1991

F.S. de Boer, C. Palamidessi

Embedding as a tool for language comparison

Computer Science/Department of Software Technology   Report CS-R9102   January

Embedding as a Tool for Language Comparison

Frank S. de Boer[1] and Catuscia Palamidessi[2]

[1]Technische Universiteit Eindhoven,
P.O. Box 513, 5600 MB Eindhoven, The Netherlands
email: wsinfdb@tuewsd.win.tue.nl

[2]Dipartimento di Informatica, Università di Pisa,
Corso Italia 40, 56125 Pisa, Italy
email: katuscia@dipisa.di.unipi.it

[2]Centre for Mathematics and Computer Science,
P.O. Box 4079, 1009 AB Amsterdam, The Netherlands
email: katuscia@cwi.nl

Abstract

This paper addresses the problem of defining a formal tool to compare the expressive power of different concurrent logic languages. We refine the basic notion of embedding [13] by adding some "reasonable" conditions, specific for concurrent frameworks. The new notion, called *modular embedding*, is used to define a preorder among these languages, representing the different degree of their expressivity. We show that this preorder is not trivial (i.e. it does not collapse into one equivalence class) by proving that Flat CP cannot be embedded into Flat GHC, and that Flat GHC cannot be embedded into a language without communication primitives in the guards, whilst the converses hold.

C.R. Categories: D.1.3, D.3.1, F.1.2, F.3.2, F.4.1.

Key Words and Phrases: concurrent logic languages, compositional semantics, embedding.

Note: Part of this work was carried out in the context of the ESPRIT Basic Research Action (3020) Integration. The research of Frank S. de Boer was partially supported by the Dutch REX (Research and Education in Concurrent Systems) project. The stay of Catuscia Palamidessi at the Centre for Mathematics and Computer Science was partially supported by the Italian CNR (Consiglio Nazionale delle Ricerche).

1 Introduction

1.1 Motivations

From a mathematical point of view, all programming languages are equivalent, since all of them can compute the same class of functions. Yet, it is common to compare languages on the basis of their "expressive power", as opposite to the "computing power".

Unfortunately, it is rather difficult to give a general formalization of the notion of expressivity, since the scope of its application is very broad: it can refer to data structures, to control capabilities, to communication and synchronization mechanisms etc. This formalization becomes intuitively more feasible when restricting to specific classes of similar languages, the differences being limited to some particular feature.

Report CS-R9102
Centre for Mathematics and Computer Science
P.O. Box 4079, 1009 AB Amsterdam, The Netherlands

This paper represents a first step towards the development of a formal method of comparison between concurrent logic languages. We consider a class of logic languages based on the same data structures (a parametric underlying constraint system) and on the same choice and parallel operators. The differences are relative to the way in which a process can be controlled by the environment. More precisely, we consider a guarded choice operator, and, by varying what a guard can consist of, we model different kinds of interaction (synchronization) of a process with its environment.

Actually, we think that this method of comparison, and the results we present, is independent from the particular programming style (logic, imperative...), it rather relates to the kind of synchronization primitives allowed in the guards of concurrent languages. Therefore we believe that it can be extended to other concurrent paradigms, like the class of CSP dialects.

1.2 The framework

We consider a class of Flat Concurrent Logic Languages based on a constraint system [10, 11, 12]. This class will be denoted by $\mathrm{CL}_{\mathcal{G}}$, where CL stands for Concurrent Logic and $\mathcal{G}$ is a parameter which denotes the set of communication primitives that can occur in the guards. We consider the following possibilities: $\mathcal{G} = \emptyset$, $\mathcal{G} = \mathcal{A}$, or $\mathcal{G} = \mathcal{A} \cup \mathcal{T}$, where

$$\mathcal{A} = \{ask(\vartheta) \mid \vartheta \text{ is a constraint}\}$$

ask being a primitive that checks if the *store* implies a certain constrain, and block otherwise, and

$$\mathcal{T} = \{tell(\vartheta) \mid \vartheta \text{ is a constraint}\}.$$

tell being a primitive that adds a constraint to the store, if consistent, and fail otherwise.

This class can be seen as a particular instance of the cc paradigm [11], and it includes a large number of concurrent logic languages. For instance, when the constraint system is the set of equations on the Herbrand universe, $\mathrm{CL}_{\mathcal{A}}$ corresponds to Eventual Herbrand [11] and to Flat GHC in its earlier version [14]; $\mathrm{CL}_{\mathcal{A}\cup\mathcal{T}}$ corresponds to Atomic Herbrand [11], to ccH [7] and to the language of [8], that is a refined version of Flat CP [13].

1.3 The method

A natural way to compare the expressive power of two languages is to see whether all programs written in one language can be "easily" and "equivalently" translated into the other one, where equivalent is intended in the sense of the same observable behaviour. This notion has recently become popular under the name of *embedding*. The basic definition of embedding, given by Shapiro [13], is the following. Consider two languages, L and L'. Let $Prog_L$ and $Prog_{L'}$ be the sets of programs in L and L', respectively. Assume given the *observation criteria* $\mathcal{O} : Prog_L \to Obs$ and $\mathcal{O}' : Prog_{L'} \to Obs'$, where Obs, Obs' are some suitable domains. Then L *embeds* L' if there exists a mapping $\mathcal{C}$ (*compiler*) from $Prog_{L'}$ to $Prog_L$, and a mapping $\mathcal{D}$ (*decoder*) from Obs to Obs' such that for every program W in L' we have

$$\mathcal{D}(\mathcal{O}[\![\mathcal{C}(W)]\!]) = \mathcal{O}'[\![W]\!]$$

In other words, the diagram of figure 1 commutes.

This notion however is too weak (as Shapiro himself remarked) since the above equation is satisfied by any language Turing-complete. In fact, if $\mathcal{O}$ is "powerful enough" and no restrictions are imposed on $\mathcal{C}$ and $\mathcal{D}$, we can just take a $\mathcal{C}$ such that $\mathcal{O} \circ \mathcal{C}$ does not identify more programs than $\mathcal{O}'$ and then define $\mathcal{D}$ as the function such that the diagram of figure 1 commutes.

In order to use the notion of embedding as a tool for comparison of (concurrent) languages we have therefore to add some restrictions. We do this by requiring $\mathcal{C}$ and $\mathcal{D}$ to satisfy certain properties that, to our opinion, are rather "reasonable" in a concurrent framework.

A first remark is the following. In a concurrent language, where indeterminism play an important role, the domain of the observables (Obs) is in general a powerset (i.e. the elements O of Obs

$$\begin{tikzcd}
Prog_{L'} \arrow[r, "\mathcal{O}'"] \arrow[d, "\mathcal{C}"'] & Obs' \\
Prog_{L} \arrow[r, "\mathcal{O}"'] & Obs \arrow[u, "\mathcal{D}"']
\end{tikzcd}$$

Figure 1: basic embedding.

are sets). In fact, each element must represent the various possible outcomes of a computation. Moreover, each outcome will be observed independently from the other possible ones. Therefore it is reasonable to require $\mathcal{D}$ to be defined *elementwise* on the sets that are contained in *Obs*. Formally:

P1 $\forall O \in Obs.\, \mathcal{D}(O) = \{\mathcal{D}_{el}(o) \mid o \in O\}$

for some appropriate $\mathcal{D}_{el}$.

Yet, this restriction doesn't increase significantly the discriminating power of the notion of embedding. In fact, we can always define $\mathcal{C}$ in such a way that $\mathcal{O}[\![\mathcal{C}(W)]\!]$ has a cardinality bigger than $\mathcal{O}'[\![W]\!]$ and each element encodes the program W. Then it is sufficient to define a function $\mathcal{D}_{el}$ parametric on W, such that (for $\mathcal{D}$ satisfying **P1**) the diagram of figure 1 commutes. We will develop formally this argument in section 4.

Another observation is the following. When compiling a concurrent process, it might be not possible to have all the informations about the processes that will be present in the environment at run time. Therefore it is reasonable to require the "separate compilation" of the parallel processes, or, in other words, the *compositionality* of the compiler with respect to the parallel operator.

Analogously, it is useful to compile a process in a compositional way with respect to the possible indeterministic choices, so to offer the possibility to add other alternatives after the compilation is made. These properties can be formulated as follows (from now on, we will refer to processes $A, B, \ldots$ instead of programs):

P2 $\mathcal{C}(A \parallel' B) = \mathcal{C}(A) \parallel \mathcal{C}(B)$ *and* $\mathcal{C}(A +' B) = \mathcal{C}(A) + \mathcal{C}(B)$

for every pair of processes A and B in L'. (Here $\parallel$, $\parallel'$, $+$, and $+'$ represent the parallel operators and the indeterministic choice operators in L and L' respectively.)

This condition can be generalized by requiring $\mathcal{C}$ to be compositional with respect to a generic set Op of operators of L', and the compositionality to be expressed in terms of contexts. Formally

$$\mathcal{C}(op(A_1, \ldots, A_n)) = c_{op}[\mathcal{C}(A_1), \ldots, \mathcal{C}(A_n)]$$

for every n-ary $op \in Op$, and for every $A_1, \ldots, A_n$ in L'. (Here $c_{op}[\,]$ represents an n-ary context in L.)

The generalization can be motivated by the fact that it is natural to compare the expressivity of two languages upon the capability of one language to express the operators of the other one. Furthermore, it allows to deal with the case of concurrent languages which, for instance, do not have an explicit parallel operator, like Pool [1], where a more powerful mechanism (process creation) is present instead. However, in this paper we will only require the compositionality with respect to the parallel and the choice operators.

