$$

$$\frac{p \xrightarrow{a} \varepsilon}{p + q \xrightarrow{a} \varepsilon} \qquad \frac{p \xrightarrow{a} p'}{p + q \xrightarrow{a} p'} \qquad \frac{q \xrightarrow{a} \varepsilon}{p + q \xrightarrow{a} \varepsilon} \qquad \frac{q \xrightarrow{a} q'}{p + q \xrightarrow{a} q'}$$

$$\frac{p \xrightarrow{a} \varepsilon}{p ; q \xrightarrow{a} q} \qquad \frac{p \xrightarrow{a} p'}{p ; q \xrightarrow{a} p' ; q}$$

Fig. 2. BPA operational semantics

We say that the sequence $a_1, \ldots, a_n$ is a *finite trace* of p if $p \xrightarrow{a_1} p_1 \ldots \xrightarrow{a_n} p_n$. Let $Tr(p)$ represent the set of finite traces for process p. Infinite sequences of the form $a_1, a_2, \ldots$ such that $p \xrightarrow{a_1} p_1 \xrightarrow{a_2} p_2 \ldots$ are *infinite traces* of p. Let $Tr^{\infty}(p)$ denote the set of all (finite and infinite) traces for p.

We say that two processes p and q are *infinitary trace-equivalent* ($p \sim_{tr}^{\infty} q$), if and only if $Tr^{\infty}(p) = Tr^{\infty}(q)$.

Proposition 1: Let $\{X_i =_{def} p_i \mid i \in I\}$ be a process specification. Let $p, q \in \{p_i \mid i \in I\}$.

1) $Tr^{\infty}(p + q) = Tr^{\infty}(p) \cup Tr^{\infty}(q)$.
2) $Tr^{\infty}(a ; p) = \{a\alpha \mid \alpha \in Tr^{\infty}(p)\}$.
3) $Tr^{\infty}(p) = Tr^{\infty}(p[p_i/X_i])$.

Proof: We prove all the equalities by double inclusion.

1) "$\supseteq$": Let $a_1, a_2 \ldots \in Tr^{\infty}(p)$. By definition, the transition sequence $p \xrightarrow{a_1} r_1 \xrightarrow{a_2} r_2 \ldots$ is valid ($r_1, r_2, \ldots$ are basic processes). According to the rules in Fig. 2 we deduce that $p + q \xrightarrow{a_1} r_1$ so the rewrite sequence $p + q \xrightarrow{a_1} r_1 \xrightarrow{a_2} r_2 \ldots$ is also valid, leading to $a_1, a_2 \ldots \in Tr^{\infty}(p + q)$. The proof is similar for the case $a_1, a_2 \ldots \in Tr^{\infty}(q)$.

"$\subseteq$": Let $a_1, a_2 \ldots \in Tr^{\infty}(p + q)$. In this case, the sequence $p + q \xrightarrow{a_1} r_1 \xrightarrow{a_2} \ldots$ is valid. From $p + q \xrightarrow{a_1} r_1$ we identify two situations: either $p \xrightarrow{a_1} r_1 \xrightarrow{a_2} \ldots$ or $q \xrightarrow{a_1} r_1 \xrightarrow{a_2} \ldots$, leading to either $a_1, a_2 \ldots \in Tr^{\infty}(p)$ or $a_1, a_2 \ldots \in Tr^{\infty}(q)$.

2) and 3) are proved using a similar reasoning. ■

Streams are infinite lists of elements. Supposing that a_i are data elements, a stream is denoted by an enumeration of the form $a_1 : a_2 : a_3 : \ldots$ A stream represents the solution to a system of equations (such as the one provided in Section II-B). For instance $zero = 0 : zero$ is the infinite stream of elements that have the value 0 ($0 : 0 : \ldots$). For a detailed description of streams and their properties see [14], [15] and [17]. Note that streams may be regarded as particular cases of binary process algebras with guarded process specifications.

IV. Behavioral Specifications and CIRC

We assume the reader familiar with the algebraic specifications [7]. In this section we present the notations and the concepts used in the paper.

