On the Expressive Power of Concurrent Constraint Programming Languages

Mogens Nielsen
Catuscia Palamidessi
Frank D. Valencia

BRICS Report Series RS-02-22

ISSN 0909-0878 May 2002

On the Expressive Power of Temporal Concurrent Constraint Programming Languages

Mogens Nielsen[*]
BRICS, University of Aarhus
mn@brics.dk

Catuscia Palamidessi[†]
Penn State University
catuscia@cse.psu.edu

Frank D. Valencia[*]
BRICS, University of Aarhus
fvalenci@brics.dk

May 8, 2002

Abstract

The tcc paradigm is a formalism for timed concurrent constraint programming. Several tcc languages differing in their way of expressing infinite behavior have been proposed in the literature. In this paper we study the expressive power of some of these languages. In particular, we show that: (1) recursive procedures with parameters can be encoded into parameterless recursive procedures with dynamic scoping, and viceversa. (2) replication can be encoded into parameterless resursive procedures with static scoping, and viceversa. (3) the languages from (1) are strictly more expressive than the languages from (2). Furthermore, we show that behavioral equivalence is undecidable for the languages from (1), but decidable for the languages from (2). The undecidability result holds even if the process variables take values from a fixed finite domain.

[*]The contribution of Mogens Nielsen and Frank D. Valencia to this research has been supported by Basic Research in Computer Science, Centre of the Danish National Research Foundation.

[†]The contribution of Catuscia Palamidessi to this research has been supported by the NSF-POWRE grant EIA-0074909.

1 Introduction

Timed concurrent constraint programming (tcc) was introduced in [15] as an extension of concurrent constraint programming (ccp) aimed at specifying timed systems, following the paradigms of Synchronous Languages ([1]). As argued in [15, 19, 17, 12], tcc has a *declarative* nature that distinguishes it from other timed formalisms. Indeed, tcc programs (or processes) can be viewed as first-order linear-time temporal logic formulas ([15, 12]). Furthermore, tcc languages have simple fully-abstract semantics based on solutions of equations ([15, 12]).

In tcc time is conceptually divided into *discrete intervals (or time units)*. Intuitively, in a particular timed interval, a ccp process P receives a stimulus (i.e. piece of information represented as a constraint) c from the environment, it executes with this stimulus as the initial store, and when it reaches its resting point, it responds to the environment with the resulting store d. The resting point also determines a residual process Q, which is then executed in the next time interval.

The finite tcc processes provide for the telling and asking of information, and basic operators for parallel composition, locality and unit-delay. In the literature there are several tcc process languages variants differing in their way of extending standard finite processes in order to express infinite behavior. The main purpose of this paper is to study the expressive power of a few fundamental representatives of these processes languages. This way, we believe that we can contribute to the better understanding of tcc languages.

We shall study in detail the following extensions of finite process:

- $\texttt{rep}$: obtained by adding a replication operator similar to the one of the π-calculus ([10]).
- $\texttt{rec}_\texttt{p}$: obtained by adding recursion given with formal parameters, but no free variables in procedure bodies.
- $\texttt{rec}_\texttt{i}$: same as $\texttt{rec}_\texttt{p}$, but where the actual parameters in recursive calls are identical to the formal parameters.
- $\texttt{rec}_\texttt{d}$: obtained by adding procedures without parameters, but with free variables in procedure bodies, with dynamic scoping.
- $\texttt{rec}_\texttt{s}$: same as $\texttt{rec}_\texttt{d}$ but with static scoping.

- $\texttt{rec}_0$: recursion given by procedures without parameters and with no free variables in procedure bodies.

The expressive power of these process languages is compared relatively to the standard notion of input-output behavior ([17, 12]) for tcc processes. Namely, one language is considered at least as expressive as another if the input-output behavior expressed by a process in the latter can be expressed also by a process in the former. Our comparison results can be summarized as follows:

- $\texttt{rec}_\texttt{p}$ and $\texttt{rec}_\texttt{d}$ are equally expressive, and strictly more expressive than the other tcc languages,

- $\texttt{rep}$, $\texttt{rec}_\texttt{s}$ and $\texttt{rec}_\texttt{i}$ are equally expressive, and strictly more expressive than $\texttt{rec}_0$.

We actually show a strong separation result between $\texttt{rec}_\texttt{p}/\texttt{rec}_\texttt{d}$ and $\texttt{rep}/\texttt{rec}_\texttt{s}/\texttt{rec}_\texttt{i}$, namely that input-output equivalence is undecidable for the languages in the the first class, even if we fix an underling constraint system with a finite domain, but decidable for the languages in the second class for arbitrary constraint systems. The undecidability result is obtained by a reduction from the Post's correspondence problem ([13]). The decidability result is obtained by a reduction to Büchi automata ([2]) following similar results in [17] and [11] establishing the finite-state representability of $\texttt{rep}$ processes.

The expressiveness gaps illustrated above may look surprising to those acquainted with the π-calculus, because the π-calculus correspondents of $\texttt{rep}, \texttt{rec}_\texttt{i}$ and $\texttt{rec}_\texttt{p}$ have all the same expressive power. Our interpretation of this difference is that the π-calculus has some powerful mechanisms (synchronous communication and mobility) which compensate for the weakness of replication and of the lower forms of recursion.

The paper is structured as follows. The first section is devoted to describing the semantics of the various tcc languages. Section 3 first introduces the equivalences and their corresponding congruences arising from the input-output behavior and the output behavior on the empty input. Then it states the relationship between the equivalences and their congruences for the various languages. Section 4 presents the undecidability of the input-output equivalence for $\texttt{rec}_\texttt{p}$ processes in a finite-domain constraint system. Section 5 presents the decidability of the input-output equivalence for $\texttt{rep}$ processes in arbitrary constraint systems. Finally, Section 6 presents encodings preserving the input-output semantics, and the classification of the tcc languages as stated above.

2 TCC Languages

In this section we describe the various tcc languages. We shall use the syntax of (the deterministic fragment of) the ntcc calculus introduced in [12].

2.1 Constraint Systems.

Concurrent constraint languages are parameterized by a *constraint system.* A constraint system provides a signature from which syntactically denotable objects called *constraints* can be constructed, and an entailment relation $\vdash$ specifying interdependencies between such constraints. For our purposes it will suffice to consider the notion of constraint system based on First-Order Predicate Logic, as it was done in [22] [1].

Definition 2.1. A *constraint system* is a pair (Σ, Δ) where Σ is a signature specifying constant, functions and predicate symbols, and Δ is a consistent first-order theory over Σ.

Given a constraint system (Σ, Δ), let $\mathbf{L}$ be the underlying first-order language $(\Sigma, \mathcal{V}, \mathcal{S})$, where $\mathcal{V} = \{x, y, z, \ldots\}$ is a countable set of variables and $\mathcal{S}$ is the set of logical symbols including $\wedge$, $\vee$, $\Rightarrow$, $\exists$, $\forall$, `true` and `false` which denote logical conjunction, disjunction, implication, existential and universal quantification, and the always true and false predicates, respectively. *Constraints,* denoted by $c, d, \ldots$ are first-order formulae over $\mathbf{L}$. We use $fv(c)$ and $bv(c)$ to designate the set of *free* and *bound* variables of c, respectively. We say that c *entails* d in Δ, written $c \vdash d$, if the formula $c \Rightarrow d$ holds in all models of Δ. As usual, in this paper *we shall require* $\vdash$ *to be decidable.*

We say that c is equivalent to d, written $c \approx d$, iff $c \vdash d$ and $d \vdash c$. Henceforth, $\mathcal{C}$ is the set of constraints modulo $\approx$ in (Σ, Δ).

The following is a very simple finite-domain constraint system.

Definition 2.2 (Finite-Domain Constraint System). Let $n > 0$. Define $\mathbf{FD}[n]$ as the constraint system s.t.

- Σ is given by the constants symbols $0, 1, ..., n-1$ plus the equality $=$ and

[1]See [18] for a more general notion of constraints based on Scott's information systems.

- Δ is given by the axioms for equality ([20]) $x = x$, $x = y \Rightarrow y = x$, $x = y \wedge y = z \Rightarrow x = z$ plus $v = w \Rightarrow \texttt{false}$ for each two different constants v, w in Σ.

Intuitively $\mathbf{FD}[n]$ provides a theory of variables ranging over a finite domain of values $\{0, \ldots, n-1\}$ with syntactic equality over these values. We shall use $\mathbf{FD}[n]$ as the underlying constraint system in the examples and our undecidability results.

2.2 Finite Processes

Processes $P, Q, \ldots \in \mathit{Proc}$ are built from constraints in $c \in \mathcal{C}$ and variables $x \in \mathcal{V}$ in the underlying constraint system. The processes that define *finite behavior* are given by the following syntax:

$$\begin{array}{rcllll} P, Q & ::= & \mathbf{skip} & \mid \ \mathbf{tell}(c) & \mid & \mathbf{when}\ c\ \mathbf{do}\ P \\ & \mid & P \parallel Q & \mid \ \mathbf{next}\, P & \mid & (\mathbf{local}\, x)\, P \\ & \mid & (\mathbf{local}\, x)\, P & & & \end{array}$$

Process **skip** does nothing. Process $\mathbf{tell}(c)$ adds the constraint c to the current store, thus making c available to other processes in the current time interval. Process **when** c **do** P performs the action of asking c in the current time interval. If during the current time interval this information can eventually be inferred from the store d (i.e., $d \vdash c$) then process P is executed within the same time interval, otherwise the process is precluded from execution. Process $P \parallel Q$ represents the parallel composition of P and Q. In one time unit (or interval) P and Q operate concurrently, communicating through the store. We shall use $\prod_{i \in I} P_i$, where I is finite, to denote the parallel composition of all P_i.

Process $(\mathbf{local}\, x)\, P$ behaves like P, except that all the information on x produced by P can only be seen by P and the information on x produced by other processes cannot be seen by P. We then say that $(\mathbf{local}\, x)\, P$ *binds* x *in* P. Given a process Q, we can define, in the standard way, its *bound variables* $\mathit{bv}(Q)$ as the set of variables with a bound occurrence in Q, and its *free variables* $\mathit{fv}(Q)$ as the set of variables with a non-bound occurrence in Q. We use $(\mathbf{local}\, x_1 x_2 \ldots x_n)\, P$ as an abbreviation of $(\mathbf{local}\, x_1)\, (\mathbf{local}\, x_2)\, \ldots (\mathbf{local}\, x_n)\, P$.

The only move of $\mathbf{next}\, P$ is a unit-delay for the activation of P. The process $\mathbf{unless}\ c\ \mathbf{next}\, P$ is similar, but P will be activated only if c cannot be eventually inferred from the store during the current time interval. We use $\mathbf{next}^n(P)$ as an abbreviation for $\mathbf{next}(\mathbf{next}(\ldots(\mathbf{next}\,P)\ldots))$, where **next** is repeated n times.