A final point is that the embedding must preserve the behaviour of the original process with respect to deadlock (and/or failure) and success. Intuitively, a system non deadlock-free cannot be considered equivalent to a system deadlock-free. Therefore we require the termination mode of the target language not to be affected by the decoder (*termination invariance*). In other words, a deadlock [failure] in $\mathcal{O}[\![\mathcal{C}(A)]\!]$ must correspond to a deadlock [failure] in $\mathcal{O}'[\![A]\!]$, and a success must correspond to a success. Formally

$$\textbf{P3} \quad \forall O \in \mathit{Obs}. \forall o \in O.\ \mathit{tm}'(\mathcal{D}_{el}(o)) = \mathit{tm}(o)$$

where tm and tm' extract the information concerning the termination mode from the observables of L and L' respectively.

An embedding is called *modular* if it satisfies the three properties **P1**, **P2** and **P3** above. In the following we will omit the word modular when its presence is clear from the context.

1.4 Technical results

We show that the definition of modular embedding "makes sense" by proving that $\mathrm{CL}_{\emptyset}$ cannot embed $\mathrm{CL}_{\mathcal{A}}$ (Flat GHC) and that $\mathrm{CL}_{\mathcal{A}}$ cannot embed $\mathrm{CL}_{\mathcal{A}\cup\mathcal{T}}$ (Flat CP) (***separation results***). As far as we now, this was never formally proved before, also not for other restrictions on the notion of embedding.

1.5 The technique

In general, it is easy to prove directly that L embeds L' by showing how to translate all the operators of L' into L (for instance, it is easy to show that Flat CP can embed Flat GHC). A direct approach is however not feasible to prove that L *doesn't* embed L'. In fact, in principle we should check that the equation above does not hold for every possible compiler $\mathcal{C}$ and decoder $\mathcal{D}$ (satisfying the requirements **P1**,**P2** and **P3** above).

A solution is to work at the level of semantics, instead of the syntactical one. Our technique is quite general and can be explained in an abstract way as follows. We consider a *compositional* semantics $\mathcal{M} : \mathit{Prog}_L \rightarrow P$ (where P is some appropriate semantical domain encoding the termination mode), which is *correct* and *termination invariant* with respect to the observables, i.e. there exists a termination invariant abstraction $\mathcal{R} : P \rightarrow \mathit{Obs}$ such that the diagram of figure 2 commutes. Then we prove that the image of $\mathcal{M}$ ($\mathcal{M}(\mathit{Prog}_L)$) satisfies a certain property π, that cannot be satisfied by any semantical domain for L', when compositionality and termination invariance are required. Finally, we reason by contradiction: consider $\mathcal{M}' = \mathcal{M} \circ \mathcal{C}$, and $\mathcal{R}' = \mathcal{D} \circ \mathcal{R}$. We have that $\mathcal{M}'$ is compositional (since $\mathcal{M}$ and $\mathcal{C}$ are compositional) and that $\mathcal{R}'$ is termination invariant (since $\mathcal{R}$ and $\mathcal{D}$ are termination invariant). Therefore we obtain a compositional, correct and termination invariant semantics for L', whose image satisfies π. The situation is illustrated by the commutative diagram in figure 3, obtained by composing the diagrams of figure 1 and 2.

$$\begin{tikzcd}
 & P \arrow[dr, "\mathcal{R}"] & \\
\mathit{Prog}_L \arrow[ur, "\mathcal{M}"] \arrow[rr, "\mathcal{O}"'] & & \mathit{Obs}
\end{tikzcd}$$

Figure 2: a compositional and correct semantics $\mathcal{M}$.

More specifically, we use the compositional operational semantics developed in [3, 4, 5] (for a similar class of languages). This model is rather simple, therefore it is suitable for the investigation

$$\begin{tikzcd}
Prog_{L'} \arrow[rr, "\mathcal{O}'"] \arrow[dd, "\mathcal{C}"'] \arrow[dr, "\mathcal{M}'"] & & Obs' \\
& P \arrow[ur, "\mathcal{R}'"] \arrow[dr, "\mathcal{R}"] & \\
Prog_{L} \arrow[rr, "\mathcal{O}"'] \arrow[ur, "\mathcal{M}"] & & Obs \arrow[uu, "\mathcal{D}"']
\end{tikzcd}$$

Figure 3: the compositional and correct semantics $\mathcal{M}'$.

on the properties of the different languages of this class. We prove that the semantics of $\mathrm{CL}_{\mathcal{A}}$ [$\mathrm{CL}_{\emptyset}$] satisfies a certain closure property that in general is not satisfied by $\mathrm{CL}_{\mathcal{AUT}}$ [$\mathrm{CL}_{\mathcal{A}}$]. This closure property gives rise to an observable distinction that cannot be hidden by the decoder, when **P1**,**P2** and **P3** are required.

1.6 Related works

Bougé [2] has presented similar separation results for three CSP-dialects which closely correspond to the languages we study: CSP with (input and) output guards, CSP with input guards, and CSP with no communication primitives in the guards. The method he uses is based on showing that, given some communication graph, one dialect admits symmetric solutions to the *election problem*, whilst another dialect doesn't. These solutions are required to satisfy certain conditions about termination, and this is rather similar to what we do by requiring termination invariance. In order to generalize this approach so to obtain separation results (i.e. general statements concerning the non-embeddability), Bougé requires the compiler to translate parallel processes into parallel ones and, additionally, to preserve the topology of the network, namely the structure of the communication graph. This last restriction would be too strong for concurrent logic languages, and also difficult to formulate, since the communication network evolves dynamically in the logic paradigm.

Concerning the field of logic languages, Shapiro [13] has defined the notion of *natural embedding* (between logic programs), that suggests a method of comparison based on the complexity of the compiler. This notion consists of the basic definition plus the following restriction on the decoder $\mathcal{D}$:

$$\forall W \in Prog_{L'}.\forall p \in Pred_W.\ \mathcal{D}(\mathcal{O}[\![\mathcal{C}(W)]\!]_{|p}) = \mathcal{O}[\![\mathcal{C}(W)]\!]_{|p}$$

(where $Pred_W$ is the set of predicates in W) i.e., $\mathcal{D}$, restricted to observables of atoms corresponding to the original atoms of W, is the identity. This prevents the simulation of the variables of the source language with constants of the target language. In other terms the embedding must *absorb* the basic execution mechanism of logic languages, the unification. This restriction cannot easily be compared with the ones we give, actually we think that they are rather orthogonal.

1.7 Plan of the paper

This paper is organized as follows. Next section introduces the class $\mathrm{CL}_{\mathcal{G}}$ and its standard semantics, specified via a transition system. In section 3 this transition system is enriched so to derive a compositional semantics based on linear sequences. In section 4 we present the basic notion of embedding, and we show that it is not strong enough, by proving that $\mathrm{CL}_{\emptyset}$ embeds every other language of the class. In section 5 we introduce the new notion of modular embedding, and in section 6 we prove that it separates $\mathrm{CL}_{\mathcal{A}}$, $\mathrm{CL}_{\mathcal{A}}$ (Flat GHC), and $\mathrm{CL}_{\mathcal{A}\cup\mathcal{T}}$ (Flat CP). Finally, in section 7 we discuss the scope of this result.

2 The class $\mathrm{CL}_{\mathcal{G}}$

In this section we present the class of languages $\mathrm{CL}_{\mathcal{G}}$. Similar definitions can be found in [11, 12, 5]. The main difference with [12, 5] is that we do not deal here with an explicit hiding operator, because the results we present do not depend upon it. We refer to [11] for a detailed introduction to concurrent logic languages based on constraints.

2.1 The syntax

A constraint system is any system of partial information that supports the notions of *consistency* and *entailment*. For the sake of simplicity we consider here constraint systems based on first-order languages, however our results can be extended to more general settings. Let *Var* be a set of variables with typical elements $x, y, \ldots$, let *Fun* be a set of function symbols $a, b, \ldots, f, g, \ldots$, and let Pred_C be a set of predicate symbols. Terms on *Var* and *Fun* will be denoted by $t, u \ldots$. Let $\Sigma = (\mathit{Var}, \mathit{Fun}, \mathit{Pred}_C)$. A constraint system Γ is a first-order theory in Σ. A set *Con* of constraints, with typical elements $\vartheta, \sigma \ldots$, is a subset of the formulas of Γ. Given the constraints ϑ, ϑ_1, and ϑ_2, we say that ϑ is *consistent* if $\Gamma \models \exists\vartheta$, where $\exists\vartheta$ denotes the existential closure of ϑ, and that ϑ_1 *entails* ϑ_2 if $\Gamma \models \vartheta_1 \Rightarrow \vartheta_2$. Consistency and entailment are usually assumed to be decidable. We consider a fixed Γ, so we will omit references to it.