Let the triple $\mathcal{B} = (S, \Sigma, E)$ denote a many sorted algebraic specification, where S is the set of *sorts*, $\Sigma = \{\Sigma_{w,s} \mid w \in S^*, s \in S\}$ is an $S^* \times S$-indexed set of operations, named *algebraic signature*, and E is a set of Σ-*equations*. Let $(\forall X)\ t = t'$ `if` c be a Σ-*equation*. t and t' are Σ-terms over variables X in a fixed S-indexed set of variables and, c is the *condition* of the equation consisting of a finite set of pairs (u_i, v_i) of terms over variables X (note that the condition can also be empty). We often write the condition of an equation as $\wedge_{i \in I}(u_i = v_i)$.

A *behavioral specification* is a pair of the form $(\mathcal{B}, \Delta)$, where Δ is a set of Σ-contexts, called *derivatives*. A derivative in Δ is a context written as $\delta[*{:}h]$, where $*$ is a special variable of sort h. S consists of *hidden sorts*, $H = \{h \mid \delta[*{:}h] \in \Delta\}$, and *visible sorts*, $V = S \setminus H$.

A Δ-*experiment* for the hidden sort $h \in H$ is inductively defined as follows: each derivative for the hidden sort $h \in H$ with visible result sort is a Δ-experiment for h; if C is a Δ-experiment for h' and δ a behavioral operation for h with result sort h', then $C[\delta]$ is a Δ-experiment for h. As above, $C[\delta]$ denotes the term obtained from C by replacing δ for the distinguished variable $*{:}h'$.

Behavioral equivalence between two terms of hidden sort is defined as the "indistinguishability under experiments". We denote the relation of behavioral equivalence by $\equiv$. If T_1 and T_2 are terms of hidden sort, then $T_1 \equiv T_2$ if and only if $C[T_1] = C[T_2]$, for all Δ-experiments C.

Next we provide the behavioral specifications we considered for BPA and streams, as case studies.

A. BPA Behavioral Specification

The equational specification of the processes is defined by the following items:

- a sort *Alph* for the atomic actions (the alphabet)
- a sort *Pid* for the process variables
- a sort *Pexp* for the process terms (expressions)
- the constructors for process terms:

$$Alph < Pexp, \quad Pid < Pexp$$
$$\_+\_ : Pexp\ Pexp \to Pexp$$
$$\_;\_ : Pexp\ Pexp \to Pexp$$

- a sort *Peq* together with the constructor

$$\_=_{def}\_ : Pid\ Pexp \to Peq$$

for the process equations

- a sort *Set*{*Peq*} together with the constructors

$$Peq < Set\{Peq\} \text{and}$$
$$\_,\_ : Set\{Peq\}\ Set\{Peq\} \to Set\{Peq\}$$

for the sets of process equations.

Now we provide the behavioral aspect of the specification. We consider two special process constants: $\perp$ such that $p \xrightarrow{a} \perp$ iff $\nexists q.p \xrightarrow{a} q$, and ε such that $\forall a.a \xrightarrow{a} \varepsilon$ (this corresponds to a successful termination). We also consider two derivatives. The first one, $*$:*Pexp* $\{A$:$Alph\}$, has the intended meaning $Tr^{\infty}(p\{a\}) = \{\alpha \mid a\alpha \in Tr^{\infty}(p)\}$ and it is similar to Brzozowski derivatives for the regular expressions. The second derivative is $\perp?(*$:*Pexp*$)$, and has the meaning $\perp?(p) =$ *true* iff $\forall a.p \xrightarrow{a} \perp$.

The derivatives are defined as

$$\_\{\_\} : \textit{Pexp}\ \textit{Alph} \to \textit{Pexp}$$
$$\perp? : \textit{Pexp} \to \textit{Bool}$$

together with the following equations:

$$\begin{array}{ll}
(p+q)\{a\} = p\{a\} + q\{a\} & \perp?(p+q) = \perp?(p) \vee \perp?(q) \\
(p\,;q)\{a\} = p\{a\}\,;q \ \texttt{if} & \perp?(p\,;q) = \perp?(p) \\
\quad p \neq \varepsilon \wedge p \neq \perp & \perp?(\perp) = \textit{true} \\
a\{a\} = \varepsilon & \perp?(\varepsilon) = \textit{false} \\
b\{a\} = \perp \ \ \texttt{if} \ \ b \neq a & \perp?(a) = \textit{false} \\
\varepsilon\{a\} = \perp & \perp?(X) = \textit{false} \\
X\{a\} = r\{a\} &
\end{array}$$

where $X =_{def} r$ is a process equation in the specification, p and q are basic processes and a is an element from *Alph*.