2.3 Semantics of Finite Process

Operationally, the current information is represented as a constraint $c \in \mathcal{C}$, so-called *store*. Following standard lines ([18]), we extend the syntax with a construct $(\mathbf{local}\,x, d)\,P$ which represents the evolution of a process of the form $(\mathbf{local}\,x)\,Q$, where d is the local information (or private store) produced during this evolution. Initially d is "empty", so we regard $(\mathbf{local}\,x)\,P$ as $(\mathbf{local}\,x, \mathtt{true})\,P$.

The operational semantics will be given in terms of the reduction relations $\longrightarrow, \Longrightarrow \subseteq Proc \times \mathcal{C} \times Proc \times \mathcal{C}$ defined in Table 1. The *internal transition*

$$\langle P, c\rangle \longrightarrow \langle Q, d\rangle$$

should be read as "P with store c reduces, in one internal step, to Q with store d ". The *observable transition*

$$P \xRightarrow{(c,d)} Q$$

should be read as "P on input c from the environment, reduces in one time unit to Q and outputs d to the environment". Process Q is the process to be executed in the next time unit. Such a reduction is obtained from a sequence of internal reduction starting in P with initial store c and terminating in a process Q' with store d. Crudely speaking, Q is obtained by removing from Q' what was meant to be executed only during the current time interval. In tcc the store d is not automatically transferred to the next time unit. If needed, information in d can be transfered to next time unit by process P.

Let us describe some of the rules for the internal transitions. Rules $\mathrm{R_T}, \mathrm{R_W}, \mathrm{R_{PL}}, \mathrm{R_{PR}}, \mathrm{R_U}$ follow [18] and they should be self-explanatory. Rule $\mathrm{R_U}$ says that if c is entailed by the current store, then the execution of the process P (in the next time interval) is precluded.

Rule $\mathrm{R_L}$ is the standard rule for locality (or hiding) in Concurrent Constraint Programming (see [18, 4]). This rules deserves further explanation as it plays a key role in the results of this paper. Consider the process $Q = \mathbf{local}\ (x, c)\ \mathbf{in}\ P$. We distinguish between the *external* (corresponding to Q) and the *internal* point of view (corresponding to P). From the internal point of view, the information about x, possibly appearing in the "global" store d, cannot be observed. Thus, before reducing P we should first hide the information about x that Q may have in d. We can do this by existentially quantifying x in d. Similarly, from the external point of view, the observable information about x that the reduction of internal agent P may produce (i.e., c') cannot be observed. Thus we hide it by existentially quantifying x in c' before adding it to the global store corresponding to the evolution of Q. Additionally, we should make c' the new private store of the evolution of the internal process for its future reductions.

Let us now describe the rule for the observable transitions. Rule $\mathrm{R_O}$ says that an observable transition from P labeled by (c, d) is obtained by performing a terminating sequence of internal transitions from $\langle P, c\rangle$ to $\langle Q, d\rangle$, for some Q. The process to be executed in the next time interval, $F(Q)$ ("future" of Q), is obtained by removing from Q "when" processes which could not be executed during the current time interval and any local information which has been stored in Q, and by "unfolding" the sub-terms within $\mathbf{next}\,R$ expressions. More precisely:

Definition 2.3 (Future Function). Let $F : \mathit{Proc} \rightharpoonup \mathit{Proc}$ be defined by

$$F(P) = \begin{cases} \mathbf{skip} & \text{if } P = \mathbf{skip} \\ \mathbf{skip} & \text{if } P = \mathbf{when}\ c\ \mathbf{do}\ Q \\ F(P_1) \parallel F(P_2) & \text{if } P = P_1 \parallel P_2 \\ (\mathbf{local}\,x)\,F(Q) & \text{if } P = (\mathbf{local}\,x, c)\,Q \\ Q & \text{if } P = \mathbf{next}\,Q \\ Q & \text{if } P = \mathbf{unless}\ c\ \mathbf{next}\,Q \end{cases}$$

Remark. Function F is not total, but this is not a problem since whenever we need to apply F to a P (Rules $\mathrm{R_O}$ in Table 1), all the sub-processes of P not considered in the definition of F will occur within a "next" or "unless" expression.

In the following sections we consider several ways in which tcc languages can express infinite behavior through the time intervals.

2.4 Replication

One simple way to express infinite behavior in tcc is by using a replicator operator as in [12] and [5][2]. Let us extend the syntax of processes as

[2]More precisely, [5] uses the **hence** operator. However, $\mathbf{hence}\,P$ is equivalent to $\mathbf{next}\,!\,P$ and, similarly $!\,P$ is equivalent to $P \parallel \mathbf{hence}\,P$.

follows.

$$P := \ldots \mid\ !P \tag{1}$$

the operator "!" represents a delayed version of the replication operator of the $\pi-$calculus ([10]): $!\,P$ represents $P \parallel \mathbf{next}\,P \parallel \mathbf{next}^2 P \parallel \ldots$, i.e. unboundedly many copies of P but one at a time. We shall use `rep` to denote the language using this operator for infinite behavior.

The operational semantics of `rep` is obtained by adding to the rules in Table 1 the rule for replication:

$$\mathrm{R_{REP}}\ \frac{}{\langle !\,P, c\rangle \ \longrightarrow\ \langle P \parallel \mathbf{next}\,!\,P, c\rangle} \tag{2}$$

Rule $\mathrm{R_{REP}}$ specifies that the process $!\,P$ produces a copy P at the current time unit, and then persists in the next time unit.

2.5 Recursion

An alternative to define infinite behavior in tcc languages is by using recursion as it was done in [15, 16, 23] . We extend the syntax of finite processes by:

$$P := \ldots \mid A(y_1, \ldots, y_n) \tag{3}$$

Process $A(y_1, \ldots, y_n)$ is an *identifier* with arity n. We assume that every such an identifier has a (recursive) *definition* $A(x_1, \ldots, x_n) \stackrel{\mathtt{def}}{=} P$ where the x_i's are pairwise distinct, and the intuition is that $A(y_1, \ldots, y_n)$ behaves as P with y_i replacing x_i for each i. We presuppose an underlying set of definitions $\mathcal{D}$. We shall often use the notation $\vec{x}$ as an abbreviation of $x_1, x_2, \ldots, x_n$ if n is unimportant or obvious. We shall sometimes say that $A(\vec{y})$ is an *invocation* with *actual parameters* $\vec{y}$ and given $A(\vec{x}) \stackrel{\mathtt{def}}{=} P$ we shall refer to P as its *body* and to $\vec{x}$ as its *formal parameters*

Following [15] we require any process to depend only on finitely many definitions and recursion to be "next" guarded. For example, given $A(\vec{x}) \stackrel{\mathtt{def}}{=} P$, every invocation $A(\vec{y})$ in P must occur within the scope of a "next" or "unless" operator operator. This avoids non-terminating sequences of internal reductions (i.e., non-terminating computation within a time interval).

We can formalize the two requirements above as follows. Given $A_1(\vec{x}_1) \stackrel{\mathtt{def}}{=} P_1$ and $A_2(\vec{x}_2) \stackrel{\mathtt{def}}{=} P_2$ we say that A_1 (directly) *depends* on A_2, written $A_1 \rightsquigarrow A_2$, if there is an invocation $A_2(\vec{y})$ in P_1. The first requirement can be then formalized by requiring the strict ordering induced by $\rightsquigarrow^*$ (the reflexive and transitive closure of $\rightsquigarrow$)[3] to be well founded. For the second requirement, suppose that $A_1 \rightsquigarrow A_2 \rightsquigarrow \ldots \rightsquigarrow A_n \rightsquigarrow A_{n+1} = A_1$, where $A_i(\vec{x}_1) \stackrel{\texttt{def}}{=} P_i$. We shall require that for at least one i, $1 \leq i \leq n$, the occurrences of A_{i+1} in P_i are within the scope of a "next" or an "unless" operator.

Furthermore, for the simplicity of the presentation let us assume that *the free variables in definitions' bodies are formal parameters.* More precisely, for each $A(x_1, \ldots, x_n) \stackrel{\texttt{def}}{=} P$, we have $fv(P) \subseteq \{x_1, \ldots, x_n\}$. This requirement is imposed on the recursive versions of the π-calculus.

We shall use $\texttt{rec}_\texttt{p}$ to denote the tcc language with recursion with the above syntactic restriction. The operational rules for $\texttt{rec}_\texttt{p}$ are obtained by adding to the rules in Table 1 the rule for recursion:

$$\mathrm{R}_{\mathrm{REC}}\ \frac{A(\vec{x}) \stackrel{\texttt{def}}{=} P \quad \langle P[\vec{y}/\vec{x}] \rangle \longrightarrow \langle P', d' \rangle}{\langle A(\vec{y}), d \rangle \longrightarrow \langle P', d' \rangle} \tag{4}$$

As usual $P[y_1 \ldots y_n / x_1 \ldots x_n]$ is the process that results from syntactically replacing every free occurrence of x_i by y_i using α-conversion wherever needed to avoid capture.

2.5.1 Identical Parameters Recursion.

An interesting tcc language considered in [15] arises from $\texttt{rec}_\texttt{p}$ by requiring the parameters not to change through recursive invocations. In the π-calculus this restriction does not cause any loss of expressive power since such form of recursion can encode replication and replication can encode general recursion (see [10]).

An example satisfying this restriction on recursion is $R_P(\vec{x}) \stackrel{\texttt{def}}{=} P \parallel \mathbf{next}\, R_P(\vec{x})$. Here the actual parameters of the invocation in the definition's body are the same as the formal parameters of R_P. An example not satisfying the restriction is $R'_P(\vec{x}) \stackrel{\texttt{def}}{=} P \parallel \mathbf{next}\, (\mathbf{local}\, \vec{x})\, R'_P(\vec{x})$. Here the actual parameters, although syntactically the same, are bound and therefore different from those of the formal parameters. One can generalize this for a set of mutually recursive definitions as follows. Suppose

[3]The relation $\rightsquigarrow^*$ is a preordering. By induced strict ordering we mean the strict component of $\rightsquigarrow^*$ modulo the equivalence relation obtained by taking the symmetric closure of $\rightsquigarrow^*$.

that $A_1 \leadsto A_2$ and $A_2 \leadsto^* A_1$ with $A_1(\vec{x}_1) \stackrel{\texttt{def}}{=} P_1$ and $A_2(\vec{x}_2) \stackrel{\texttt{def}}{=} P_2$ in the underlying set of definitions $\mathcal{D}$. Then for each invocation $A_2(\vec{y})$ in P_1 we should require $\vec{y} = \vec{x}_2$ and $\vec{y} \notin bv(P_1)$. In other words the actual parameters of the invocation A_2 in P_1 (i.e., $\vec{y}$) should be syntactically the same as its formal parameters (i.e., $\vec{x_2}$). Furthermore, they should not be bound in P_1 to avoid cases such as $R'_P(\vec{x})$ above. The processes of tcc with identical parameters are those of $\texttt{rec}_\texttt{p}$ that satisfy this requirement. We shall refer to this language as $\texttt{rec}_\texttt{i}$.