We now describe the class $\mathrm{CL}_{\mathcal{G}}$ based on Γ. The parameter $\mathcal{G}$ specifies the *communication primitives* used in the guards. We restrict here to the following cases: $\mathcal{G} = \emptyset$, $\mathcal{G} = \mathcal{A}$, or $\mathcal{G} = \mathcal{A} \cup \mathcal{T}$, where

$$\mathcal{A} = \{ask(\vartheta) \mid \vartheta \in \mathit{Con}\}$$

and

$$\mathcal{T} = \{tell(\vartheta) \mid \vartheta \in \mathit{Con}\}.$$

Let Pred_L be a set of predicate symbols (procedure names), with typical elements $p, q, r, \ldots$, disjoint from Pred_C. The set of processes $\mathit{Proc}_{\mathcal{G}}$ in $\mathrm{CL}_{\mathcal{G}}$, with typical elements $A, B, \ldots$, is described by the following grammar

$$\boxed{\begin{array}{l} A ::= ask(\vartheta) \mid tell(\vartheta) \mid A \parallel A \mid G \mid p(\vec{t}) \mid W; A \\[1ex] G ::= g \rightarrow A \mid G + G \\[1ex] W ::= p(\vec{x}) \text{ :- } A \mid W, W \end{array}}$$

The effect of the primitives $ask(\vartheta)$ and $tell(\vartheta)$ is defined with respect to a given *store* σ, that represents the constraint accumulated during the computation. The primitive $ask(\vartheta)$ checks wether ϑ is entailed by σ, and blocks otherwise; $tell(\vartheta)$ checks if ϑ is consistent with σ, and in that case it adds ϑ, otherwise it fails. The symbol $\parallel$ represents the parallel operator. A *guarded process* $g \rightarrow A$ first executes g (if possible) and then it behaves like A. The *guard* g belongs to $\mathcal{G}$. For the sake of uniformity, we assume $g = tell(true)$ in the case $\mathcal{G} = \emptyset$. The indeterministic choice is represented by $+$ and it is guarded, namely, $+$ applies to guarded processes and selects those ones whose guard is enabled. The process $p(\vec{t})$ is a *procedure call*, where the sequence of terms $\vec{t}$ represents the list of actual parameters. Finally, $W; A$ represents the process A in the scope of the set W of *procedure declarations*, namely, objects of the form $p(\vec{x})$:- A where $\vec{x}$ represents the list of formal parameters. W is usually called *program*, and its elements *clauses*.

Given the list of actual parameters $\vec{t}$, an *instantiation* of $p(\vec{x})$:- A (w.r.t. $\vec{t}$) is an object of the form $p(\vec{t})$:- A', where A' is obtained from A by simultaneously replacing every (occurrence of a) formal parameter by its corresponding actual parameter, and by renaming all the other variables so to avoid clashes with $\vec{t}$. Given a program W we denote by $Inst(W)$ the set of all the instantiations of its clauses.

2.2 The operational model

The operational model of $\mathrm{CL}_{\mathcal{G}}$ is uniformly described by a transition system $T = (Conf, \longrightarrow_W)$. The configurations $Conf$ are pairs consisting of a process or a *termination mode*, and a store. The termination modes α are the symbols ss, $f\!f$ and dd, that denote success, failure and deadlock respectively. A transition $\langle A, \sigma\rangle \longrightarrow_W \langle A', \sigma'\rangle$ must be read as follows: $\langle A, \sigma\rangle$ can make a transition step, (resulting in $\langle A', \sigma'\rangle$), assuming that A is within the scope of the program W. We regard a program as a set, i.e. the order in which clauses occur is not relevant, and the basic operation is the set union. The rules of T are described in table 1.

We assume the presence of a renaming mechanism that takes care of using fresh variables each time a clause is considered (in **R5**). For the sake of simplicity we do not describe this renaming mechanism in T. The interested reader can find in [12, 3, 5] various formal approaches to this problem.

The first four rules describe the way in which communication and synchronization is achieved in this language. Rule **R5** describes the replacement of a procedure call (in the scope of W) by the body of the procedure definition (in W). A procedure call fails (**R6**) if undefined. Rule **R7** verifies that the transitions made under the assumption of a program W are indeed within the scope of W. Finally, **R8-R12** are the usual rules for compound statements (note that parallelism is described as interleaving).

The result of a terminating computation consists of the final store (up to logical equivalence), together with the termination mode. This is formally represented by the notion of *observables*.

Definition 2.1 *The observables of the class* $\mathrm{CL}_{\mathcal{G}}$ *are given by the function* $\mathcal{O} : Proc_{\mathcal{G}} \rightarrow Obs$, *where* $Obs = \mathcal{P}(Con \times \{ss, f\!f, dd\})$, *defined as*

$$\mathcal{O}[\![A]\!] = \{\langle\sigma, \alpha\rangle \mid \langle A, true\rangle \longrightarrow_{\emptyset}^{*} \langle\alpha, \sigma\rangle\}_{\Leftrightarrow},$$

where the subscript $\Leftrightarrow$ *denotes the closure under logical equivalence, and* $\longrightarrow_{\emptyset}^{*}$ *denotes the transitive closure of* $\longrightarrow_{\emptyset}$.

In the above definition the subscript $\emptyset$ in $\longrightarrow_{\emptyset}$ represents the absence of any external program.

3 A compositional semantics for $\mathrm{CL}_{\mathcal{G}}$

In this section we enrich the model of the previous section so to obtain a semantics compositional with respect to the parallel and the choice operators. The model we present here is essentially a simplified version of the ones developed in [3, 4, 5] for similar languages, and actually the semantics we obtain is compositional also with respect to other operators, but this is not of our interest in this paper. For a more detailed presentation of this section we refer to [5].

The behaviour of a process is described as a sequence of *interactions* with its environment (the other parallel processes). Interactions are modeled as *assume/tell* constraints. An assume constraint is an assumption about the constraint provided by the environment, whereas a tell constraint is produced by the goal itself.

Definition 3.1

- *The set of assume constraints is* $Con_A = \{\vartheta^A \mid \vartheta \in Con\}$.
- *The set of tell constraint is* $Con_T = \{\vartheta^T \mid \vartheta \in Con\}$.

Table 1: The Transition System T

R1	$\langle ask(\vartheta),\sigma\rangle \longrightarrow_W \langle ss,\sigma\rangle$	**if** $\models \sigma \Rightarrow \vartheta$
R2	$\langle ask(\vartheta),\sigma\rangle \longrightarrow_W \langle dd,\sigma\rangle$	**if** $\not\models \sigma \Rightarrow \vartheta$
R3	$\langle tell(\vartheta),\sigma\rangle \longrightarrow_W \langle ss,\sigma \wedge \vartheta\rangle$	**if** $\models \exists(\sigma \wedge \vartheta)$
R4	$\langle tell(\vartheta),\sigma\rangle \longrightarrow_W \langle \mathit{ff},\sigma\rangle$	**if** $\not\models \exists(\sigma \wedge \vartheta)$
R5	$\langle p(\vec{t}),\sigma\rangle \longrightarrow_W \langle A,\sigma\rangle$	**if** $p(\vec{t})$:- $A \in \mathit{Inst}(W)$
R6	$\langle p(\vec{t}),\sigma\rangle \longrightarrow_W \langle \mathit{ff},\sigma\rangle$	**if** $p(\vec{t})$:- $\ldots \notin \mathit{Inst}(W)$
R7	$\dfrac{\langle A,\sigma\rangle \longrightarrow_{W\cup W'} \langle A',\sigma'\rangle \mid \langle \alpha,\sigma'\rangle}{\langle W;A,\sigma\rangle \longrightarrow_{W'} \langle W;A',\sigma'\rangle \mid \langle \alpha,\sigma'\rangle}$	
R8	$\dfrac{\langle g,\sigma\rangle \longrightarrow_W \langle ss,\sigma'\rangle \mid \langle \alpha,\sigma\rangle}{\langle g \rightarrow A,\sigma\rangle \longrightarrow_W \langle A,\sigma'\rangle \mid \langle \alpha,\sigma\rangle}$	**if** $\alpha \in \{\mathit{ff}, dd\}$
R9	$\dfrac{\langle A,\sigma\rangle \longrightarrow_W \langle A',\sigma'\rangle \mid \langle ss,\sigma'\rangle}{\begin{array}{l} \langle A \parallel B,\sigma\rangle \longrightarrow_W \langle A' \parallel B,\sigma'\rangle \mid \langle B,\sigma'\rangle \\ \langle B \parallel A,\sigma\rangle \longrightarrow_W \langle B \parallel A',\sigma'\rangle \mid \langle B,\sigma'\rangle \\ \langle A+B,\sigma\rangle \longrightarrow_W \langle A',\sigma'\rangle \mid \langle ss,\sigma'\rangle \\ \langle B+A,\sigma\rangle \longrightarrow_W \langle A',\sigma'\rangle \mid \langle ss,\sigma'\rangle \end{array}}$	
R10	$\dfrac{\langle A,\sigma\rangle \longrightarrow_W \langle dd,\sigma\rangle \quad \langle B,\sigma\rangle \longrightarrow_W \langle \alpha,\sigma\rangle}{\begin{array}{l} \langle A \parallel B,\sigma\rangle \longrightarrow_W \langle \alpha,\sigma\rangle \\ \langle B \parallel A,\sigma\rangle \longrightarrow_W \langle \alpha,\sigma\rangle \\ \langle A+B,\sigma\rangle \longrightarrow_W \langle dd,\sigma\rangle \\ \langle B+A,\sigma\rangle \longrightarrow_W \langle dd,\sigma\rangle \end{array}}$	**if** $\alpha \in \{\mathit{ff}, dd\}$
R11	$\dfrac{\langle A,\sigma\rangle \longrightarrow_W \langle \mathit{ff},\sigma\rangle}{\begin{array}{l} \langle A \parallel B,\sigma\rangle \longrightarrow_W \langle \mathit{ff},\sigma\rangle \\ \langle B \parallel A,\sigma\rangle \longrightarrow_W \langle \mathit{ff},\sigma\rangle \end{array}}$	
R12	$\dfrac{\langle A,\sigma\rangle \longrightarrow_W \langle \mathit{ff},\sigma\rangle \quad \langle B,\sigma\rangle \longrightarrow_W \langle \mathit{ff},\sigma\rangle}{\begin{array}{l} \langle A+B,\sigma\rangle \longrightarrow_W \langle \mathit{ff},\sigma\rangle \\ \langle B+A,\sigma\rangle \longrightarrow_W \langle \mathit{ff},\sigma\rangle \end{array}}$	

- *The set of assume/tell constraints, with typical element ϑ^ℓ, is $Con_{AT} = Con_A \cup Con_T$.*

The compositional semantics is based on a transition system $T^C = (Conf^C, \longrightarrow_W^C)$. A first difference from T is that the store is represented by a finite sequence c of assume/tell constraints ($c \in Con^*_{AT}$). The *store* defined by such a sequence is the conjunction of all constraints, ignoring the assume/tell mode.