The function $Tr^{\infty}(\_)$ is extended as follows:

$$Tr^{\infty}(\perp\,;p) = Tr^{\infty}(\perp) = \emptyset$$
$$Tr^{\infty}(\perp + p) = Tr^{\infty}(p)$$
$$Tr^{\infty}(\varepsilon) = \{\lambda\}, \text{where } \lambda \text{ is the empty trace}$$
$$Tr^{\infty}(\varepsilon\,;p) = Tr^{\infty}(p)$$

The equations associated to this extension that need to be added to the specification are:

$$\perp\,;p = \perp$$
$$\perp + p = p$$
$$\varepsilon\,;p = p$$

Theorem 1: If $a \in$ *Alph* and p is a process therm s.t. $p \notin \{\perp, \varepsilon\}$ then $Tr^{\infty}(p\{a\}) = \{\alpha \mid a\alpha \in Tr^{\infty}(p)\}$.

Proof: We proceed by structural induction on p.

Case $p = p_1 + p_2$. According to the equations used for defining the derivative $\_\{\_\}$ and to Proposition 1, the equalities $Tr^{\infty}((p+q)\{a\}) = Tr^{\infty}(p\{a\} + q\{a\}) = Tr^{\infty}(p\{a\}) \cup Tr^{\infty}(q\{a\})$ hold. By the induction hypothesis, we deduce that $Tr^{\infty}(p\{a\}) \cup Tr^{\infty}(p\{a\}) = \{\alpha \mid a\alpha \in Tr^{\infty}(p)\} \cup \{\alpha \mid a\alpha \in Tr^{\infty}(q)\} = \{\alpha \mid a\alpha \in Tr^{\infty}(p) \cup Tr^{\infty}(q)\} = \{\alpha \mid a\alpha \in Tr^{\infty}(p+q)\}$.

The other cases are proved in a similar manner. ■

Proposition 2: Let p be a basic process and $a, a_i \in$ *Alph*, $i \geq 1$.

1) $a \in Tr^{\infty}(p) \Leftrightarrow p\{a\} \neq \perp$.
2) $a_1, \ldots, a_n \in Tr(p) \Leftrightarrow p\{a_1\} \ldots \{a_n\} \neq \perp$.
3) $a_1, a_2 \ldots \in Tr^{\infty}(p) \Leftrightarrow (\forall n) a_1, \ldots, a_n \in Tr(p) \Leftrightarrow (\forall n) p\{a_1\} \ldots \{a_n\} \neq \perp$.

Two processes p and q are *behavioral equivalent* ($p \equiv q$) if and only if $C[p] =_E C[q]$ for all experiments C (see [13] for more details). In our case, the experiments are of the form $\perp?(*$:$Pexp\{a_1\} \ldots \{a_n\})$.

Theorem 2: If $p \equiv q$ then $p \sim_{tr}^{\infty} q$.

Proof: Recall that $p \sim_{tr}^{\infty} q$ if and only if $Tr^{\infty}(p) = Tr^{\infty}(q)$. We proceed by proving this equality. $a_1, \ldots, a_n \in Tr^{\infty}(p) \Leftrightarrow \perp?(p\{a_1\} \ldots \{a_n\}) =$ *false* $\Leftrightarrow \perp?(q\{a_1\} \ldots \{a_n\}) =$ *false* $\Leftrightarrow a_1, \ldots, a_n \in Tr^{\infty}(q)$. $a_1, a_2, \ldots \in Tr^{\infty}(p) \Leftrightarrow (\forall n) a_1, \ldots, a_n \in Tr(p) \Leftrightarrow (\forall n) a_1, \ldots, a_n \in Tr(q) \Leftrightarrow a_1, a_2, \ldots \in Tr^{\infty}(q)$.

■

B. Streams Behavioral Specification

In order to specify streams, we consider two sorts: a hidden sort $Stream$ for the streams and a visible sort $Data$ for the stream elements. The streams are defined in terms of the derivatives head ($hd : Stream \to Data$) and tail ($tl : Stream \to Stream$).