2.6 Parameterless Recursion.

Tcc languages with parameterless recursion have been considered in [15] and [16]. We shall refer to identifiers with arity zero and their corresponding definitions as *constant* identifiers and *constant* definitions, respectively. We omit the "( )" in $A(\,)$.

Given a parameterless definition $A \stackrel{\texttt{def}}{=} P$, requiring all variables in $fv(P)$ to be formal parameters, as we did in $\texttt{rec}_\texttt{p}$, would be too restrictive. This would mean that the body P has no free variables and processes in ccp communicate through free variables. For example, it would be impossible to define the process that every two time units tells $x = 1$. Consequently, let us consider a fragment allowing only parameterless recursion *with free variables in the bodies of constant definitions.*

Now assuming that the operational rules for parameterless recursion are the same as for $\texttt{rec}_\texttt{p}$, one may wonder about the scope of the free variables in definitions bodies. Is it some kind of *dynamic* scoping similar to that of CCS ([9]) and, most notably, as it is in the standard model of concurrent constraint programming ([18])? Is it *static* as in most programming languages?.

The next section answers this question. Let us first illustrate what we mean by dynamic and static scoping.

Example 2.1. Consider a constant identifier A with the following definition

$$\begin{array}{rl} A \stackrel{\texttt{def}}{=} & \textbf{tell}(x = 1) \\ \parallel & \textbf{next}\,(\textbf{local}\,x)\,(A \parallel \textbf{when}\ x = 1\ \textbf{do}\ \textbf{tell}(z = 1)) \end{array}$$

In the case of dynamic scoping, an outside invocation A causes the execution $\textbf{tell}(z = 1)$ in the second time interval. The reason is that $(\textbf{local}\,x)$ binds the x resulting from the unfolding of the A inside the definition's body[4]. In fact, the telling of $x = 1$, in the second time unit, will not be visible in the store. In the case of static scoping, $(\mathbf{local}\, x)$ does not bind the x of the unfolding of A because such an x is intuitively a “global” variable, and hence $\mathbf{tell}(z = 1)$ will not be executed. In fact, the telling of $x = 1$, will also be visible in the store in the second time interval. $\square$

2.6.1 Parameterless Recursion with Dynamic Scoping

Rule $\mathrm{R_L}$ combined with $\mathrm{R_{REC}}$ causes the parameterless recursion to have dynamic scoping[5]. As illustrated in the example below, the idea is that since $(\mathbf{local}\, x)\, P$ reduces to a process of the form $(\mathbf{local}\, x)\, Q$, the x's occurring free in the unfolding of invocations in P get bounded. We shall refer to the language allowing only parameterless recursion with free-variables in the procedure bodies as $\mathtt{rec_d}$; parameterless recursion with dynamic scoping.

Example 2.2. Let A as defined in Example 2.1. Let us abbreviate the definition as $A \overset{\mathsf{def}}{=} \mathbf{tell}(x = 1) \parallel P$. We have the following reduction of $(\mathbf{local}\, x)\, A$ on store $\mathtt{true}$.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\langle \mathbf{tell}(x=1), \mathtt{true}\rangle \longrightarrow \langle \mathbf{skip}, x=1\rangle}\ \mathrm{R_T}}{\langle \mathbf{tell}(x=1) \parallel P, \mathtt{true}\rangle \longrightarrow \langle \mathbf{skip} \parallel P, x=1\rangle}\ \mathrm{R_{PL}}}{\langle A, \mathtt{true}\rangle \longrightarrow \langle \mathbf{skip} \parallel P, x=1\rangle}\ \mathrm{R_{REC}}}{\langle (\mathbf{local}\, x, \mathtt{true})\, A, \mathtt{true}\rangle \longrightarrow \langle (\mathbf{local}\, x, x=1)\, (\mathbf{skip} \parallel P), \mathtt{true}\rangle}\ \mathrm{R_L}$$

Thus $(\mathbf{local}\, x)\, A$ in store $\mathtt{true}$ reduces to $(\mathbf{local}\, x, x = 1)\, (\mathbf{skip} \parallel P)$ in store $\mathtt{true}$. Notice that the free x in A's body become local to $(\mathbf{local}\, x, x = 1)\, (\mathbf{skip} \parallel P)$, i.e, it now occurs in the local store but not in the global one. $\square$

Remark. It should be noticed that, unlike in $\mathtt{rec_p}$, we cannot freely α-convert processes in $\mathtt{rec_d}$ without changing behavior. For example, we could α-convert the $(\mathbf{local}\, x)\, A$ in the above example into $(\mathbf{local}\, z)\, A$ (since $A[z/x]$ is syntactically equal to A) but the behavior of $(\mathbf{local}\, z)\, A$ would not be the same as that of $(\mathbf{local}\, x)\, A$. We could solve this problem by defining the substitutions $[z/x]$ to be relabeling functions as in CCS instead of syntactic replacements. We can see in Table 1, however, that

[4]Just as in the CCS definition $A \overset{\mathsf{def}}{=} a.\mathbf{O} \parallel \tau.(A \parallel \bar{a}.\mathbf{O})\backslash a$, process $\bar{a}.\mathbf{O}$ can communicate through a with the unfolding of A.

[5]Rules $\mathrm{R_L}$ and $\mathrm{R_{REC}}$ are the same in ccp, hence the observations made in this section regarding dynamic scoping apply to ccp as well.

no syntactic substitutions will be applied in the reductions of $\mathtt{rec_d}$ as this deals only with constant definitions. Therefore, the operational semantics in $\mathtt{rec_d}$ does not appeal to α-conversion.

2.6.2 Parameterless Recursion with Static Scoping

From the previous section it follows that if we want to have static scoping as in [15] we should replace the rule for local behavior R_L .

Rule R'_{L} defines locality for the parameterless recursion with static scoping language henceforth referred to as $\mathtt{rec_s}$.

$$\mathrm{R}'_{\mathrm{L}}\ \frac{\langle P[y/x], d\rangle \longrightarrow \langle P', d'\rangle \quad y \text{ is fresh}}{\langle (\mathbf{local}\, x)\, P, d\rangle \longrightarrow \langle P', d'\rangle} \tag{5}$$

As in [8], we use the notion of *fresh* variable meaning that it does not occur elsewhere in a process, definition or the store. It will be convenient to presuppose that the set of variables $\mathcal{V}$ is partitioned into two infinite sets $\mathcal{F}$ and $\mathcal{V} - \mathcal{F}$. We shall assume that the fresh variables are taken from $\mathcal{F}$ and that no input from the environment or process, other than the ones generated when applying R'_{L}, can contain variables in $\mathcal{F}$.

The fresh variables introduced by R'_{L} are not to be visible from the outside. We hide these fresh variables, as it is done in [17], by using existential quantification in the output constraint of observable transitions. More precisely, we replace the rule for the observable transitions $\mathrm{R_O}$ with the rule

$$\mathrm{R}'_{\mathrm{O}}\ \frac{\langle P, c\rangle \longrightarrow^* \langle Q, d\rangle \not\longrightarrow}{P \xRightarrow{(c, \exists_{\mathcal{F}} d)} F(Q)} \tag{6}$$

where $\exists_{\mathcal{F}} d$ represents the constraint resulting from the existential quantification in d of free occurrences of variables in $\mathcal{F}$.

In order to see why R'_{L} causes static scoping in $\mathtt{rec_s}$, suppose that P in Rule R'_{L} in Equation 5 contains an invocation A with $A \stackrel{\mathtt{def}}{=} R$. When replacing x with y in P, A remains the same since $A[y/x]$ is A. Furthermore, since y is chosen from $\mathcal{F}$, there will be no capture of free variables in R when unfolding A. This causes the scoping to be static. Let us illustrate this by revisiting the previous example.

Example 2.3. Let A and P as in the previous example. We have the following reduction of $(\mathbf{local}\, x)\, A$ in store $\mathtt{true}$.

$$\dfrac{\dfrac{\dfrac{\overline{\langle \mathbf{tell}(x=1), \mathtt{true}\rangle \;\longrightarrow\; \langle \mathbf{skip}, x=1\rangle}\ \mathrm{R_T}}{\langle \mathbf{tell}(x=1) \parallel P, \mathtt{true}\rangle \;\longrightarrow\; \langle \mathbf{skip} \parallel P, x=1\rangle}\ \mathrm{R_{PL}}}{\langle A, \mathtt{true}\rangle \longrightarrow \langle \mathbf{skip} \parallel P, x=1\rangle}\ \mathrm{R_{REC}}}{\langle (\mathbf{local}\, x)\, A, \mathtt{true}\rangle \;\longrightarrow\; \langle \mathbf{skip} \parallel P, x=1\rangle}\ \mathrm{R'_L}$$

Thus $(\mathbf{local}\, x)\, A$ in store $\mathtt{true}$ reduces to $\mathbf{skip} \parallel P$ in store $(x = 1)$ making the free x in A's body, as oppose to the previous example, visible in the "global" store . $\square$

Remark. Notice that, as in $\mathtt{rec_d}$, in $\mathtt{rec_s}$ we do not need α-conversion since in the reductions of $\mathtt{rec_s}$ we only use syntactic replacements of variables by fresh variables (i.e., there will not be captures).

2.7 Summary of TCC Languages

We described several languages based on the literature of (Timed) ccp. We have $\mathtt{rep}$ the tcc language with replication and $\mathtt{rec_p}$ the tcc language with recursion instead. A special case of $\mathtt{rec_p}$ is $\mathtt{rec_i}$ which restricts the parameters not to change through the recursive invocations. We also have the parameterless recursion languages $\mathtt{rec_d}$ and $\mathtt{rec_s}$. The former deals with dynamic-scoping while the later deals with static scoping.

For the sake of completeness, we consider here an additional language: $\mathtt{rec}_0$, the language with neither parameters nor free variables in the bodies of definitions.

Notation. Henceforward we use $\mathcal{L}$ to designate the set of tcc languages $\{\mathtt{rep}, \mathtt{rec_p}, \mathtt{rec_i}, \mathtt{rec_d}, \mathtt{rec_s}, \mathtt{rec}_0\}$. In the following sections, we shall sub-index sets and relations with the appropriate tcc language name to make it clear what is the language under consideration. For example $\longrightarrow_{\mathtt{rec_p}}$ means that the reduction under consideration is that of $\mathtt{rec_p}$. Similarly, $\mathit{Proc}_{\mathtt{rec_p}}$ denotes the set of processes in $\mathtt{rec_p}$. Often we shall omit the sub-index when it is unimportant or clear from the context.

3 Process Equivalences

In the following we use α, α' to represent elements of $\mathcal{C}^\omega$ and β to represent an element of $\mathcal{C}^*$. Notation $\beta.\alpha$ represents the concatenation of β and α.