Definition 3.2 *The function store : $Con^*_{AT} \rightarrow Con$ is defined as follows:*

- $store(\lambda) = true$
- $store(\vartheta^\ell.c) = \vartheta \wedge store(c)$

where λ denotes the empty sequence.

The notations for the entailment and consistency extend naturally to sequences: $\models c \Rightarrow \vartheta$ stands for $\models store(c) \Rightarrow \vartheta$, and $\models \exists(c \wedge \vartheta)$ stands for $\models \exists(store(c) \wedge \vartheta)$.

Table 2 describes the rules for T^C. For the sake of convenience, we drop the superscript C in the transition relation.

The last rule **C13** models (an assumption on) a transition made by the environment. All the other rules essentially mimic the ones of the transition system T.

We define now a compositional semantics $\mathcal{M}$ based on the transition system T^C. The meaning of a process will be defined as the sets of sequences of assume/tell constraints, ended by the termination mode, corresponding to all the possible computations. We are interested only in describing finite computations. However, in order to describe failure compositionally, we must assign a non-empty semantics also to non-terminating programs. This is done by adding an "artificial" termination mode, $\perp$, that represents an "unfinished" computation.

We denote the set of sequences as $Seq = Con^*_{AT}.\{ss, ff, dd, \perp\}$, the typical element will be represented by s.

In the sequel, $\mathcal{P}$ will denote the powerset operation. The semantical domain of $\mathcal{M}$, $\mathcal{P}(Seq)$, will be denoted by P.

Definition 3.3 *The semantics $\mathcal{M} : Procg \rightarrow P$ is defined as*

$$\mathcal{M}[\![A]\!] = \{c.\alpha \mid \langle A, \lambda\rangle \longrightarrow_\emptyset^* \langle \alpha, c\rangle\} \cup \{c.\perp\mid \langle A, \lambda\rangle \longrightarrow_\emptyset^* \langle A', c\rangle\}$$

Next we show that $\mathcal{M}$ is correct and compositional.

3.1 Correctness of $\mathcal{M}$

A semantics is correct if there exists a mapping that allows to obtain the observables of a process starting from its denotation. This mapping is usually not injective, i.e. its application forgets some of the informations encoded in the semantical domain. For this reason it is called *abstraction operator*.

To show the correctness of $\mathcal{M}$, let's first consider which are the informations (encoded by $\mathcal{M}$) that are not observable. By definition, given a process, $\mathcal{O}$ produces the results of the computations that are

- terminating, and
- carried out by the process itself, i.e. without the help of any environment.

The sequences in $\mathcal{M}$ that terminate with the symbol $\perp$ and/or contain assumptions (constraints labeled by assume mode) do not correspond to any observable computation, and therefore must be eliminated. The results can be extracted from the remaining sequences by considering the final store associated with them, and then closing under logical equivalence.

This notion of abstraction is formalized by the following operator $\mathcal{R}$.

Table 2: The Transition System T^C

C1	$\langle ask(\vartheta), c\rangle \longrightarrow_W \langle ss, c.true^T\rangle$	**if** $\models c \Rightarrow \vartheta$
C2	$\langle ask(\vartheta), c\rangle \longrightarrow_W \langle dd, c\rangle$	**if** $\not\models c \Rightarrow \vartheta$
C3	$\langle tell(\vartheta), c\rangle \longrightarrow_W \langle ss, c.\vartheta^T\rangle$	**if** $\models \exists(c \wedge \vartheta)$
C4	$\langle tell(\vartheta), c\rangle \longrightarrow_W \langle ff, c\rangle$	**if** $\not\models \exists(c \wedge \vartheta)$
C5	$\langle p(\vec{t}), c\rangle \longrightarrow_W \langle A, c.true^T\rangle$	**if** $p(\vec{t})$:- $A \in Inst(W)$
C6	$\langle p(\vec{t}), c\rangle \longrightarrow_W \langle ff, c\rangle$	**if** $p(\vec{t})$:- $\ldots \notin Inst(W)$
C7	$\dfrac{\langle A, c\rangle \longrightarrow_{W \cup W'} \langle A', c'\rangle \mid \langle \alpha, c'\rangle}{\langle W; A, c\rangle \longrightarrow_{W'} \langle W; A', c'\rangle \mid \langle \alpha, c'\rangle}$	
C8	$\dfrac{\langle g, c\rangle \longrightarrow_W \langle ss, c'\rangle \mid \langle \alpha, c\rangle}{\langle g \rightarrow A, c\rangle \longrightarrow_W \langle A, c'\rangle \mid \langle \alpha, c\rangle}$	**if** $\alpha \in \{ff, dd\}$
C9	$\dfrac{\langle A, c\rangle \longrightarrow_W \langle A', c.\vartheta^T\rangle \mid \langle ss, c.\vartheta^T\rangle}{\begin{array}{l}\langle A \parallel B, c\rangle \longrightarrow_W \langle A' \parallel B, c.\vartheta^T\rangle \mid \langle B, c.\vartheta^T\rangle \\ \langle B \parallel A, c\rangle \longrightarrow_W \langle B \parallel A', c.\vartheta^T\rangle \mid \langle B, c.\vartheta^T\rangle \\ \langle A+B, c\rangle \longrightarrow_W \langle A', c.\vartheta^T\rangle \mid \langle ss, c.\vartheta^T\rangle \\ \langle B+A, c\rangle \longrightarrow_W \langle A', c.\vartheta^T\rangle \mid \langle ss, c.\vartheta^T\rangle\end{array}}$	
C10	$\dfrac{\langle A, c\rangle \longrightarrow_W \langle dd, c\rangle \quad \langle B, c\rangle \longrightarrow_W \langle \alpha, c\rangle}{\begin{array}{l}\langle A \parallel B, c\rangle \longrightarrow_W \langle \alpha, c\rangle \\ \langle B \parallel A, c\rangle \longrightarrow_W \langle \alpha, c\rangle \\ \langle A+B, c\rangle \longrightarrow_W \langle dd, c\rangle \\ \langle B+A, c\rangle \longrightarrow_W \langle dd, c\rangle\end{array}}$	**if** $\alpha \in \{ff, dd\}$
C11	$\dfrac{\langle A, c\rangle \longrightarrow_W \langle ff, c\rangle}{\begin{array}{l}\langle A \parallel B, c\rangle \longrightarrow_W \langle ff, c\rangle \\ \langle B \parallel A, c\rangle \longrightarrow_W \langle ff, c\rangle\end{array}}$	
C12	$\dfrac{\langle A, c\rangle \longrightarrow_W \langle ff, c\rangle \quad \langle B, c\rangle \longrightarrow_W \langle ff, c\rangle}{\begin{array}{l}\langle A+B, c\rangle \longrightarrow_W \langle ff, c\rangle \\ \langle B+A, c\rangle \longrightarrow_W \langle ff, c\rangle\end{array}}$	
C13	$\langle A, c\rangle \mid \langle ss, c\rangle \longrightarrow_W \langle A, c.\vartheta^A\rangle \mid \langle ss, c.\vartheta^A\rangle$	**if** $\models \exists(c \wedge \vartheta)$

Definition 3.4

- *The partial operator Result* : $Seq \rightarrow Con \times \{ss, ff, dd\}$ *is defined as*

$$Result(c.\alpha) = \langle store(c), \alpha \rangle \quad \textit{if } c \textit{ contains only tell constraints, and } \alpha \neq \perp$$

- *The operator* $\mathcal{R} : \mathcal{P} \rightarrow Obs$ *is given by*

$$\mathcal{R}(S) = \bigcup_{c \in S} \{Result(c)\}_{\Leftrightarrow}$$

Note that $\mathcal{R}$ preserves the termination mode. Next lemma is useful to prove the correctness of $\mathcal{M}$.

Lemma 3.5 *The rules* **C1-C12** *of* T^C *mimic the rules* **R1-R12** *of* T*, in the sense that if*

$$\langle A, c \rangle \longrightarrow_W \langle A', c' \rangle$$

is a **Ci** *transition step in* T^C*, then*

$$\langle A, store(c) \rangle \longrightarrow_W \langle A', store(c') \rangle$$

is a **Ri** *transition step in* T.

Proof Immediate by case analysis of the rules in T^C. □

Theorem 3.6 (Correctness of $\mathcal{M}$) *The observables* $\mathcal{O}$ *can be obtained by* $\mathcal{R}$*-abstraction from* $\mathcal{M}$*, namely* $\mathcal{O} = \mathcal{R} \circ \mathcal{M}$.