The behavior for an operation over streams is defined using these two derivatives. For instance, the operations not, zip, f, and $morse$ introduced in Section II-B are defined by the following equations:

$$hd(not(s)) = \overline{hd(s)}$$
$$tl(not(s)) = not(tl(s))$$
$$hd(zip(s_1, s_2)) = hd(s_1)$$
$$tl(zip(s_1, s_2)) = zip(s_2, tl(s_1))$$
$$hd(f(s)) = hd(s)$$

$$hd(tl(f(s))) = \overline{hd(s)}$$
$$tl(tl(f(s))) = f(tl(s))$$
$$hd(morse) = 0$$
$$hd(tl(morse)) = 1$$
$$tl(tl(morse)) = zip(tl(morse), not(tl(morse)))$$

As $Data$ is the only visible sort in our specification, experiments over streams are of the form $hd(*)$, $hd(tl(*))$, $hd(tl(tl(*)))\ldots$, where $*$ is a variable of sort $Stream$.

C. The CIRC Theorem Prover

CIRC is an automated tool used for inductive and coinductive theorem proving, created as an extension of Maude. This tool can be used in program verification, behavioral equivalence checking and for proving process bisimilarity.

As presented in [9], the circular coinduction engine implements the proof system given in [13] by a set of reduction rules $(\mathcal{B}, \mathcal{F}, \mathcal{G}) \Rightarrow (\mathcal{B}, \mathcal{F}', \mathcal{G}')$, where $\mathcal{B}$ represents the (original) algebraic specification, $\mathcal{F}$ is the set of frozen axioms and $\mathcal{G}$ is the current set of proof obligations.

In order to present these rules, we introduce the underlying entailment relation used in CIRC:

$$\mathcal{E} \vdash_{\rightarrow\leftarrow} (\forall X) t = t' \texttt{ if } \wedge_{i\in I}\,(u_i = v_i) \;\text{ iff }\; \mathrm{nf}(t) = \mathrm{nf}(t')$$

where $\mathrm{nf}(t)$ is computed as follows:

- the variables X of the equations are turned into fresh constants;
- the condition equalities $u_i = v_i$ are added as equations to the specification;
- the equations in the specification are oriented and used as rewrite rules.

The normal form $\mathrm{nf}(e)$ of an equation e of the form $(\forall X) t = t' \texttt{ if } c$ is $(\forall X)\mathrm{nf}(t) = \mathrm{nf}(t') \texttt{ if } c$, where the constants from the normal forms are turned back into the corresponding variables.

In order to implement the circularity principle [13], we use a freezing operator $\boxed{-}$, declared in CIRC as the operator `[* *]`. For an equation $(\forall X)\ t = t'$ if c, the corresponding frozen equation is: $(\forall X)\ \boxed{t} = \boxed{t'}$ if c. The role of the freezing operator is to forbid the application of the coinductive hypotheses when the current goal does not match it at the top.

Here is a brief description of the reduction rules included in the CIRC prover engine:

[Done]: $(\mathcal{B}, \mathcal{F}, \emptyset) \Rightarrow \cdot$
- applied whenever the set of proof obligations is empty; indicates the termination of the reduction process.

[Reduce]: $(\mathcal{B}, \mathcal{F}, \mathcal{G} \cup \{\boxed{e}\}) \Rightarrow (\mathcal{B}, \mathcal{F}, \mathcal{G})$ `if` $\mathcal{B} \cup \mathcal{F} \vdash_{\rightarrow\leftarrow} e$
- applied whenever the current goal is a $\vdash_{\rightarrow\leftarrow}$-consequence of $\mathcal{B} \cup \mathcal{F}$; it removes $\boxed{e}$ from the set of goals.

[Derive]: $(\mathcal{B}, \mathcal{F}, \mathcal{G} \cup \{\boxed{e}\}) \Rightarrow (\mathcal{B}, \mathcal{F} \cup \{\boxed{e}\}, \mathcal{G} \cup \{\boxed{\Delta(e)}\})$
`if` $\mathcal{B} \cup \mathcal{F} \not\vdash_{\rightarrow\leftarrow} e$ and e is hidden
- applied when the current goal e is hidden and it is not a $\vdash_{\rightarrow\leftarrow}$-consequence; it adds the current goal to the hypothesis and its derivatives to the set of goals. $\Delta(e)$ denotes the set $\{\delta[e] \mid \delta \in \Delta\}$.

[Normalize]: $(\mathcal{B}, \mathcal{F}, \mathcal{G} \cup \{\boxed{e}\}) \Rightarrow (\mathcal{B}, \mathcal{F}, \mathcal{G} \cup \{\boxed{\mathrm{nf}(e)}\})$
- removes the current goal from the set of proof obligations and adds its normal form as a new goal.