Let us consider infinite sequence of observable transitions

$$P \;=\; P_1 \xRightarrow{(c_1,c_1')} P_2 \xRightarrow{(c_2,c_2')} P_3 \xRightarrow{(c_3,c_3')} \ldots$$

This sequence can be interpreted as an *interaction between the system P and an environment.* At the time unit i, the environment provides a *stimulus* c_i and P_i produces c_i' as *response*. We then regard (α, α') as a *reactive* observation of P. If $\alpha = c_1.c_2.c_3.\ldots$ and $\alpha' = c_1'.c_2'.c_3'\ldots$, we represent the above interaction as $P \xRightarrow{(\alpha,\alpha')}{}^{\omega}$. Given P we shall refer to the set of all its reactive observations as the *input-output behavior* of P.

Alternatively, if $\alpha = \mathtt{true}^\omega$, we can interpret the run as an *interaction among the parallel components in P without the influence of an external environment* (i.e., each component is part of the environment of the others). In this case α is called the *empty* input sequence and α' is regarded as a *timed* observation of such an interaction in P. We shall refer to the set of all timed observations of a process P as the *output* behavior of P [6].

The following definition summarizes the observables above mentioned.

Definition 3.1 (Equivalences). For each tcc language $\ell \in \mathcal{L}$ let us define

1. The input-output (or stimulus-response) relation of a process P in ℓ as
$$io_\ell(P) = \{(\alpha, \alpha') \mid P \xRightarrow{(\alpha,\alpha')}{}^{\omega}_{\ell}\}$$

2. The output relation of a process P in ℓ as
$$o_\ell(P) = \{\alpha' \mid P \xRightarrow{(\mathtt{true}^\omega,\alpha')}{}^{\omega}_{\ell}\}$$

Furthermore, define $P \sim^{io}_\ell Q$ iff $io_\ell(P) = io_\ell(Q)$ and $P \sim^{o}_\ell Q$ iff $o_\ell(P) = o_\ell(Q)$.

Let us now to consider the largest congruences included in $\sim^{io}_\ell$ and $\sim^{o}_\ell$, respectively. More precisely,

Definition 3.2. Let $\ell \in \mathcal{L}$. We define $P \approx^{io}_\ell Q$ iff for every process context $C[\cdot]$ in ℓ, $C[P] \sim^{io}_\ell C[Q]$, and similarly $P \approx^{o}_\ell Q$ iff for every process context $C[\cdot]$, $C[P] \sim^{o}_\ell C[Q]$.

[6]In [11] the term "language" instead of "output" is used. We have changed terminology to avoid confusions with "tcc language".

As usual a process context $C[\cdot]$ is a process term with a single hole such that placing a process in the hole yields a well-formed process.

The following theorem relate the equivalences and their congruences for the various tcc languages.

Theorem 3.1. *For each $\ell \in \mathcal{L}$,*

1. *If $\ell \neq \mathtt{rec_s}$ then $\approx^{io}_\ell = \approx^{o}_\ell = \sim^{io}_\ell \subset \sim^{o}_\ell$.*
2. *If $\ell = \mathtt{rec_s}$ then $\approx^{io}_\ell = \approx^{o}_\ell \subset \sim^{io}_\ell \subset \sim^{o}_\ell$.*

Proof. Here we prove (1) $\approx^{io} = \approx^{o}$ and (2) $\approx^{io}_{\mathtt{rec_s}} \subset \sim^{io}_{\mathtt{rec_s}}$. The other cases follow from results in [11].

(1) Obviously $\approx^{io} \subseteq \approx^{o}$. We want to prove that $P \approx^{o} Q$ implies $P \approx^{io} Q$. Suppose that $P \approx^{o} Q$ but $P \not\approx^{io} Q$. Then there must exist a context $C[\cdot]$ s.t $C[P] \not\sim^{io} C[Q]$. Consider the case $io(C[P]) \not\supset io(C[Q])$. Take an $\alpha = c_1.c_2\ldots$ such that $(\alpha, \alpha') \in io(C[Q])$ but $(\alpha, \alpha') \notin io(C[P])$. There must then be a prefix of α' which differs from all other prefixes of sequences α'' s.t. $(\alpha, \alpha'') \in io(C[P])$. Suppose that this is the $n-$th prefix. One can verify that for the context

$$C'[\cdot] = C[\cdot] \ \parallel \ \prod_{i \leq n} \mathbf{next}^{\,i}\, \mathbf{tell}(c_i),$$

$o(C'[P]) \neq o(C'[Q])$. This contradicts our assumption $P \approx^{o} Q$. The case $io(C[Q]) \not\supset io([P])$ is symmetric. Therefore $P \not\approx^{o} Q$ as required

(2) The inclusion is obvious. As for the proper inclusion, take $A \overset{\mathtt{def}}{=} \mathbf{tell}(c)$ with $c = (x = y)$ in any underlying constraint system with equality. Notice that for $\ell = \mathtt{rec_s}$, $A \sim^{io}_\ell \mathbf{tell}(c)$ but

$$(\mathbf{local}\, x)\, A \sim^{io}_\ell \mathbf{tell}(c) \not\sim^{io}_\ell \mathbf{skip} \sim^{io}_\ell (\mathbf{local}\, x)\, \mathbf{tell}(c).$$

$\square$

The theorem states that the input-output and output congruences coincide for all languages. It also states that the input-output behavior is a congruence for every tcc language but $\mathtt{rec_s}$. As expected (see Item (2)), the input-output behavior of an arbitrary process $(\mathbf{local}\, x)\, P$ in $\mathtt{rec_s}$ cannot be inferred from the input-output behavior of P only. This reveals a distinction between $\mathtt{rec_s}$ and the other tcc languages and, in fact, between $\mathtt{rec_s}$ and the standard model of concurrent constraint programming ([18]).

In the following sections we shall classify the tcc languages based on the decidability of their input-output equivalence.

4 Undecidability Results

In this section we first state that $\sim^{io}_{\texttt{rec}_\texttt{p}}$ is undecidable for processes with an underlying finite-domain constraint system. Recall that a finite-domain constraint system $\mathbf{FD}[n]$ (see Definition 2.2) provides a theory of variables ranging over a finite domain of values $D = \{0, 1, \ldots, n-1\}$ with syntactic equality over these values. We shall also prove a stronger version of this result establishing that $\sim^{io}_{\texttt{rec}_\texttt{p}}$ is undecidable even for the finite-domain constraint system with one single constant $\mathbf{FD}[1]$, i.e., $|D| = 1$. In sections 6 we shall give an input-output preserving encoding from $\texttt{rec}_\texttt{p}$ into the parameterless recursion language $\texttt{rec}_\texttt{d}$. Therefore, $\sim^{io}_{\texttt{rec}_\texttt{d}}$ is undecidable as well.

Our proof of undecidability will proceed by a reduction from the Post's correspondence problem (PCP)[13]. Let us recall the following definition.

Definition 4.1. A PCP *instance* is a tuple (W, V), where $W = \{w_0, \ldots, w_n\}$ and $V = \{v_0, \ldots, v_n\}$ are two set of words over the alphabet $\{0, 1\}$. A *solution* to this instance is a sequence of indexes $i_0, \ldots, i_m$ in $I = \{0, \ldots, n\}$ s.t.

$$w_{i_0}.w_{i_2} \ldots w_{i_m} = v_{i_0}.v_{i_2} \ldots v_{i_m}.$$

In the PCP we are given an instance (V, W) and we are asked whether there is a solution for such an instance. The PCP is known to be undecidable [13], even if we confine our attention to instances involving non-empty words only and to solutions where the first index is required to be 0.

Theorem 4.1. *Given $P, Q \in Proc_{\texttt{rec}_\texttt{p}}$ in a finite-domain constraint system, the question of whether $P \sim^{io}_{\texttt{rec}_\texttt{p}} Q$ or not is undecidable.*

Proof. Here we give a reduction from the PCP where the instances involve non-empty words only and the solutions are required to have 0 as their first index.

Let (V, W) be a PCP instance where $W = \{w_0, \ldots, w_n\}$ and $V = \{v_0, \ldots, v_n\}$ are sets of non-empty words. Let $\mathbf{FD}[m]$ (Definition 2.2) be the underlying constraint system where $m = max(|V|, 2)$ (i.e., we need at least two constants in the encoding below). For each $i \in I = \{0, \ldots, |V| - 1\}$, we shall a define process $A_i(b_1, b_2, index, x)$ which intuitively does the following:

1. It waits until is told that $b_1 = 1$ to start writing w_i, one symbol per time unit. Each such a symbol, say s, will be written in x by telling $x = s$. Similarly, it waits until $b_2 = 1$ to start writing v_i, one symbol per time unit. Each such a symbol will also be written in x.

2. It spawns a process $A_j(b'_1, b'_2, \mathit{index}, x)$ when the environment inputs an index $\mathit{index} = j$ in I.

3. It sets $b_1 = 0$ and $b'_1 = 1$ when it finishes writing w_i, i.e., $|w_i|$ time units later after it started writing (this way it announces that its job of writing w_i is done, and allows A_j to start writing w_j). Similarly, it sets $b_2 = 0$ and $b'_2 = 1$ when it finishes writing v_i.

4. It aborts unless the environment provides an *index* in I. It also aborts if an inconsistency arises: Either two symbols (one from a W word and another from a V word) are written in x in the same time unit and they do not match (thus generating `false`), or the environment itself inputs `false`.

Thus, intuitively the A_i's keep writing W and V words, as the environment dictates, as long as the symbols match and the environment keeps providing indexes in I at each time unit.

We use the following constructs:

$$W_{c,P}(\vec{x}) \stackrel{\mathtt{def}}{=} \textbf{when } c \textbf{ do } P \parallel \textbf{unless } c \textbf{ next } W_{c,P}(\vec{x})$$
$$R_Q(\vec{y}) \stackrel{\mathtt{def}}{=} P \parallel \textbf{next } R_Q(\vec{y})$$

where $\mathit{fv}(P) \cup \mathit{fv}(c) = \{\vec{x}\}$ and $\mathit{fv}(Q) = \{\vec{y}\}$. The former waits until c holds and then it triggers P. The latter repeats Q at each time (like the "!" operator in `rep`). We use the more readable notation **wait** c **do** P and **repeat** Q for $W_{c,P}(\vec{x})$ and $R_Q(\vec{y})$, respectively.