Proof Let A be a process and let $c.\alpha \in \mathcal{M}[\![A]\!]$ such that c contains only tell constraints and $\alpha \neq \perp$. Then there is a derivation in T^C of the form

$$\langle A, \lambda \rangle \; = \; \langle A_0, c_0 \rangle \longrightarrow_{\emptyset} \ldots \langle A_i, c_i \rangle \longrightarrow_{\emptyset} \ldots \langle A_n, c_n \rangle \; = \; \langle \alpha, c \rangle$$

such that $\alpha \in \{ss, ff, dd\}$ and the rule **C13** is never used.

By lemma 3.5, there exists in T a derivation

$$\langle A, true \rangle \; = \; \langle A_0, store(c_0) \rangle \longrightarrow_{\emptyset} \ldots \langle A_i, store(c_i) \rangle \longrightarrow_{\emptyset} \ldots \langle A_n, store(c_n) \rangle \; = \; \langle \alpha, store(c) \rangle.$$

Therefore $\langle store(c), \alpha \rangle \in \mathcal{O}[\![A]\!]$. □

3.2 Compositionality of $\mathcal{M}$

To show the compositionality of $\mathcal{M}$ we define the semantic operators $\tilde{\|}$ and $\tilde{+}$, corresponding to the parallel and to the choice operators of the language.

The operator $\tilde{\|}$, first introduced in [9], allows to combine sequences of assume/tell constraints that are equal at each point, apart from the modes, so modeling the interaction of a process with its environment. It is similar to the (more popular) interleaving operator, the difference is that it applies to sequences containing already all the informations concerning the way in which processes interleave (the assumptions specify "where" and "what"). Hence the application of $\tilde{\|}$ amounts to verify that the assumptions made by one process are indeed validated by the other process (i.e. it tells or assumes the same constraints). In the positive case, the elements of the resulting sequence are labeled by tell whenever they are labeled by tell in at least one of the two sequences, by assume otherwise (a constraint is produced by a pair of parallel processes whenever it is produced by at least one of the two).

Concerning the definition of $\tilde{+}$, we observe that the execution of a guard is made visible by telling a constraint. Therefore, $\tilde{+}$ must select a sequence starting with a tell constraint (if any). In case both sequences start with an assumption, then it must be the same constraint (so modeling an assumption made by the whole process), and, in the positive case, the choice is postponed. The application of $\tilde{+}$ delivers different sequences, since there is the possibility that both sequences start with a tell, corresponding to the possibility that both guards are enabled.

Definition 3.7

- *The* partial *operators on sequences* $\tilde{\|} : Seq \times Seq \longrightarrow Seq$ *and* $\tilde{+} : Seq \times Seq \longrightarrow P$ *are defined as*

  - $s_1 \tilde{\|} s_2 = s_2 \tilde{\|} s_1$
  - $(\vartheta^A.s_1)\tilde{\|}(\vartheta^\ell.s_2) = \vartheta^\ell.(s_1 \tilde{\|} s_2)$
  - $ss\tilde{\|}\alpha = \alpha$
  - $\mathit{ff}\tilde{\|}\alpha = \mathit{ff}$
  - $dd\tilde{\|}dd = dd$
  - $\perp \tilde{\|} \perp = \perp$
  - $s_1 \tilde{+} s_2 = s_2 \tilde{+} s_1$
  - $(\vartheta^T.s_1)\tilde{+}s_2 = \vartheta^T.s_1$
  - $(\vartheta^A.s_1)\tilde{+}(\vartheta^A.s_2) = \vartheta^A.(s_1\tilde{+}s_2)$
  - $ss\tilde{+}\alpha = ss$
  - $\mathit{ff}\tilde{+}\alpha = \alpha$
  - $dd\tilde{+}dd = dd$
  - $\perp \tilde{+} \perp = \perp$

- *The operators on sets of sequences* $\tilde{\|}, \tilde{+} : P \times P \longrightarrow P$ *are defined by:*

$$S_1\tilde{\|}S_2 = \{s_1\tilde{\|}s_2 \mid s_1 \in S_1 \ \wedge \ s_2 \in S_2\}$$

$$S_1\tilde{+}S_2 = \bigcup\{s_1\tilde{+}s_2 \mid s_1 \in S_1 \ \wedge \ s_2 \in S_2\}$$

Theorem 3.8 (Compositionality of $\mathcal{M}$)

$\|)$ $\quad \mathcal{M}[\![A_1 \parallel A_2]\!] = \mathcal{M}[\![A_1]\!]\tilde{\|}\mathcal{M}[\![A_2]\!]$

$+)$ $\quad \mathcal{M}[\![A_1 + A_2]\!] = \mathcal{M}[\![A_1]\!]\tilde{+}\mathcal{M}[\![A_2]\!]$

Proof Consider the transition system T'^C obtained from T^C by transforming the rule **C9** into the following one

$$\mathbf{C9'} \quad \frac{\langle A,c\rangle \longrightarrow_W \langle A',c.\vartheta^T\rangle \mid \langle ss,c.\vartheta^T\rangle \quad \langle B,c\rangle \longrightarrow_W \langle B,c.\vartheta^A\rangle}{\begin{array}{l}\langle A \parallel B,c\rangle \longrightarrow_W \langle A' \parallel B,c.\vartheta^T\rangle \mid \langle B,c.\vartheta^T\rangle\\ \langle B \parallel A,c\rangle \longrightarrow_W \langle B \parallel A',c.\vartheta^T\rangle \mid \langle B,c.\vartheta^T\rangle\\ \langle A+B,c\rangle \longrightarrow_W \langle A',c.\vartheta^T\rangle \mid \langle ss,c.\vartheta^T\rangle\\ \langle B+A,c\rangle \longrightarrow_W \langle A',c.\vartheta^T\rangle \mid \langle ss,c.\vartheta^T\rangle\end{array}}$$

We have that T'^C is equivalent to T^C, namely the transitions generated by T'^C coincide with the ones generated by T^C. In fact, the premise $\langle B,c\rangle \longrightarrow_W \langle B,c.\vartheta^A\rangle$ in **C9'** is always validated by an application of **C13**, since the other premise $\langle A,c\rangle \longrightarrow_W \langle A',c.\vartheta^T\rangle \mid \langle ss,c.\vartheta^T\rangle$ ensures that $\models \exists(c \wedge \vartheta)$. The rest follows by easy induction on the length of the sequences. □

In the following examples, we assume the constraint system to support the usual equality theory on the Herbrand universe.

Example 3.9 *Consider the program*

$$W_1 \ = \ \{p_1(x,y) :\text{-}\ \mathit{ask}(x=a) \rightarrow \mathit{tell}(y=b)\}$$

and consider the process $A_1 = W_1;p_1(x,y)$. *We have*

$$\begin{array}{rcl}\mathcal{M}[\![A_1]\!] &=& \{\ (x=a)^A.\mathit{true}^T.(y=b)^T.\mathit{true}^A.ss\ ,\ \ (x=b)^A.\mathit{ff}\ ,\ \ dd\ ,\ \ \ldots\ \}\\ \mathcal{O}[\![A_1]\!] &=& \{\langle \mathit{true}, dd\rangle\}.\end{array}$$

Consider now the program

$$W_2 = \{p_2(x,y) :\text{-}\ \mathit{tell}(x=a) \rightarrow \mathit{ask}(y=b)\}$$

and consider the process $A_2 = W_2; p_2(x,y)$. We have

$$\begin{aligned}\mathcal{M}[\![A_2]\!] &= \{\ (x=a)^T.true^A.(y=b)^A.true^T.ss\ ,\quad (x=b)^A.\mathit{ff}\ ,\quad (x=a)^T.dd\ ,\ \ \dots\ \}\\ \mathcal{O}[\![A_2]\!] &= \{\langle x=a, dd\rangle\}\end{aligned}$$

Let now consider the compound processes $A_1 \parallel A_2$ and $A_1 + A_2$. We have

$$\begin{aligned}\mathcal{M}[\![A_1 \parallel A_2]\!] &= \mathcal{M}[\![A_1]\!]\tilde{\parallel}\mathcal{M}[\![A_2]\!]\\ &= \{\ (x=a)^T.true^T.(y=b)^T.true^T.ss\ ,\quad (x=b)^A.\mathit{ff}\ ,\ \ \dots\ \}\\ \mathcal{O}[\![A_1 \parallel A_2]\!] &= \{\langle (x=a \wedge y=b), ss\rangle\}.\\ \mathcal{M}[\![A_1+A_2]\!] &= \mathcal{M}[\![A_1]\!]\tilde{+}\mathcal{M}[\![A_2]\!]\\ &= \{\ (x=a)^T.true^A.(y=b)^A.true^T.ss\ ,\quad (x=a)^T.dd\ ,\ \ \dots\ \}\\ \mathcal{O}[\![A_1+A_2]\!] &= \{\langle x=a, dd\rangle\}.\end{aligned}$$

4 Basic embedding

In this section we discuss the notion of *embedding* proposed by Shapiro in [13]. We rephrase his notion in order to deal with processes rather than with programs.

Definition 4.1 (Embedding (Shapiro, [13])) *Let L, L' be two (concurrent) languages, let $Proc_L$, $Proc_{L'}$ be the respective sets of processes, and let $\mathcal{O} : Proc_L \to Obs$, $\mathcal{O}' : Proc_{L'} \to Obs'$ be their observation criteria. The language L embeds L' iff there exists a compiler $\mathcal{C} : Proc_{L'} \to Proc_L$, and a decoder $\mathcal{D} : Obs \to Obs'$ such that, for every $A \in Proc_{L'}$, the following equation holds*

$$\mathcal{D}(\mathcal{O}[\![\mathcal{C}(A)]\!]) = \mathcal{O}'[\![A]\!] \tag{1}$$

or, in other words, the diagram of figure 1 in section 1.3 (where Prog is replaced by Proc) commutes.