[Fail]: $(\mathcal{B}, \mathcal{F}, \mathcal{G} \cup \{\boxed{e}\}) \Rightarrow$ *fail*
`if` $\mathcal{B} \cup \mathcal{F} \not\vdash_{\rightarrow\leftarrow} e$ and e is visible
- stops the reduction process with failure whenever the current goal e is visible and the corresponding normal forms are different.

The system above is parametric over an entailment relation $\vdash_{\rightarrow\leftarrow}$ that satisfies the properties presented in [13].

It is easy to see that the reduction rules [Done], [Reduce], and [Derive] implement the proof rules with the same names given in [13]. The reduction rules [Normalize] and [Fail] have no correspondent in the proof system and are used to ease the user interaction with the prover.

As already mentioned, CIRC is a behavioral extension of Maude. The derivatives (behavioral operations) are declared in a separate CIRC module (`ctheory ... endctheory`), which extends the Maude-specific functional theory module. Each derivative is specified using the keyword `derivative` (the shortened form `der` can also be used). When specifying processes one has to consider declarations of the form `derivative *:Pexp{a}` for each $a \in Alph$ and `derivative bot?(*:Pexp)`. For the case of streams, the derivative declarations are: `derivative hd(*:Stream)` and `derivative tl(*:Stream)`.

In order to test the examples presented in this paper, the reader may use the online version of CIRC:

http://fsl.cs.uiuc.edu/index.php/Special:CircOnline.

V. Simplifying Rules and Proof Checking

Recall from the example presented in Section II-A that our task is to prove that $X \sim^{\infty}_{tr} U$, where $X =_{def} (a\,;X\,;a) + a$, $U =_{def} (a\,;V) + a$, and $V =_{def} (a\,;a\,;V) + (a\,;a)$. We provide the specification for these processes:

```
ctheory BPA is
  including BPA-EQ .

  ops a : -> Alph .
  ops X U V : -> Pid .

  eq pmain =
     ( X =def (a ; X ; a) + a ),
     ( U =def (a ; V) + a ),
     ( V =def (a ; a ; V) + (a ; a) ) .

  derivative *:Pexp { a } .
  derivative bot?(*:Pexp) .

  scx *:Pexp  + PX:Pexp .
  scx PX:Pexp + *:Pexp .
endctheory
```

Here `pmain` represents the specification of all processes. Also, we explicitly specify the operator "+" as special context [10].

We proceed by adding the goal $U = X$ in the usual manner and trying to prove it by coinduction. We also choose to view all the details of the steps performed during the proof session by using the command (`set show details on .`):

```
Maude> (add goal U = X .)
Maude> (set show details on .)
Maude> (coinduction .)

Hypo [* U *] = [* X *]
  added and coexpanded to
1. [* U{a} *] = [* X{a} *]
2. [* bot?(U) *] = [* bot?(X) *]

Goal [* U{a} *] = [* X{a} *] normalized to
     [* V + epsilon *] = [* epsilon + X ; a *]

Hypo [* V + epsilon *] = [* epsilon + X ; a *]
  added and coexpanded to
1. [* (V + epsilon)a *] =
   [* (epsilon + X ; a)a *]
2. [* bot?(V + epsilon) *] =
   [* bot?(epsilon + X ; a) *]

Goal [* bot?(U) *] = [* bot?(X) *]
  normalized to
     [* false *] = [* false *]
Goal [* false *] = [* false *]
  proved by reduction.

Goal [* (V + epsilon)a *] =
     [* (epsilon + X ; a)a *]
  normalized to
     [* a + a ; V *] = [* a + X ; a ; a *]

[...]

Stopped:
  the number of prover steps was exceeded.
```

The prover exceeds the maximum number of steps allowed. It is easy to see that the circular coinduction algorithm produces an infinite set of new goals. A solution to this problem is using simplification rules, as described in [12]. By analyzing the provided output, we see that instead of applying a derivation, the prover could use a simplification rule for the second step. The rule is the one specified in the motivating example:

$$\frac{E_1 + E_2 = E_1' + E_2'}{E_1 = E_1'} \text{ if } E_2 = E_2'$$

For instance, according to our example, this rule simplifies the goal $\boxed{a + a\,;V} = \boxed{a + X\,;a\,;a}$ to $\boxed{a\,;V} = \boxed{X\,;a\,;a}$.