Below we define $A_i(b_1, b_2, \mathit{index}, x)$ for each $i \in I$ according to Items 1-4. The local variable *ichosen* is used as flag to check whether the environment input an index.

```
$A_i(b_1, b_2, index, x) \stackrel{\texttt{def}}{=} (\mathbf{local}\, b'_1\ b'_2\ ichosen)\,($
  $\mathbf{wait}\ b_1 = 1\ \mathbf{do}\ (W_i(x)$
                $\parallel \mathbf{next}\,^{|w_i|}(\mathbf{tell}(b_1 = 0) \parallel \mathbf{tell}(b'_1 = 1)))$
$\parallel \mathbf{wait}\ b_2 = 1\ \mathbf{do}\ (V_i(x)$
                $\parallel \mathbf{next}\,^{|v_i|}(\mathbf{tell}(b_2 = 0) \parallel \mathbf{tell}(b'_2 = 1)))$
$\parallel \prod_{j \in I} \mathbf{when}\ index = j\ \mathbf{do}\ (\mathbf{tell}(ichosen = 1)$
                $\parallel \mathbf{next}\, A_j(b'_1, b'_2, index, x))$
$\parallel Abort(ichosen))$
```

Process $W_i(x)$ writes, one by one, the w_i symbols in x (notation $w_i(n)$ denotes the $n-$th element of w_i). Process $V_i(x)$ is defined analogously.

$$W_i(x) \stackrel{\texttt{def}}{=} \prod_{0 \le k \le |w_i|-1} \mathbf{next}\,^{k}\mathbf{tell}(x = w_i(k)),$$

$$V_i(x) \stackrel{\texttt{def}}{=} \prod_{0 \le k \le |v_i|-1} \mathbf{next}\,^{k}\mathbf{tell}(x = v_i(k))$$

Process *Abort* aborts, according to Item 4 above, by telling `false` thereafter (thus creating a constant inconsistency).

```
$Abort(ichosen) \stackrel{\texttt{def}}{=}$
$\parallel \mathbf{unless}\ ichosen = 1\ \mathbf{next}\, \mathbf{repeat}\ \mathbf{tell}(\texttt{false})$
$\parallel \mathbf{when}\ \texttt{false}\ \mathbf{do}\ \mathbf{repeat}\ \mathbf{tell}(\texttt{false})$
```

Let us now define a process $B_i(b_1, b_2, index, x, ok)$ for each $i \in I$ that behaves exactly like $A_i(b_1, b_2, index, x)$, but in addition it outputs $ok = 1$ if it stops writing v_i and w_i exactly in the same time interval. This happens when b_1 and b_2 are set to zero in the same unit and it will imply that a solution of the form $w_{i_0}.\dots.w_i = v_{i_0}.\dots.v_i$ for the PCP (V, W) has been found.

```
$B_i(b_1, b_2, index, x, ok) \stackrel{\texttt{def}}{=} (\mathbf{local}\, b'_1\ b'_2\ ichosen)\,($
  $\mathbf{wait}\ b_1 = 1\ \mathbf{do}\ (W_i(x)$
                $\parallel \mathbf{next}\,^{|w_i|}(\mathbf{tell}(b_1 = 0) \parallel \mathbf{tell}(b'_1 = 1)))$
$\parallel \mathbf{wait}\ b_2 = 1\ \mathbf{do}\ (V_i(x)$
                $\parallel \mathbf{next}\,^{|v_i|}(\mathbf{tell}(b_2 = 0) \parallel \mathbf{tell}(b'_2 = 1)))$
$\parallel \prod_{j \in I} \mathbf{when}\ index = j\ \mathbf{do}\ (\mathbf{tell}(ichosen = 1)$
                $\parallel \mathbf{next}\, B_j(b'_1, b'_2, index, x, ok))$
$\parallel Abort(ichosen)$
$\parallel \mathbf{wait}\ b_1 = 0 \wedge b_2 = 0\ \mathbf{do}\ \mathbf{tell}(ok = 1))$
```

Since we require the first index in a solution for PCW (W, V) to be 0, we define two processes $A(\mathit{index}, x)$ and $B(\mathit{index}, x, \mathit{ok})$ which trigger A_0 and B_0 as follows .

$$
\begin{aligned}
&A(\mathit{index}, x) \stackrel{\texttt{def}}{=} (\mathbf{local}\, b_1\ b_2)\,(\\
&\quad \mathbf{tell}(b_1 = 1) \parallel \mathbf{tell}(b_2 = 1) \parallel A_0(b_1, b_2, \mathit{index}, x))
\end{aligned}
$$

$$
\begin{aligned}
&B(\mathit{index}, x, \mathit{ok}) \stackrel{\texttt{def}}{=} (\mathbf{local}\, b_1\ b_2)\,(\\
&\quad \mathbf{tell}(b_1 = 1) \parallel \mathbf{tell}(b_2 = 1) \parallel B_0(b_1, b_2, \mathit{index}, x, \mathit{ok}))
\end{aligned}
$$

One can verify that the only difference between a process $A(\mathit{index}, x)$ and $B(\mathit{index}, x, \mathit{ok})$ is that the latter eventually tells $\mathit{ok} = 1$ iff there is a solution to the PCP (V, W). Therefore, $A(\mathit{index}, x) \sim^{io}_{\texttt{rec}_\texttt{p}} B(\mathit{index}, x, \mathit{ok})$ iff the answer to to PCP (W, V) is negative. It follows that $\sim^{io}_{\texttt{rec}_\texttt{p}}$ is undecidable for finite domain constraint systems. $\Box$

We now prove a stronger version of the above theorem; input-output equivalence in undecidable in $\texttt{rec}_\texttt{p}$ even if we fix the underlying constraint system to be $\mathbf{FD}[1]$, which is the finite-domain constraint system whose only constant is 0.

Theorem 4.2. *Fix* $\mathbf{FD}[1]$ *to be the underlying constraint system. The question of whether* $P \sim^{io}_{\texttt{rec}_\texttt{p}} Q$ *or not is undecidable.*

Proof. Let us consider the proof of Theorem 4.1. Let $m = \mathit{max}(|V|, 2)$. Thus every value that a variable can take is in $D = \{0, \ldots, m-1\}$. For each variable y in the process definitions let us introduce m new variables $y_0, \ldots, y_{m-1}$. Replace the declarations of y (whether as a local or as formal parameter) by the corresponding declarations of the $y_0, \ldots, y_{m-1}$. Replace each constraint $y = v$ in the process definitions with $y_v = 0$. Create an inconsistency (i.e. by telling `false`) whenever $y_v = y_w$ for any two different values v and w in D (since $(y = v \wedge y = w) = \texttt{false}$). $\Box$

From the above theorem and Theorem 3.1 we obtain the following result.

Corollary 4.1. *The input-output and output congruences* $\approx^{io}_{\texttt{rec}_\texttt{p}}$ *and* $\approx^{o}_{\texttt{rec}_\texttt{p}}$ *are undecidable for processes in the finite-domain constraint system* $\mathbf{FD}[1]$.

Notice that $\mathbf{FD}[1]$ is a very simple constraint system (i.e., only equality and one single constant). So, the undecidability results for other constraint systems providing theories with equality and an at least one constant symbol follow from Theorem 4.2. This includes almost all constraint system of interest (e.g. the Herbrand constraint system [14], the Kanh constraint [18], Enumerated Types [14] and modular arithmetic [12]) .

5 Decidability Results

In this section we establish that $\sim^{io}_{\texttt{rep}}$ is decidable for *arbitrary constraint systems.* In section 6 we shall show via encodings that `rep`, $\texttt{rec}_\texttt{i}$, $\texttt{rec}_\texttt{s}$ have the same expressive power. We then conclude that the corresponding equivalences for $\texttt{rec}_\texttt{i}$ and $\texttt{rec}_\texttt{s}$ are also decidable.

The key for our decidability result is that the transitions of processes in `rep` can be represented by finite-state machines. This follows similar results in [17] and [11].

Example 5.1. Let $Q = !!P$ with $P = \textbf{tell}(c)$. The following is an observable transition sequence in `rep`.

$$Q \xRightarrow{(c_1,d_1)} !P \parallel Q \xRightarrow{(c_2,d_2)} !P \parallel !P \parallel Q \xRightarrow{(c_3,d_3)} \dots \xRightarrow{(c_n,d_n)} \prod_n !P \parallel Q \dots$$

where for $1 \leq i \leq n$ $d_i = c_i \wedge c$.

Thus process Q has an infinite number of derivatives. This illustrates that in a transition system where states are the elements of $Proc_{\texttt{rep}}$ it is possible to have infinite paths where all states are different.

Nevertheless, from standard results in ccp ([18]), in all the tcc languages in $\mathcal{L}$ one copy of P does exactly the same job than two copies, i.e. $P \parallel P$ [7]. So $!P \parallel !P \parallel \dots \parallel !P$ and $!P$ behaves the same way. Below we establish this property more precisely.

Definition 5.1. For each tcc language $\ell \in \mathcal{L}$ define $\equiv_\ell$ as the smallest congruence satisfying the axiom $P \equiv_\ell P \parallel P$ for $P \in Proc_\ell$

The following property states that $\equiv$ is preserved by input-output congruence.

Proposition 5.1. *For each $\ell \in \mathcal{L}$, $\equiv_\ell \subset \approx^{io}_\ell$.*

[7]Notice that this does mean that $!P$ and P behave the same way.

Definition 5.2. We say that Q is a *derivative* of P if there is a sequence $P \xRightarrow{(c_1,c_2)} \dots \xRightarrow{(c_n,d_n)} Q$. Define $Der(P)$ as the set of all derivatives of P.

The following properties are used for constructing automata representing processes in `rep`.

Lemma 5.1. *Let $P \in Proc_{\mathtt{rep}}$. Then the set $Der(P)$ modulo $\equiv$ is finite.*

Proof. The proof can be established by induction on the structure of P following analogous proofs in [17] and [11]. $\square$

Proposition 5.2. *Relation $\equiv_{\mathtt{rep}}$ is decidable.*

We shall characterize the input-output behavior (see Definition 3.1) in terms of ω-regular languages, i.e., the languages accepted by Büchi automata. Recall that Büchi automata are ordinary finite-state automata equipped with an acceptance condition that is appropriate for ω-sequences: an ω-sequence is accepted if the automaton can read it from left to right while visiting a sequence of states in which some final state occurs infinitely often. This condition is called *Büchi acceptance* ([2]).

Our plan is then to construct Büchi automata for the input-output behavior in which the transitions are labeled with input-output constraints. The problem, however, is that there are infinitely many input constraints (even if the underlying constraint system is finite domain as there are infinitely many variables). In [17] is shown how to compute a set containing the "relevant" inputs for hiding-free processes in arbitrary constraint systems. Below we extend the result to arbitrary processes.

Definition 5.3. Given $S \subseteq \mathcal{C}$, let $\overline{S}$ be the closure under conjunction and implication of S. Let $C : Proc \to \mathcal{C}$ be defined as:
$C(\mathbf{skip}) = \{\mathtt{true}\}$
$C(\mathbf{tell}(c)) = \{c\}$
$C(\sum_{i \in I} \mathbf{when}\ c_i\ \mathbf{do}\ P_i) = \bigcup_{i \in I}\{c_i\} \cup C(P_i)$
$C(\mathbf{unless}\ c\ \mathbf{next}\ P) = \{c\} \cup C(P)$
$C(P \parallel Q) = C(P) \cup C(Q)$
$C(!\,P) = C(\star\, P) = C(\mathbf{next}\, P) = C(P)$
$C((\mathbf{local}\, x)\, P) = \{\exists_x c, \forall_x c \mid c \in \overline{C(P)}\}$

Define the *relevant input constraints* of P, $\mathcal{C}(P)$, as the set (modulo logical equivalence) $\overline{C(P)}$.