We will denote by $L' \leq L$ the existence of an embedding from L' into L. It is immediate to see that $\leq$ is a preorder relation, in fact the reflexivity is given by the possibility of defining $\mathcal{C}$ and $\mathcal{D}$ as the identity, and the transitivity is guarantied by the commutativity of the diagram in figure 4 (obtained by doubling and composing the commutative diagram of figure 1).

Remark 4.2 *If $L' \subseteq L$ then $L' \leq L$. Therefore, $\mathrm{CL}_\emptyset \leq \mathrm{CL}_{\mathcal{A}} \leq \mathrm{CL}_{\mathcal{AUT}}$. Note that the notion of observables (cfr. def. 2.1) is defined in the same way for all the languages of this class.*

Also the reverse relation holds, i.e. all languages of the class $\mathrm{CL}_{\mathcal{G}}$ are equivalent. In fact, as observed by Shapiro himself, definition 4.1 is "too weak": the above equation is satisfied by any pair of Turing-complete languages.

Our goal is to refine the notion of embedding so to capture the *distinctions* between concurrent logic languages. This will be done by requiring $\mathcal{C}$ and $\mathcal{D}$ to satisfy certain properties, specific for concurrency.

As discussed in section 1.3, it is reasonable to force $\mathcal{D}$ to be defined *elementwise*. Therefore the first requirement is the existence of a partial mapping $\mathcal{D}_{el} : Obs \to Obs$ such that

P1 $\forall O \in Obs.\ \mathcal{D}(O) = \{\mathcal{D}_{el}(\langle\sigma,\alpha\rangle) \mid \langle\sigma,\alpha\rangle \in O\}$

However, this restriction is not strong enough:

Theorem 4.3 *If the underlying constraint system contains at least one non-constant symbol, then all languages of the class $\mathrm{CL}_{\mathcal{G}}$ can be embedded into $\mathrm{CL}_\emptyset$ (with a decoder satisfying **P1**).*

$$\begin{tikzcd}
Proc_{L''} \arrow[r, "\mathcal{O}''"] \arrow[d, "\mathcal{C}'"'] & Obs'' \\
Proc_{L'} \arrow[r, "\mathcal{O}'"] \arrow[d, "\mathcal{C}"'] & Obs' \arrow[u, "\mathcal{D}'"'] \\
Proc_{L} \arrow[r, "\mathcal{O}"'] & Obs \arrow[u, "\mathcal{D}"']
\end{tikzcd}$$

Figure 4: transitivity of $\leq$.

Proof We assume (without loss of generality) that the constraint system supports the usual equality theory on the Herbrand universe. Let $\lceil\ \rceil : \mathit{Proc}_{\mathcal{G}} \longrightarrow \mathit{Term}_{\Gamma}$ be some injective representation (Gödelization) of the processes in $\mathrm{CL}_{\mathcal{G}}$ as terms in Γ.

Define the compiler $\mathcal{C} : \mathit{Proc}_{\mathcal{G}} \longrightarrow \mathrm{CL}_{\emptyset}$ as follows.

$$\mathcal{C}(A) = W_A; p_A(x,y)$$

where

$$\begin{array}{lll} W_A = \{ & p_A(x,y) :- & \mathit{tell}(\mathit{true}) \rightarrow \mathit{tell}(y = \lceil A \rceil) \\ & & + \\ & & \mathit{tell}(\mathit{true}) \rightarrow (\mathit{tell}(x = f(x')) \parallel p_A(x',y)) \ \}. \end{array}$$

It is easy to see that

$$\mathcal{O}[\![W_A; p_A(x,y)]\!] \supseteq \{\langle x = f^i(z) \wedge y = \lceil A \rceil, ss\rangle \mid i \geq 0\}$$

where $f^i(z)$ is the term obtained by applying i times the constructor f to z.

Consider an enumeration $\langle \sigma_i, \alpha_i\rangle$ of the elements of $\mathcal{O}[\![A]\!]$ and define $\mathcal{D}_{el}$ as the partial mapping

$$\mathcal{D}_{el}(\langle x = f^i(z) \wedge y = \lceil A \rceil, ss\rangle) = \langle \sigma_i, \alpha_i\rangle.$$

We obtain

$$\begin{array}{rcl} \mathcal{D}(\mathcal{O}[\![\mathcal{C}(A)]\!]) & = & \mathcal{D}(\mathcal{O}[\![W_A; p_A(x,y)]\!]) \\ & = & \{\mathcal{D}_{el}(\langle\sigma,\alpha\rangle) \mid \langle\sigma,\alpha\rangle \in \mathcal{O}[\![W_A; p_A(x,y)]\!]\} \\ & = & \{\mathcal{D}_{el}(\langle x = f^i(z) \wedge y = \lceil A \rceil, ss\rangle) \mid i \geq 0\} \\ & = & \{\langle\sigma_i, \alpha_i\rangle \mid i \geq 0\} \\ & = & \mathcal{O}[\![A]\!] \end{array}$$

□

5 Modular embedding

In this section we further refine the notion of embedding. Beside **P1**, we require the compiler and the decoder to satisfy the properties discussed in section 1.3, namely the *compositionality* with respect to $\parallel$ and $+$, and the preservation of the termination mode (*termination invariance*).

For the languages of the class $\mathrm{CL}_{\mathcal{G}}$ these properties can be formally described as follows:

P2 $\forall A, B \in \mathit{Proc}_{\mathcal{G}}.\ \mathcal{C}(A \parallel B) = \mathcal{C}(A) \parallel \mathcal{C}(B)$ and $\mathcal{C}(A + B) = \mathcal{C}(A) + \mathcal{C}(B)$

P3 $\forall\sigma.\forall\alpha.\exists\vartheta.\ \mathcal{D}_{el}(\langle\sigma,\alpha\rangle) = \langle\vartheta,\alpha\rangle$

Note that **P3** implies that $\mathcal{D}_{el}$ is total.

The requirement **P2** on the compiler is actually more restrictive than simple compositionality w.r.t. $+$ and $\|$ (as explained in the introduction), and it can be justified as follows. Since the languages we study are one the extension of the other, and the differences between them consist of the kind of the guard g in the guarded statement $g \rightarrow A$, we can phrase the problem of the expressive power of these languages as the question:

can a guard operator $g \rightarrow$ in L' be expressed in terms of the operators of L?

In other words, whether a guard operator $g \rightarrow$ in L' can be translated into a context $c_g[\ ]$ in L such that for every process A in L' we have

$$\mathcal{D}(\mathcal{O}[\![A']\!]) = \mathcal{O}[\![A]\!]$$

where A' is obtained by replacing every occurrence of a guard operator $g \rightarrow$ by $c_g[\ ]$. This amounts to require the existence of a translation that only transforms the guard operators, and it is invariant with respect to $+$ and $\|$. Such a translation can be seen as a particular case of a compiler that satisfies **P2**.

Definition 5.1 *Let L, L' be two languages of the class* $\mathrm{CL}_{\mathcal{G}}$. *There exists a modular embedding (or, simply, embedding) from L' into L if equation (1) (in definition 4.1) holds, with $\mathcal{C}$ and $\mathcal{D}$ satisfying* **P1**, **P2**, *and* **P3** *above. The associated preorder relation will still be denoted by $\leq$.*

Of course, also for this notion of embedding, we have $\mathrm{CL}_{\emptyset} \leq \mathrm{CL}_{\mathcal{A}} \leq \mathrm{CL}_{\mathcal{AUT}}$. In the next section, we will show that this ordering is strict.

6 Separation results

In this section we prove $\mathrm{CL}_{\mathcal{AUT}} \not\leq \mathrm{CL}_{\mathcal{A}} \not\leq \mathrm{CL}_{\emptyset}$. In the proofs we make use of the following properties, which follow immediately from the definition of $\mathcal{M}$ and $\mathcal{O}$.

Proposition 6.1

1. *If $c.s \in \mathcal{M}[\![A]\!]$, then $c.\perp \in \mathcal{M}[\![A]\!]$.*
2. *If $c_1.c_2.\alpha \in \mathcal{M}[\![A]\!]$ and $c \in Con^*_{\mathcal{A}}$ and $\models \exists(c_1 \wedge c \wedge c_2)$ and $\alpha \neq dd$, then $c_1.c.c_2.\alpha \in \mathcal{M}[\![A]\!]$.*
3. *If $c.ss \in \mathcal{M}[\![G_1]\!]$, then $c.ss \in \mathcal{M}[\![G_1 + G_2]\!]$.*
4. *If $\langle\sigma_1, ss\rangle \in \mathcal{O}[\![A_1]\!]$ and $\langle\sigma_2, ss\rangle \in \mathcal{O}[\![A_2]\!]$ and $\models \exists(\sigma_1 \wedge \sigma_2)$, then $\langle\sigma_1 \wedge \sigma_2, ss\rangle \in \mathcal{O}[\![A_1 \parallel A_2]\!]$.*

6.1 $\mathrm{CL}_{\mathcal{AUT}} \not\leq \mathrm{CL}_{\mathcal{A}}$

In this section we prove that $\mathrm{CL}_{\mathcal{AUT}} \not\leq \mathrm{CL}_{\mathcal{A}}$. As explained in section 1.5, the essence of the proof is based on the following property $\pi_{\mathcal{A}}$, that is satisfied by the semantics of $Proc_{\mathcal{A}}$, and not by the one of $Proc_{\mathcal{AUT}}$. The property $\pi_{\mathcal{A}}$ says that a process in $Proc_{\mathcal{A}}$ that is ready to produce a constraint ϑ will fail if the environment first produces a constraint inconsistent with ϑ. This is due to the absence of tell in the guards: a process cannot make a choice based on the consistency of constraints to be told.