After adding the corresponding simplification rule

```
  csrl (E1:Pexp + E2:Pexp) =
       (E1':Pexp + E2':Pexp)
    => (E1:Pexp = E1':Pexp)
     if (E2:Pexp = E2':Pexp)
  .
```

to the specification and running the example for the second time, we still encounter the problem of infinite rewriting. By using a similar technique, we identify another simplification rule:

$$\frac{E_1 = E_2 + (a\,;a\,;E_3) + (X\,;a\,;a\,;a\,;E_3)}{E_1 = E_2 + V\,;E_3}$$

After adding the corresponding rule

```
 srl (E1:Pexp) =
     (E2:Pexp +
     (a ; a ; E3:Pexp) +
     (X ; a ; a ; a ; E3:Pexp))
  => (E1:Pexp) = (E2:Pexp + (V ; E3:Pexp))
 .
```

to the specification, the engine manages to prove our goal. Note that the prover prints all the intermediate properties proved during the session.

```
Maude> (add goal U = X .)
Maude> (coinduction .)
Proof succeeded.
  Number of derived goals: 14
  Number of proving steps performed: 86
  Max. number of proving steps is set to: 256
Proved properties:
  a ; V = a ; a ; a + a ; V ; a ; a
  V = a ; a + V ; a ; a
  a ; V = a ; a ; a + X ; a ; a ; a ; a
  V = a ; a + X ; a ; a ; a
  a ; V = X ; a ; a
  V = X ; a
  U = X
```

In order to illustrate the use of simplifying rules, we provide the key parts of the proof obtained by using the command (`show proof .`). The proof is given in terms of the inference rules described in [13]:

```
[...]
[* V *] = [* a ; a + V ; a ; a *]
---------------------------------- [Simplify]
[* V *] = [* a ; a + a ; a ; a ; a +
            X ; a ; a ; a ; a ; a *]

[...]

1. |||-  [* V{a} *] = [* (X ; a){a} *]
2. |||-  [* bot?(V) *] = [* bot?(X ; a) *]
---------------------------------- [Derive]
|||-  [* V *] = [* X ; a *]


[* V *] = [* X ; a *]
---------------------------------- [Simplify]
[* V + epsilon *] = [* epsilon + X ; a *]
```

```
|-  [* V + epsilon *] = [* epsilon + X ; a *]
---------------------------------- [Normalize]
|-  [* U{a} *] = [* X{a} *]


1. |||-  [* U{a} *] = [* X{a} *]
2. |||-  [* bot?(U) *] = [* bot?(X) *]
---------------------------------- [Derive]
|||-  [* U *] = [* X *]
```

The problem that arises when using simplification rules during a proof is the soundness of the proof itself. One could, for instance, use a simplification rule that transforms the current goal into $true = true$, thus allowing CIRC to "prove" any goal.

In order to check the correctness of the proof, we need to consider the set $\mathcal{F}$ of all the lemmas obtained during the execution of the prover:

- the goals proved using [Derive], such as: $\boxed{U = X}$, $\boxed{V = X\,;a}$, etc.
- the goals before and after applying [Simplify], such as: $\boxed{V + epsilon} = \boxed{epsilon + X\,;a}$, etc.

If we manage to prove that all the properties in $\mathcal{F}$ hold without using the simplification rules, then the proof is correct. This follows directly from Theorem 2 in [8] for $(\mathcal{B}, \emptyset, \mathcal{F}) \Rightarrow^* (\mathcal{B}, \mathcal{F}', \emptyset)$.

Considering the language ROC! presented in [4], the strategy applied for proving the goals in $\mathcal{F}$ is:

$$([\text{Normalize}] \triangleright [\text{Reduce}] \triangleright [\text{Derive}])!$$

We mention that $\triangleright$ has the semantics of an "orelse"-like operator, while ! imposes the prover to apply the indicated strategy for as many times as possible (*i.e.* until it succeeds or it fails).

The command that starts the execution of the prover for the set $\mathcal{F}$ automatically obtained from the last proof is `(check proof .)`. The user dialog for checking the correctness of the proof is:

```
Maude> (check proof .)
Check proof succeeded.
```

This message indicates that CIRC manages to successfully check the correctness of the proof that uses simplification rules.

An important note is that if CIRC is able to prove the current goal only by using equational reduction, then the simplification rule is not applied.