Definition 5.4. Let $d \in \mathcal{C}$. Define the *strongest consequence* of d in P, written $d(P)$, as the unique constraint (modulo logical equivalence) $e \in \mathcal{C}(P)$ such that $d \vdash e$ and $e \vdash e'$ for every $e' \in \mathcal{C}(P)$ such that $d \vdash e'$.

Notice that that $d(P)$ always exists since $\mathcal{C}(P)$ is closed under conjunction. Furthermore, it can be computed since $\vdash$ is decidable and $\mathcal{C}(P)$ is finite.

The next lemma intuitively states that $\mathcal{C}(P)$ indeed contains the relevant input constraints of P.

Lemma 5.2. *For all $P \in Proc_{\mathtt{rep}}$, $P \overset{(c, c\wedge d)}{\Longrightarrow} P'$ if and only if $P \overset{(c(P), d)}{\Longrightarrow} P'$.*

Proof. Here we outline the main aspects of the proof of the "only if" direction. For simplicity, we assume that P contains no nesting of local operator. Any reduction $P \overset{(c, c\wedge d)}{\Longrightarrow} Q$ is obtained from a sequence of transitions $\langle P_0, c\rangle \longrightarrow^* \langle P_n, c \wedge d\rangle \not\longrightarrow$ where $P = P_0$ and $Q = F(P_n)$. For simplicity, let us also assume that such a sequence involves the application of at least one application of $\mathrm{R_W}$ (i.e., that the execution of a "when" operator takes place). It follows that the sequence can be represented as the sequence:

$$\begin{array}{l}\langle P_0, c\rangle \longrightarrow^* \langle P_1, c \wedge c_1\rangle \longrightarrow \langle P_1', c \wedge c_1\rangle \longrightarrow^* \ldots \\ \longrightarrow^* \langle P_i, c \wedge c_i\rangle \longrightarrow \langle P_i', c \wedge c_i\rangle \\ \longrightarrow^* \langle P_{i+1}, c \wedge c_{i+1}\rangle \longrightarrow \left\langle P_{i+1}', c \wedge c_{i+1}\right\rangle \longrightarrow^* \ldots\end{array}$$

satisfying the conditions below. The reductions $\langle P_i, c \wedge c_i\rangle \longrightarrow \langle P_i', c \wedge c_i\rangle$ are obtained from a derivation whose topmost (or root) rule is either $\mathrm{R_W}$ or $\mathrm{R_U}$. In other words the reduction involves the execution of a "when" or an "unless" operator. Furthermore, each of the $\langle P_i, c \wedge c_i\rangle \longrightarrow^* \langle P_{i+1}, c \wedge c_{i+1}\rangle$ involves no application of $\mathrm{R_W}$ or $\mathrm{R_U}$.

Suppose that g_i is the constraint guard of the "when" or "unless" operator when deriving $\langle P_i, c \wedge c_i\rangle \longrightarrow \langle P_i', c \wedge c_i\rangle$. We can infer that $e_i \wedge \exists_{\vec{x}_i}(c \wedge c_i) \vdash g_i$ where $\vec{x}_i$ and e_i are a vector of at most one variable and a local store introduced by rule $\mathrm{R_L}$ (the vector can be empty and e_i can be $\mathtt{true}$ meaning that $\mathrm{R_L}$ was not applied). This implies $\exists_{\vec{x}_i}(c \wedge c_i) \vdash (e_i \Rightarrow g_i)$, and then $\exists_{\vec{x}_i}(c \wedge c_i) \vdash \forall_{\vec{x}_i}(e_i \Rightarrow g_i)$. If follows $c \wedge c_i \vdash \forall_{\vec{x}_i}(e_i \Rightarrow g_i)$ and thus $c \vdash c_i \Rightarrow \forall_{\vec{x}_i}(e_i \Rightarrow g_i)$. Let $d_i = c_i \Rightarrow \forall_{\vec{x}_i}(e_i \Rightarrow g_i)$. Notice that $d_i \wedge c_i \wedge e_i \vdash g_i$. From the definition of $\mathcal{C}(P)$ we have $d_i \in \mathcal{C}(P)$ since g_i appears within $\underline{\text{some }}(\mathbf{local}\,\vec{x})\,Q$ in P (i.e., $g_i \in C(Q)$) and the local store e_i is then in $\overline{C(Q)}$. Let $c' = \bigwedge_{i \in \{1,\ldots,n\}} d_i$. By induction on n we can show that $\langle P_0, c' \rangle \longrightarrow^* \langle P_n, d \rangle$. Trivially $c \vdash c' \in \mathcal{C}(P)$ since $\mathcal{C}(P)$ is also closed under conjunction. Hence by definition $c \vdash c(P) \vdash c'$. We can then show that $\langle P_0, c(P) \rangle \longrightarrow^* \langle P_n, d \rangle$, thus concluding the proof. $\square$

Corollary 5.1. *$(c_1.c_2.\dots \;,\; c_1 \wedge c_1'.c_2 \wedge c_2', \dots) \in io(P)$ iff $(c_1(P).c_2(P).\dots \;,\; c_1'.c_2'.\dots) \in io(P)$.*

Having found the set of relevant input constraints for a given process we can now proceed to define a finite-state automaton representing its behavior.

5.1 Input-Output Automata

Given an arbitrary process P and a finite set of (input) constraints S, we shall construct an automaton A_P^S which recognizes the input-output behavior of P *restricted to* inputs in S. The start state is P and each transition from state Q to state R with label (c, d), where $c \in S$, represents an observable reduction $Q \xRightarrow{(c,d)} R$ in `rep`. The construction is given in the proof of the following lemma which also states the language accepted by A_P^S.

Lemma 5.3. *Given $P \in Proc_{\texttt{rep}}$ and a finite set of constraints S, one can effectively construct a Büchi automaton A_P^S recognizing the set of all $(c_1, c_1').(c_2, c_2') \dots$ such that $c_1, c_2 \dots \in S^\omega$ and $(c_1, c_2 \dots \;,\; c_1'.c_2' \dots) \in io(P)$*

Proof. Here is the algorithm that constructs A_P^S. (1) Make P to be an accepting and the start state. (2) Choose a state Q from the current transition graph and compute a reduction $Q \xRightarrow{(c,d)} R$ (such computation always terminates). The choice should satisfy that there is not already an edge labeled with (c, d) from Q to some $R' \equiv R$. If such a choice is not possible then stop. (3) Else if there is already a state $R' \equiv R$ then create an edge labeled with (c, d) from Q to it. Otherwise, create a new (accepting) state R and edge from Q to it with label (c, d). (4) Go to (2).

From the finiteness of S, the decidability of $\equiv_{\texttt{rep}}$ (Proposition 5.2) and Lemma 5.1 it follows that the algorithm terminates. The partial correctness of the construction is easy to verify. $\square$

From Corollary 5.1 and the above Lemma it follows that the automaton $A_P^{\mathcal{C}(P)}$ provides a finite representation of the input-output behavior of P.

Furthermore, from the above lemma it follows that the question of whether P and Q have the same input-output behavior *restricted to* S can be reduced to whether A_P^S and A_Q^S accept the same language.

Therefore, the question of whether P and Q have the same *output* behavior can be reduced to whether A_P^S and A_Q^S with $S = \{\texttt{true}\}$ accept the same language. Since language equivalence for Büchi automata is decidable [21], we can conclude that $\sim^o_{\texttt{rep}}$ is decidable for arbitrary constraint systems.

Theorem 5.1. *For any $P, Q \in Proc_{\texttt{rep}}$ over an arbitrary constraint system the question whether or not $P \sim^o_{\texttt{rep}} Q$ is decidable.*

Similarly, by appealing to Corollary 5.1, it follows that the question of whether P and Q have the same *input-output* behavior can be reduced to whether A_P^S and A_Q^S with $S = \mathcal{C}(P) \cup \mathcal{C}(Q)$ accept the same language.

Theorem 5.2. *Given $P, Q \in Proc_{\texttt{rep}}$ over an arbitrary constraint system. The question whether or not $P \sim^{io}_{\texttt{rep}} Q$ is decidable*

From the above theorem and Theorem 3.1 we obtain the following result.

Corollary 5.2. *The input-output and output congruences $\approx^{io}_{\texttt{rep}}$ and $\approx^{o}_{\texttt{rep}}$ are decidable for processes over arbitrary constraint systems.*

These decidability results in `rep` with arbitrary constraint system are to be contrasted to the undecidability results in $\texttt{rec}_{\texttt{p}}$ with the simple finite-domain constraint system **FD**[1].

6 Classification of the tcc languages

In this section we discuss the relation between the various tcc languages, and we classify them on the basis of their expressive power.

Figure 1 shows the sub-language inclusions and the encodings preserving the input-output semantics between the various tcc versions. Classes I, II, III represent a partition based on the expressive power: two languages are in the same class if and only if they have the same expressive power. We will first discuss the separation results, and then the equivalences.

Given the encodings, which will be proved later, the separation between Classes II and III follows immediately from the results of Sections 4 and 5. From the proof of Theorem 4.1 it follows that $\texttt{rec}_\texttt{p}$ is capable of expressing the "behavior" of Post Correspondence problems, and hence clearly capable of expressing input-output behaviors not accepted by Büchi automata, and hence not in `rep` (Lemma 5.3). For example, consider the PCP instance (V, W) with $W = \{w_0 = aa, w_1 = b\}$ and $V = \{v_0 = aaa, v_1 = a\}$, where $a = 0$ and $b = 1$. Define the constraints $c_0 = (\mathit{index} = 0)$, $c_1 = (\mathit{index} = 1)$ and $d = (x = a)$. Let P be the process $A(\mathit{index}, x)$ in the proof of Theorem 4.1. It is easy to verify that P on input $c_0^n.c_1^\omega$ contributes to output with $d^{2n}.\texttt{false}^\omega$ (i.e., it outputs $(c_0 \wedge d)^n.(c_1 \wedge d)^n.\texttt{false}^\omega$). It follows from Lemma 5.3 and simple arguments from automata theory that no process in `rep` can then exhibit the input-output behavior of $A(\mathit{index}, x)$.

The separation between Classes I and II, on the other hand, follows from the fact that without parameters or free variables the recursive calls cannot communicate with the external environment, hence in $\texttt{rec}_0$ a process can produce information on variables for a finite number of time intervals only. More precisely, we have the following result:

Proposition 6.1. *For every P in $\texttt{rec}_0$, if $(c_1.c_2.c_3.\ldots, d_1.d_2.d_3.\ldots) \in io(P)$ then there exist n such that, for all $k > n$, if $\exists_x c_k = c_k$ then $\exists_x d_k = d_k$, i.e. if c_k does not contain information about x then d_k does not contain information about x either.*

In `rep`, on the contrary, it is possible to express process which produce information about certain variables indefinitely through the time intervals. For instance, the process $!\,\mathbf{tell}(x = 1)$ has an input-output sequence of the form $(\texttt{true}\,.\,\texttt{true}\,.\,\texttt{true}\,.\ldots, x = 1.x = 1.x = 1.\ldots)$.