Proposition 6.2 *The semantics of $Proc_{\mathcal{A}}$ satisfies the following closure property:*

$\pi_{\mathcal{A}}$) *For each $A \in Proc_{\mathcal{A}}$*
if $c.\sigma_1^T.s \in \mathcal{M}[\![A]\!]$ and $\models \exists(c \wedge \sigma_2)$ and $\not\models \exists(c \wedge \sigma_1 \wedge \sigma_2)$, then $c.\sigma_2^A.ff \in \mathcal{M}[\![A]\!]$

Proof Let $A, c, \sigma_1, s, \sigma_2$ be such that the premises of the proposition hold. We observe that, since $\models \exists(c \wedge \sigma_2)$ and $\not\models \exists(c \wedge \sigma_1 \wedge \sigma_2)$, $\sigma_1 \neq \mathit{true}$ holds. Therefore σ_1^T must be generated by an application of one of the rules **C9** and **C3** in table 2. In case **C3** is applied, then A is of the form $\mathit{tell}(\sigma_1)$. In case that **C9** is applied, then A must be of the form $A_1 \parallel A_2$, since the processes of the form $A_1 + A_2$ in $\mathit{Proc}_{\mathcal{A}}$ cannot contain $\mathit{tell}(\sigma_1)$ in the guard (and therefore σ_1^T could not be generated). Moreover, the A_i (say, A_1) that occur in the premise of **C9** must, for the same reason, be either of the form $\mathit{tell}(\sigma_1)$ or $B_1 + B_2$. Applying recursively this reasoning, we deduce that A is of the form $(\ldots(\mathit{tell}(\sigma_1) \parallel \ldots) \parallel \ldots)$, and that the transition step is caused by an application of **C3** to $\mathit{tell}(\sigma_1)$, and propagated to A by a number of applications of **C9**.

Consider now the computation in which this transition step is replaced by an application of the rule **C13**, with assumption σ_2^A. We have that, as a next step, **C4** can be applied to $\mathit{tell}(\sigma_1)$, so obtaining a failure, which is propagated to A by a number of applications of **C11**. □

We reason now by contradiction, so we assume the existence of an embedding from $\mathrm{CL}_{\mathcal{A} \cup \mathcal{T}}$ to $\mathrm{CL}_{\mathcal{A}}$.

Lemma 6.3 *Assume* $\mathrm{CL}_{\mathcal{A} \cup \mathcal{T}} \leq \mathrm{CL}_{\mathcal{A}}$. *Then* $\mathcal{C}$ *satisfies the following property:*

If $A_i = \mathit{tell}(\vartheta_i) \rightarrow \mathit{tell}(\mathit{true})$ *for* $i = 1, 2$, *and* $\not\models \exists(\vartheta_1 \wedge \vartheta_2)$,

then there exist σ_1, σ_2 *such that* $\langle \sigma_i, ss \rangle \in \mathcal{O}[\![\mathcal{C}(A_i)]\!]$ *for* $i = 1, 2$, *and* $\not\models \exists(\sigma_1 \wedge \sigma_2)$.

Proof Let ϑ_i, A_i be such that the premises of the property hold. It is easy to see that

$$\mathcal{O}[\![A_i]\!] = \{\langle \vartheta_i, ss \rangle\} \quad \text{for } i = 1, 2, \tag{2}$$

and

$$\mathcal{O}[\![A_1 \parallel A_2]\!] = \{\langle \vartheta_1, \mathit{ff} \rangle\,,\ \langle \vartheta_2, \mathit{ff} \rangle\}. \tag{3}$$

As a consequence of (2), by **P1**, there exists $\sigma_1, \sigma_2, \alpha_1, \alpha_2$ such that

$$\langle \sigma_i, \alpha_i \rangle \in \mathcal{O}[\![\mathcal{C}(A_i)]\!] \quad \text{and } \mathcal{D}_{el}(\langle \sigma_i, \alpha_i \rangle) = \langle \vartheta_i, ss \rangle \quad \text{for } i = 1, 2.$$

Furthermore, by **P3**,

$$\alpha_1 = \alpha_2 = ss.$$

Assume now, by contradiction, that $\models \exists(\sigma_1 \wedge \sigma_2)$. By proposition 6.1(4) we have

$$\langle \sigma_1 \wedge \sigma_2, ss \rangle \in \mathcal{O}[\![\mathcal{C}(A_1) \parallel \mathcal{C}(A_2)]\!] = \text{ (by } \mathbf{P2}) \ \mathcal{O}[\![\mathcal{C}(A_1 \parallel A_2)]\!].$$

By **P3**, we then obtain

$$\mathcal{D}_{el}(\langle \sigma_1 \wedge \sigma_2, ss \rangle) = \langle \ldots, ss \rangle \in \mathcal{O}[\![A_1 \parallel A_2]\!],$$

that contradicts (3). □

Theorem 6.4 $\mathrm{CL}_{\mathcal{A} \cup \mathcal{T}} \not\leq \mathrm{CL}_{\mathcal{A}}$

Proof Assume, by contradiction, that $\mathrm{CL}_{\mathcal{A} \cup \mathcal{T}} \leq \mathrm{CL}_{\mathcal{A}}$ holds. Let ϑ_i, A_i be defined as in lemma 6.3. Let $A = A_1 + A_2$. By lemma 6.3 and by proposition 6.1(3) we have that there exist σ_1, σ_2 such that

$$\langle \sigma_i, ss \rangle \in \mathcal{O}[\![\mathcal{C}(A_1) + \mathcal{C}(A_2)]\!] = \text{ (by } \mathbf{P2}) \ \mathcal{O}[\![\mathcal{C}(A)]\!] \quad \text{for } i = 1, 2, \quad \text{and } \not\models \exists(\sigma_1 \wedge \sigma_2).$$

Since $\mathcal{O} = \mathcal{R} \circ \mathcal{M}$, we have that there exist $s_1, s_2 \in \mathcal{M}[\![\mathcal{C}(A)]\!] \cap (\mathit{Con}_{\mathcal{T}}^{*} \times \{ss\})$ such that $\not\models \exists(s_1 \wedge s_2)$. Consider a decomposition of s_1, s_2

$$s_1 = c_1 . \psi_1^T . s_1'$$

$$s_2 = c_2.\psi_2^T.s_2'$$

such that

$$\models \exists(c_1 \wedge c_2) \quad \text{and} \quad \models \exists(c_1 \wedge c_2 \wedge \psi_1) \quad \text{and} \quad \models \exists(c_1 \wedge c_2 \wedge \psi_2) \quad \text{and} \quad \not\models \exists(c_1 \wedge c_2 \wedge \psi_1 \wedge \psi_2).$$

By proposition 6.1(1) we have

$$c_1.\psi_1^T.\perp \in \mathcal{M}[\![\mathcal{C}(A)]\!],$$

therefore, by proposition 6.1(2)

$$c_1.\tilde{c}_2.\psi_1^T.\perp \in \mathcal{M}[\![\mathcal{C}(A)]\!]$$

holds, where $\tilde{c}_2$ is obtained from c_2 by turning the tell modes into ask. Finally, by proposition 6.2, we obtain

$$s \overset{\text{def}}{=} c_1.\tilde{c}_2\psi_2^A.\mathit{ff} \in \mathcal{M}[\![\mathcal{C}(A)]\!].$$

Analogously, by proposition 6.1(1) we have

$$c_2.\psi_2^T.\perp \in \mathcal{M}[\![\mathcal{C}(A)]\!]$$

and, by proposition 6.1(2)

$$s' \overset{\text{def}}{=} \tilde{c}_1.c_2.\psi_2^T.\perp \in \mathcal{M}[\![\mathcal{C}(A)]\!]$$

holds, where $\tilde{c}_1$ is obtained from c_1 by turning the tell modes into ask. By definition of $\tilde{\|}$, s can be composed with s' so obtaining

$$s\tilde{\|}s' = c_1.c_2.\psi_2^T.\mathit{ff} \in \mathcal{M}[\![\mathcal{C}(A)]\!]\tilde{\|}\mathcal{M}[\![\mathcal{C}(A)]\!] = \text{(by theorem 3.8) } \mathcal{M}[\![\mathcal{C}(A) \parallel \mathcal{C}(A)]\!].$$

Since $\mathcal{O} = \mathcal{R} \circ \mathcal{M}$, we have

$$\langle \mathit{store}(c_1.c_2.\psi), \mathit{ff}\rangle \in \mathcal{O}[\![\mathcal{C}(A) \parallel \mathcal{C}(A)]\!] = \text{(by } \mathbf{P2}\text{) } \mathcal{O}[\![\mathcal{C}(A \parallel A)]\!]. \tag{4}$$

Consider now the process $A \parallel A$. It is easy to see that

$$\mathcal{O}[\![A \parallel A]\!] = \{\langle \vartheta_1, ss\rangle \text{ , } \langle \vartheta_2, ss\rangle\}. \tag{5}$$

Since $\mathcal{D}_{el}$ must preserve the termination mode (**P3**), (4) and (5) together generate a contradiction. □

6.2 $\mathrm{CL}_{\mathcal{A}} \not\leq \mathrm{CL}_{\emptyset}$

In this section we prove that $\mathrm{CL}_{\mathcal{A}} \not\leq \mathrm{CL}_{\emptyset}$. Again, the essence of the proof is based on a property ($\pi_\emptyset$) satisfied by the semantics of $\mathit{Proc}_\emptyset$ and not by the one of $\mathit{Proc}_{\mathcal{A}}$. This property says that if a process can deadlock, then also the indeterministic choice between this process and another one may deadlock. Intuitively, this holds in $\mathit{Proc}_\emptyset$ (and not in $\mathit{Proc}_{\mathcal{A}}$) because of the absence of communication primitives in the guards, that causes the choice to be independent from the environment.