VI. Generalization

In addition to the rules enumerated in Section IV-C, we present the generalization rule and prove that the system remains sound after including it:

[Generalize]: $(\mathcal{B}, \mathcal{F}, \mathcal{G} \cup \{\forall Y\ \boxed{\theta(t)} = \boxed{\theta(t')}\}) \Rightarrow$
$(\mathcal{B}, \mathcal{F}, \mathcal{G} \cup \{\forall Y\ \boxed{t} = \boxed{t'}\})$,
where $\theta : X \to T_\Sigma(Y)$ is a substitution.

Before introducing the theorem for the soundness of the system enriched with the generalization rule, we recall the substitution theorem:

Theorem 3: Let $\theta : X \to T_\Sigma(Y)$ be a substitution. The following property holds:

$$\text{if } \ \mathcal{E} \vdash (\forall Y)\ t = t' \ \text{ then } \ \mathcal{E} \vdash (\forall Y)\ \theta(t) = \theta(t').$$

Theorem 4: If $\mathcal{B}$ is a behavioral specification, G is a set of equations, and $(\mathcal{B}, \mathcal{F}_0 = \emptyset, \mathcal{G}_0 = \boxed{G}) \Rightarrow^* (\mathcal{B}, \mathcal{F}_n, \mathcal{G}_n = \emptyset)$, using [Reduce], [Derive] and [Generalize], defined over a given entailment relation $\vdash$, then $\mathcal{B} \Vdash G$.

Proof: Note that for every $i = \overline{0..n}$, there is some $\boxed{e} \in \mathcal{G}_i$ such that one of the following statements holds, each corresponding to the three rules:

[Reduce]: $\mathcal{B} \cup \mathcal{F}_i \vdash \boxed{e}$, $\mathcal{G}_{i+1} = \mathcal{G}_i - \{\boxed{e}\}$ and $\mathcal{F}_{i+1} = \mathcal{F}_i$

[Derive]: $\mathcal{G}_{i+1} = (\mathcal{G}_i - \{\boxed{e}\}) \cup \boxed{\Delta(e)}$ and $\mathcal{F}_{i+1} = \mathcal{F}_i \cup \boxed{e}$

[Generalize]: $\mathcal{G}_{i+1} = (\mathcal{G}_i - \{\boxed{e}\}) \cup \boxed{gen(e)}$ $\mathcal{F}_{i+1} = \mathcal{F}_i$

The function $gen(e)$ replaces each occurrence of a subterm that appears under both sides of e with a variable of the same sort as the subterm.

Consider the set $\mathcal{F} = \bigcup_{i=\overline{0..n}} \mathcal{F}_i$. Let us prove that $\forall i = \overline{0..n}\ \ \mathcal{B} \cup \mathcal{F} \vdash \mathcal{G}_i$ by induction over $n - i$. The *base case*, $i = n$, follows directly because $\mathcal{B} \cup \mathcal{F} \vdash \emptyset = G_n$.

For the *inductive step* we assume that $0 \leq i < n$. Let $\boxed{e} \in \mathcal{G}_i$. If $\boxed{e} \in \mathcal{G}_{i+1}$ then $\mathcal{B} \cup \mathcal{F} \vdash \boxed{e}$ by the induction hypothesis. If $\boxed{e} \notin \mathcal{G}_{i+1}$ then we distinguish three cases:

[Reduce]: $\mathcal{B} \cup \mathcal{F}_i \vdash \boxed{e}$; but $\mathcal{F}_i \subseteq \mathcal{F}$, therefore $\mathcal{B} \cup \mathcal{F} \vdash \boxed{e}$

[Derive]: $\boxed{e} \in \mathcal{F}_{i+1}$; but $\mathcal{F}_{i+1} \subseteq \mathcal{F}$, therefore $\mathcal{B} \cup \mathcal{F} \vdash \boxed{e}$

[Generalize]: $\boxed{gen(e)} \in \mathcal{G}_{i+1}$; $\mathcal{B} \cup \mathcal{F} \vdash \boxed{gen(e)}$ (by the induction hypothesis) and $\boxed{gen(e)} \vdash \boxed{e}$ (by Theorem 3), so $\mathcal{B} \cup \mathcal{F} \vdash \boxed{e}$

We proved that $\forall i = \overline{0..n}\ \ \mathcal{B} \cup \mathcal{F} \vdash \mathcal{G}_i$, and therefore, by Theorem 2 in [13], $\mathcal{B} \Vdash G$.