The rest of this section is devoted to illustrate the encodings of the various tcc languages. In the following, $[\![\cdot]\!] : \ell \to \ell'$ will represent the encoding function for each pair ℓ and ℓ'. We will say that $[\![\cdot]\!]$ is *homomorphic* wrt to the parallel operator if $[\![P \parallel Q]\!] = [\![P]\!] \parallel [\![Q]\!]$, and similarly for the other operators.

6.1 Encoding $\texttt{rep} \to \texttt{rec}_\texttt{i}$

This encoding is rather simple. The idea is to replace $!\,P$ by a call to a new process identifier R_P, defined as a process that expands P and then calls itself recursively in the next time interval. The free variables of $!\,P$, $\vec{x}$, are passed as (identical) parameters. Therefore we define

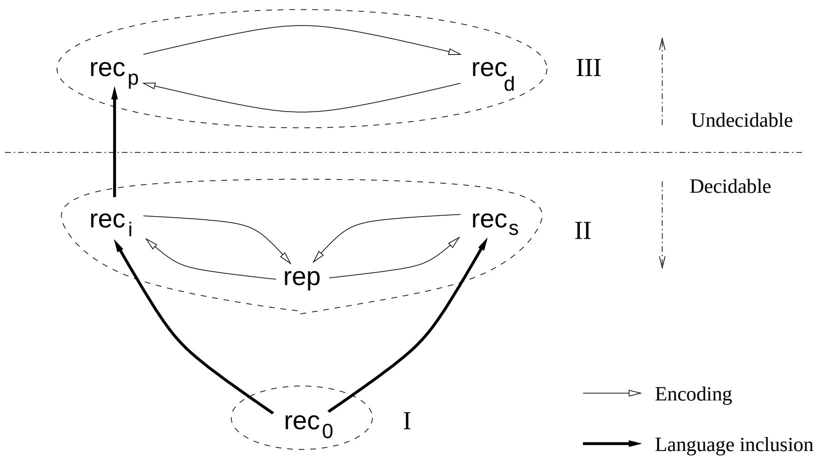

Figure 1: Classification of the various tcc languages.

$$
\begin{array}{rcl}
[\![!\, P]\!] & = & R_P(\vec{x}), \\
 & & \text{with } R_P(\vec{x}) \stackrel{\texttt{def}}{=} P \parallel \mathbf{next}\, R_P \\
 & & \text{where } \{\vec{x}\} = \mathit{fv}(P)
\end{array}
$$

and $[\![\cdot]\!]$ homomorphic on all the other operators of `rep`.

In what follows we use **repeat** P as an abbreviation of $R_P(\vec{x})$. Notice that **repeat** was already used in the proof of Theorem 4.1.

6.2 Encoding $\texttt{rec}_\texttt{s} \rightarrow \texttt{rep}$

Here the idea is to simulate a procedure definition by a replicated process that activates its body B each time there is a call for it. The activation can be done by using a construct of the form **when** c **do** B. The call, of course, will be simulated by **tell**(c).

The key case is the local operator, since we do not want to capture the free variables in the bodies of procedures. Thus, we need to α-convert the local variables with fresh variables.

In the following sections we shall use $\mathit{call}(x)$ as abbreviation of $x = 1$ (thus assuming that the underlying constraint system provides equality and at least one constant symbol). We shall also use for each identifier A, a fresh variable z_A uniquely associated to it.

We first define an auxiliary function $[\![\cdot]\!]_0 : \texttt{rec}_\texttt{s} \rightarrow \texttt{rep}$ as follows:

$$\begin{aligned}
[\![A \overset{\mathtt{def}}{=} P]\!]_0 &= \mathbf{!when}\ \mathit{call}(z_A)\ \mathbf{do}\ [\![P]\!]_0 \\
[\![A]\!]_0 &= \mathbf{tell}(\mathit{call}(z_A)) \\
[\![(\mathbf{local}\, x)\, P]\!]_0 &= (\mathbf{local}\, y)\, ([\![P]\!]_0[y/x]) \\
&\ \text{where } y \text{ is fresh}
\end{aligned}$$

and $[\![\cdot]\!]_0$ homomorphic on all the other operators of $\mathtt{rec_s}$.

Let P be an arbitrary process in $\mathtt{rec_s}$. We shall use $I(P)$ to denote the set of identifiers P depends upon (formally, $I(P)$ is the transitive closure under $\leadsto$ of the identifiers in P, see Section 2.5). The encoding $[\![\cdot]\!] : \mathtt{rec_s} \to \mathtt{rep}$ is given by

$$[\![P]\!] = (\mathbf{local}\, \vec{z})\, ([\![P]\!]_0 \parallel \prod_{1 \le i \le n} [\![A_i(\vec{x}_1) \overset{\mathtt{def}}{=} P_i]\!]_0)$$

where $I(P) = \{A_1, \dots, A_n\}$ and $\vec{z}$ are the fresh variables introduced during the translation.

6.3 Encoding $\mathtt{rec_i} \to \mathtt{rep}$

This encoding is somewhat more complex because we have to encode the passing of parameters.

A call $A(\vec{y})$ can occur in a process or in the definition of another identifier B. If there is no mutual dependency between A and B or A is a call in a process, then the actual parameters of A may be different from the formal ones, and we need to model the call by providing a copy of the replicated process that constitutes the body of A's definition and by making the appropriate parameters replacement. If, on the contrary, there is a mutual dependency between A and B (i.e. if also A depends on B) then the actual parameters coincide with the formal ones (see Section 2.5.1) and therefore we don't need to make any parameter replacement. Neither do we need to provide a copy of the replicated processes as it will be available at the top level. Note that we need this simplification in the case of mutual recursion, otherwise the translation would not terminate.

We define the auxiliary encodings $[\![\cdot]\!]_0 : \mathtt{rec_i} \to \mathtt{rep}$ for the definitions and for the calls occurring in a body, and $[\![\cdot]\!]_0^A : \mathtt{rec_i} \to \mathtt{rep}$ (where A is an identifier) for the calls occurring in a process, as follows:

$$
\begin{array}{rcl}
[\![A(\vec{x}) \overset{\texttt{def}}{=} P]\!]_0 & = & !\textbf{when } \mathit{call}(z_A) \textbf{ do } [\![P]\!]_0^A \\[1em]
[\![A(\vec{y})]\!]_0^B & = & \textbf{tell}(\mathit{call}(z_A)) \\
 & & \text{if } A \rightsquigarrow^* B \\[1em]
[\![A(\vec{y})]\!]_0^B & = & (\textbf{local}\, z_A)\, (\\
 & & \textbf{tell}(\mathit{call}(z_A)) \parallel ([\![A(\vec{x}) \overset{\texttt{def}}{=} P]\!]_0[\vec{y}/\vec{x}])) \\
 & & \text{if } A \not\rightsquigarrow^* B \\[1em]
[\![A(\vec{y})]\!]_0 & = & (\textbf{local}\, z_A)\, (\\
 & & \textbf{tell}(\mathit{call}(z_A)) \parallel ([\![A(\vec{x}) \overset{\texttt{def}}{=} P]\!]_0[\vec{y}/\vec{x}]))
\end{array}
$$

and $[\![\cdot]\!]_0$, $[\![\cdot]\!]_0^A$ homomorphic on all the other operators of $\texttt{rec}_\texttt{i}$.

The encoding of an arbitrary P in $\texttt{rec}_\texttt{i}$ into `rep` is given by

$$[\![P]\!] = (\textbf{local}\, \vec{z})\, ([\![P]\!]_0 \parallel \prod_{1 \leq i \leq n} [\![A_i(\vec{x}_i) \overset{\texttt{def}}{=} P_i]\!]_0)$$

where $I(P) = \{A_1, \ldots, A_n\}$ and $\vec{z}$ are the fresh variables introduced during the translation.

6.4 Encoding `rep` $\rightarrow$ $\texttt{rec}_\texttt{s}$

Here we take advantage of finite-state automata representation of the input-output behavior of `rep` processes given in Section 5.1.

Let P be an arbitrary process in `rep`. Let $M = A_P^{\mathcal{C}(P)}$ be the automaton representing the input-output behavior of P on the inputs of relevance for P, $\mathcal{C}(P)$ (Definition 5.3). Recall that the start state of M is P. Each transition from Q to R with label (c, d), written $\langle Q, (c, d), R\rangle$, in M represents an observable transition $Q \xRightarrow{(c,d)} R$, where $c \in \mathcal{C}(P)$.

Let T be the set of transitions of M. For each state Q of M we define an identifier A_Q as follows:

$$A_Q \overset{\texttt{def}}{=} \prod_{\langle Q,(c,d),R\rangle \in T} \textbf{when } c \textbf{ do } (\textbf{tell}(d) \parallel \textbf{unless } C \textbf{ next } A_R)$$

$$\text{where } C = \bigvee_{e \in \{c' \ \mid \ c' \neq c,\ c' \vdash c,\ \langle Q,(c',d'),R'\rangle \in T\}} e$$

Intuitively, A_Q expresses that if we are in state Q and c is the strongest constraint entailed by the the input, then the next state will be R and the output will be d, with $\langle Q, (c, d), R\rangle \in T$.

We define the encoding of P as $[\![P]\!] = A_P$.

6.5 Encoding $\texttt{rec}_\texttt{d} \longrightarrow \texttt{rec}_\texttt{p}$

Intuitively, if the free variables are treated dynamically, then they could equivalently be passed as parameters. Thus we can define the encoding as follows:

$$\begin{array}{rcl} [\![A \stackrel{\texttt{def}}{=} P]\!] & = & A(\vec{x}) \stackrel{\texttt{def}}{=} [\![P]\!], \\ & & \text{where } \vec{x} = \mathit{fv}(P) \\ \\ [\![A]\!] & = & A(\vec{x}) \end{array}$$

and $[\![\cdot]\!]$ homomorphic on all the other operators of $\texttt{rec}_\texttt{d}$.

6.6 Encoding $\texttt{rec}_\texttt{p} \longrightarrow \texttt{rec}_\texttt{d}$

The idea is to establish the link between the formal parameters $\vec{x}$ and the actual parameters $\vec{y}$ by telling the constraint $\vec{x} = \vec{y}$. However, this operation has to be encapsulated within a $(\mathbf{local}\,\vec{x})$ in order to avoid confusion with other potential occurrences of $\vec{x}$ in the same context of the call.

$$\begin{array}{rcl} [\![A(\vec{x}) \stackrel{\texttt{def}}{=} P]\!] & = & A \stackrel{\texttt{def}}{=} [\![P]\!] \\ \\ [\![A(\vec{y})]\!] & = & (\mathbf{local}\,\vec{x})\,(A \parallel \mathbf{repeat\ tell}(\vec{y} = \vec{x})) \end{array}$$

and $[\![\cdot]\!]$ homomorphic on all the other operators of $\texttt{rec}_\texttt{d}$.

6.7 Correctness of the encodings

The encodings defined in previous sections are all correct with respect to the input-output behaviors of the corresponding languages. More precisely:

Proposition 6.2. *For each encoding $[\![\cdot]\!] : \ell \rightarrow \ell'$ defined from Section 6.1 through Section 6.6 we have $io(P) = io([\![P]\!])$ for every P in ℓ.*

7 Concluding Remarks and Related Work

We have studied the expressive power of several tcc languages focusing on the decidability of their behavioral equivalences. In particular, we have shown that `rep` (i.e. tcc with replication) can be compiled into finite-state automata, while $\mathtt{rec}_\mathtt{p}$ (i.e.tcc with recursion) cannot, not even when the constraint system is based on a finite domain.

Further, we have presented behavior-preserving encodings between `rep`, $\mathtt{rec}_\mathtt{i}$ (tcc with identical parameters recursion) and $\mathtt{rec}_\mathtt{s}$ (tcc with parameterless recursion and static-scope free variables), and between `rep` and $\mathtt{rec}_\mathtt{d}$ (tcc with dynamic-scope free variables). This implies a clear distinction between dynamic and static scoping in tcc languages.

We believe that our results contribute to a better understanding of tcc languages and to clarify some conjectures made in literature. In particular, in [15] it was conjectured that $\mathtt{rec}_\mathtt{s}$ would be equivalent to $\mathtt{rec}_\mathtt{i}$ provided that definitions are allowed to be nested within the processes. Our results show that this extension is not necessary. Another consequence of our work is that the denotational semantics of $\mathtt{rec}_\mathtt{s}$ cannot be just an extension to sequences of the standard ccp construction in [18], because the semantic equations of ccp can be satisfied only by a dynamically-scoped language.

One interesting implication of our results is that, from the point of view of the expressive power, in $\mathtt{rec}_\mathtt{s}$ the **local** operator is redundant. In fact, as shown in Section 6, $\mathtt{rec}_\mathtt{s}$ can be encoded into `rep` and `rep` can be encoded into a local-free fragment of $\mathtt{rec}_\mathtt{s}$. Note that, on the contrary, locality plays a key role in the reduction of the PCP to $\mathtt{rec}_\mathtt{p}$ and in the encoding of $\mathtt{rec}_\mathtt{p}$ into $\mathtt{rec}_\mathtt{d}$.

A closely related work is [23]. Also that paper explores the expressiveness of tcc languages, but it focuses on the capability of $\mathtt{rec}_\mathtt{s}$ to encode synchronous languages. In particular, it shows that Argos ([7]) and a version of Lustre restricted to finite domains ([6]) can be encoded in $\mathtt{rec}_\mathtt{s}$. Consequently, our decidability results extend to these synchronous languages as well.

In [17] similar results show how to compile (an extension of) `rep` into finite-state automata. The fact that, in order to obtain the translation to finite-state machines, the authors restrict to `rep` already suggests some of the separation results that we have formally proved in our paper. In [17] the states are labeled with processes and transitions are labeled with output constraints rather than pair of input-output constraints. Such automata provide an execution model for $\mathtt{rec}_\mathtt{s}$ rather than a direct way of verifying input-output (or output) equivalence. In particular, the standard equivalence between two automata defined as in our construction (i.e. language equivalence) imply input-out equivalence of the processes they represent. This implication does not hold in general for the construction in [17]. Another difference wrt [17] is that we were able to compute the whole set of relevant constraints, while [17] leaves out the existentally quantified ones. This capability is a key property of our construction, because it makes it possible to translate `rep` into automata with simple states. Without it, i.e., if only a subset S of the relevant constraints could be computed, then it would probably be necessary to consider the processes in the states, like it is done in [17], to compute at run time relevant input constraints which would not be in S. It should be noticed that using the set of relevant constraints, our construction and the one in [17] (restricted to processes in `rep`) can be obtained from each other.

The tccp calculus ([3]) is another timed extension of ccp. The results in our paper do not apply automatically to that language, because tccp additionally has a nondeterministic choice, and the information about the store is carried through the time units, so the semantic setting is rather different from the languages we are considering.

A correspondence between formulae in a classic first-order linear-time logic and processes in `rep` has been established in [12]. As future work we plan to study how the results in our paper can help to establish decidability results for such a logic.

Acknowledgments

We would like to thank Maurizio Gabbrielli and Vijay Saraswat for stimulating discussion on the topics of this paper.

References

[1] G. Berry and G. Gonthier. The Esterel synchronous programming language: design, semantics, implementation. *Science of Computer Programming*, 19(2):87–152, November 1992.

[2] J. R. Buchi. On a decision method in restricted second order arithmetic. In *Proc. Int. Cong. on Logic, Methodology, and Philosophy of Science*, pages 1–11. Stanford University Press, 1962.

[3] F. de Boer, M. Gabbrielli, and M. C. Meo. A timed concurrent constraint language. *Information and Computation*, 1999. To appear.

[4] F. S. de Boer, M. Gabbrielli, E. Marchiori, and C. Palamidessi. Proving concurrent constraint programs correct. *ACM Transactions on Programming Languages and Systems*, 19(5):685–725, 1997.

[5] V. Gupta, R. Jagadeesan, and V. Saraswat. Models for concurrent constraint programming. In Ugo Montanari and Vladimiro Sassone, editors, *CONCUR '96: Concurrency Theory, 7th International Conference*, volume 1119 of *Lecture Notes in Computer Science*, pages 66–83, 26–29 August 1996.

[6] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous data-flow programming language LUSTRE. *Proceedings of the IEEE*, 79(9):1305–1320, September 1991.

[7] Florence Maraninchi. Operational and compositional semantics of synchronous automaton compositions. In W. R. Cleaveland, editor, *CONCUR '92: Third International Conference on Concurrency Theory*, volume 630 of *Lecture Notes in Computer Science*, pages 550–564, Stony Brook, New York, 24–27August 1992. Springer-Verlag.

[8] Nax P. Mendler, Prakash Panangaden, P. J. Scott, and R. A. G. Seely. A logical view of concurrent constraint programming. *Nordic Journal of Computing*, 2(2):181–220, Summer 1995.

[9] R. Milner. *Communication and Concurrency.* International Series in Computer Science. Prentice Hall, 1989. SU Fisher Research 511/24.

[10] R. Milner. *Communicating and Mobile Systems: the π-calculus.* Cambridge University Press, 1999.

[11] M. Nielsen and F. Valencia. *Temporal Concurrent Constraint Programming: Applications and Behavior*, chapter 4, pages 298–324. Springer-Verlag, LNCS 2300, February 2002.

[12] C. Palamidessi and F. Valencia. A temporal concurrent constraint programming calculus. In *Proc. of the Seventh International Conference on Principles and Practice of Constraint Programming*, 26 November 2001.

[13] E. L. Post. A variant of a recursively unsolvable problem. *Bulletion of the American Mathematical Society*, 52:264–268, 1946.

[14] V. Saraswat. *Concurrent Constraint Programming.* The MIT Press, Cambridge, MA, 1993.

[15] V. Saraswat, R. Jagadeesan, and V. Gupta. Foundations of timed concurrent constraint programming. In *Proc. of the Ninth Annual IEEE Symposium on Logic in Computer Science*, pages 71–80, 4–7 July 1994.

[16] V. Saraswat, R. Jagadeesan, and V. Gupta. Default timed concurrent constraint programming. In *Proc. of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 272–285, January 1995.

[17] V. Saraswat, R. Jagadeesan, and V. Gupta. Timed default concurrent constraint programming. *Journal of Symbolic Computation*, 22(5–6):475–520, November–December 1996.

[18] V. Saraswat, M. Rinard, and P. Panangaden. The semantic foundations of concurrent constraint programming. In *POPL '91. Proceedings of the eighteenth annual ACM symposium on Principles of programming languages*, pages 333–352, 21–23 January 1991.

[19] Vijay Saraswat, Radha Jagadeesan, and Vinheet Gupta. Programming in timed concurrent constraint languages. In B. Mayoh, E. Tyugu, and J. Penjaam, editors, *Constraint Programming: Proceedings 1993 NATO ASI Parnu, Estonia*, NATO Advanced Science Institute Series, pages 361–410, Berlin, Germany / Heidelberg, Germany / London, UK / etc., 1994. Springer Verlag.

[20] J. R. Shoenfield. *Mathematical Logic.* Addison-Wesley Publishing Company, 1967.

[21] A. Sistla, M. Vardi, and P. Wolper. The complementation problem for buchi automata with applications to temporal logic. *Theoretical Computer Science*, 49:217–237, 1987.

[22] G. Smolka. A Foundation for Concurrent Constraint Programming. In *Constraints in Computational Logics*, volume 845 of *Lecture Notes in Computer Science*, Munich, Germany, September 1994. Invited Talk.

[23] S. Tini. On the expressiveness of timed concurrent constraint programming. *Electronics Notes in Theoretical Computer Science*, 1999.

$$\mathrm{R_T}\ \frac{}{\langle \mathbf{tell}(c), d\rangle \longrightarrow \langle \mathbf{skip}, d\dot{\wedge} c\rangle}$$

$$\mathrm{R_W}\ \frac{d \vdash c}{\langle \mathbf{when}\ c\ \mathbf{do}\ P, d\rangle \longrightarrow \langle P, d\rangle}$$

$$\mathrm{R_{PL}}\ \frac{\langle P, c\rangle \longrightarrow \langle P', d\rangle}{\langle P \parallel Q, c\rangle \longrightarrow \langle P' \parallel Q, d\rangle}$$

$$\mathrm{R_{PR}}\ \frac{\langle Q, c\rangle \longrightarrow \langle Q', d\rangle}{\langle P \parallel Q, c\rangle \longrightarrow \langle P \parallel Q', d\rangle}$$

$$\mathrm{R_U}\ \frac{d \vdash c}{\langle \mathbf{unless}\ c\ \mathbf{next}\ P, d\rangle \longrightarrow \langle \mathbf{skip}, d\rangle}$$

$$\mathrm{R_L}\ \frac{\langle P, c \wedge (\exists_x d)\rangle \longrightarrow \langle P', c'\rangle}{\langle (\mathbf{local}\, x, c)\, P, d\rangle \longrightarrow \langle (\mathbf{local}\, x, c')\, P', d \wedge \exists_x c'\rangle}$$

$$\mathrm{R_O}\ \frac{\langle P, c\rangle \longrightarrow^* \langle Q, d\rangle \not\longrightarrow}{P \xRightarrow{(c,d)} F(Q)}$$

Table 1: Rules for the internal reduction $\longrightarrow$ (upper part) and the observable reduction $\Longrightarrow$ (lower part). Function F is given in Definition 2.3.