■

The user needs to pay attention when using the generalization during coinductive proofs because CIRC does not detect over-generalizations. In this way, some goals that are proved without using this rule may not be proved when using it.

VII. Conclusions and Related Work

In this paper we presented two non-trivial extensions of CIRC, a theorem prover implementing the circular coinduction principle, in order to prove a set of properties over infinite data structures and over BPA processes.

One of the paper contributions consisted in providing a new technique for checking the correctness of the proofs when

using the simplification rules introduced in [11]. As a case study we considered the infinitary trace equivalence for a non-trivial BPA example. A resembling behavioral specification for BPA is found in [10], where the infinitary completed trace equivalence is considered. The difference between the two types of equivalences is presented in [16].

We enhanced the CIRC proving engine with a generalization rule used in order to transform a goal into a more general one, that sometimes can be proved easier than the initial one. We also proved the soundness of the system enhanced with the generalization rule. The idea of using generalization is not new in the domain of automated provers (see [3]). The contribution of the article consisted in applying the rule in the context of circular coinduction in a fully automated manner.

REFERENCES

[1] J.-P. Allouche and J. Shallit. The ubiquitous prouhet-thue-morse sequence. In *Sequences and Their applications (Proc. SETA'98)*, pages 1–16. Springer-Verlag, 1999.

[2] K. A. Bartlett, R. A. Scantlebury, and P. T. Wilkinson. A note on reliable full-duplex transmission over half-duplex links. *Communications of the ACM*, 12(5):260–261, 1969.

[3] R. S. Boyer and J. S. Moore. Proof-checking, theorem-proving, and program verification. Technical report, 1984.

[4] G. Caltais, E.-I. Goriac, D. Lucanu, and G. Grigoras. A Rewrite Stack Machine for ROC! *Symbolic and Numeric Algorithms for Scientific Computing, International Symposium on*, 0:85–91, 2008.

[5] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. L. Talcott, editors. *All About Maude - A High-Performance Logical Framework, How to Specify, Program and Verify Systems in Rewriting Logic*, volume 4350 of *Lecture Notes in Computer Science*. Springer, 2007.

[6] W. Fokkink. *Introduction to Process Algebra*. Springer-Verlag, Berlin, 2000.

[7] J. Goguen and J. Meseguer. Completeness of Many-Sorted Equational Logic. *Houston Journal of Mathematics*, 11(3):307–334, 1985.

[8] G. Grigoraş and D. L. G. Caltais, E. Goriac. Automated proving of the behavioral attributes. Proceedings of the 4th Balkan Conference in Informatics (BCI'09), 2009.

[9] D. Lucanu, E.-I. Goriac, G. Caltais, and G. Roşu. CIRC : A behavioral verification tool based on circular coinduction. In *CALCO'09*, volume 5728 of *Lecture Notes in Computer Science*, pages 433–442, 2009.

[10] D. Lucanu and G. Roşu. Circular Coinduction with Special Contexts. Technical Report UIUCDCS-R-2009-3039, University of Illinois at Urbana-Champaign, 2009. Accepted for ICFEM 2009.

[11] D. Lucanu, G. Roşu, and G. Grigoraş. Regular Strategies as Proof Tactics for CIRC Prover. In *7th International Workshop on Reduction Strategies in Rewriting and Programming*, 2007. to appear in ENTCS.

[12] D. Lucanu, G. Roşu, and G. Grigoraş. Regular strategies as proof tactics for circ. *Electron. Notes Theor. Comput. Sci.*, 204:83–98, 2008.

[13] G. Roşu and D. Lucanu. Circular Coinduction – A Proof Theoretical Foundation. In *CALCO'09*, volume 5728 of *Lecture Notes in Computer Science*, pages 127–144, 2009.

[14] J. J. M. M. Rutten. Behavioural Differential Equations: A Coinductive Calculus of Streams, Automata, and Power Series. *Theoretical Computer Science*, 308(1–3):1–53, 2003.

[15] J. J. M. M. Rutten. A coinductive calculus of streams. *Mathematical Structures in Computer Science*, 15(1):93–147, 2005.

[16] R. van Glabbeek. The linear time-branching time spectrum i - the semantics of concrete, sequential processes. In *Handbook of Process Algebra, chapter 1*, pages 3–99. Elsevier.

[17] H. Zantema. Well-definedness of streams by termination. RTA'09, to app. (LNCS).