Proposition 6.5 *The semantics of $\mathit{Proc}_\emptyset$ satisfies the following closure property:*

$\pi_\emptyset$) *For each $G \in \mathit{Proc}_\emptyset$*
if $c.dd \in \mathcal{M}[\![G]\!]$ then $c.dd \in \mathcal{M}[\![G + G']\!]$.

Proof Let $G = g \rightarrow A \in \mathit{Proc}_\emptyset$, and let $c.dd \in \mathcal{M}[\![G]\!]$. Since g is of the form $\mathit{tell}(\mathit{true})$, c must be of the form:

$$c = \sigma_1^A.\dots.\sigma_k^A.\mathit{true}^T.c'$$

for some appropriate $\sigma_1 \ldots . \sigma_k$. Furthermore, by definition of $\mathcal{M}$

$$\perp \in \mathcal{M}[\![G']\!]$$

holds, hence, by proposition 6.1(2):

$$\sigma_1^A \ldots . \sigma_k^A . \perp \in \mathcal{M}[\![G']\!].$$

Therefore, by compositionality of $\mathcal{M}$, and by definition of $\tilde{+}$ we obtain

$$\begin{array}{rcl} c & = & \sigma_1^A \ldots . \sigma_k^A . \mathit{true}^T . c' \\ & = & \sigma_1^A \ldots . \sigma_k^A . (\mathit{true}^T . c' \mathbin{\tilde{+}} \perp) \\ & = & \sigma_1^A \ldots . \sigma_k^A . \mathit{true}^T . c' \mathbin{\tilde{+}} \sigma_1^A \ldots . \sigma_k^A . \perp \\ & \in & \mathcal{M}[\![G]\!] \tilde{+} \mathcal{M}[\![G']\!] \\ & = & \mathcal{M}[\![G+G']\!] \quad \text{(by theorem 3.8).} \end{array}$$

□

Theorem 6.6 $\mathrm{CL}_{\mathcal{A}} \not\leq \mathrm{CL}_{\emptyset}$.

Proof Assume, by contradiction, that $\mathrm{CL}_{\mathcal{A}} \leq \mathrm{CL}_{\emptyset}$ holds. Consider the constraints $\vartheta_1, \vartheta_2, \vartheta_3$ such that $\not\models \vartheta_3 \Rightarrow \vartheta_1$, $\models \vartheta_3 \Rightarrow \vartheta_2$, and let $A_1, A_2, A_3 \in \mathit{Proc}_{\mathcal{A}}$ be defined as follows:

$$A_1 = \mathit{ask}(\vartheta_1) \rightarrow \mathit{tell}(\mathit{true})$$

$$A_2 = \mathit{ask}(\vartheta_2) \rightarrow \mathit{tell}(\mathit{true})$$

$$A_3 = \mathit{tell}(\vartheta_3).$$

It is easy to see that

$$\mathcal{O}[\![A_1 \parallel A_3]\!] = \{\langle \vartheta_3, \mathit{dd} \rangle\} \tag{6}$$

and

$$\mathcal{O}[\![(A_1+A_2) \parallel A_3]\!] = \{\langle \vartheta_3, \mathit{ss} \rangle\}. \tag{7}$$

From (6) we have that there exists σ such that

$$\langle \sigma, \mathit{dd} \rangle \in \mathcal{O}[\![\mathcal{C}(A_1 \parallel A_3)]\!] = \text{ (by } \mathbf{P2}\text{) } \mathcal{O}[\![\mathcal{C}(A_1) \parallel \mathcal{C}(A_3)]\!].$$

Since $\mathcal{O} = \mathcal{R} \circ \mathcal{M}$, for appropriate $c_1.\alpha_1 \in \mathcal{M}[\![\mathcal{C}(A_1)]\!]$ and $c_2.\alpha_2 \in \mathcal{M}[\![\mathcal{C}(A_2)]\!]$ we have

$$c_1.\alpha_1 \tilde{\parallel} c_2.\alpha_2 \in \mathit{Con}_T^* \times \{\mathit{dd}\}.$$

By definition of $\tilde{\parallel}$, we have two cases:

case 1) $\alpha_1 = \mathit{dd}$, or

case 2) $\alpha_1 = \mathit{ss}$, and $\alpha_2 = \mathit{dd}$.

In both cases, we obtain

$$c_1.\alpha_1 \in \mathcal{M}[\![\mathcal{C}(A_1)+\mathcal{C}(A_2)]\!]$$

(in case 1 it follows from proposition 6.5, in case 2 from proposition 6.1(3)). Therefore, we have

$$c_1.\alpha_1 \tilde{\parallel} c_2.\alpha_2 \in \mathcal{M}[\![(\mathcal{C}(A_1)+\mathcal{C}(A_2)) \parallel \mathcal{C}(A_3)]\!] = \text{ (by } \mathbf{P2}\text{)} \mathcal{M}[\![\mathcal{C}((A_1+A_2) \parallel A_3)]\!].$$

This implies $\langle \sigma, \mathit{dd} \rangle \in \mathcal{O}[\![\mathcal{C}((A_1+A_2) \parallel A_3)]\!]$ that, together with (7), gives a contradiction (since $\mathcal{D}_{el}$ must preserve the termination mode (**P3**)). □

7 Interpretation of the results.

In the previous section we have seen that the three conditions required on $\mathcal{C}$ and $\mathcal{D}$ increase drastically the separation power of definition 4.1. The basic definition of embedding is in fact a variant of the classical notion of *language simulation*:

Definition 7.1 *A language L' can be simulated by L iff there exists an encoder $\mathcal{E}$, from the inputs of L' to the inputs of L, a compiler $\mathcal{C}$ from the programs of L' to the programs of L, and a decoder $\mathcal{D}$ from the outputs of L to the outputs of L' such that, for every program P in L' and for every input i: $\mathcal{D}(\mathcal{C}(P)(\mathcal{E}(i))) = P(i)$.*

It is well known that this definition is satisfied by any pair of languages Turing-complete.

In our opinion there are essentially two reasons why our definition of modular embedding is stronger then definition 7.1:

- The requirement of termination invariance implies the possibility to observe also failure and deadlock (beside success) and this gives more information about the internal structure of the program [1]. In other words, whilst the success set is completely characterized by the minimal model (that is related to the extensional definition of a function) failure and deadlock require to consider also other models. A program containing a loop can have the same minimal model (and the same success set) of a program with no loops, but the other models (and their failure sets) are different.
- The notion of indeterminism in concurrent languages is different from the notion of nondeterminism in sequential languages. In the theory of computation the nondeterministic languages are seen just as mean to express in a compact way algorithms for search problems. In the observables, only the successful outcomes are taken into account (all the paths that do not solve the problem are eliminated). As a consequence, nondeterministic languages can be easily implemented (via backtracking) into deterministic ones: The nondeterministic Turing machine is equivalent to the deterministic one. This would not be possible when also failures are relevant, that is the case of indeterministic languages.

8 Future work

We are currently investigating [6] the application of the notion of modular embedding to separate other pairs of languages in the FCP family [13]. However, the way in which this notion is formulated makes it suitable for any concurrent framework, and we believe that it can be fruitfully used also for comparing the expressive power of concurrent languages out of the logical paradigm.

Another direction of research is the investigation of different conditions on the compiler and the decoder, that bring to other notions of embedding.

Acknowledgements We thank Joost Kok, Jan Rutten and Ehud Shapiro for stimulating discussions and encouragements. We acknowledge the department of Software Technology of CWI for providing a stimulating working environment. Finally, we thank Krzysztof Apt for having suggested us relevant literature on the subject.

References

[1] P. America, J. de Bakker, J.N. Kok, and J. Rutten. Denotational semantics of a Parallel Object-Oriented Language. *Information and Computation*, 83(2):152–205, 1989.

[2] L. Bougé. Metric semantics for concurrency. *Acta Informatica*, 25:179–201, 1988.

[1]This observation is due to H. Blair.

[3] F.S. de Boer and C. Palamidessi. Concurrent logic languages: Asynchronism and language comparison. In *Proc. of the North American Conference on Logic Programming*, Series in Logic Programming, pages 175–194. The MIT Press, 1990. Full version available as technical report TR 6/90, Dipartimento di Informatica, Università di Pisa.

[4] F.S. de Boer and C. Palamidessi. On the asynchronous nature of communication in concurrent logic languages: A fully abstract model based on sequences. In J.C.M. Baeten and J.W. Klop, editors, *Proc. of Concur 90*, volume 458 of *Lecture Notes in Computer Science*, pages 99–114, Amsterdam, 1990. Springer-Verlag. Full version available as report at the Technische Universiteit Eindhoven.

[5] F.S. de Boer and C. Palamidessi. A fully abstract model for concurrent constraint programming. In *Proc. of TAPSOFT*, 1991.

[6] F.S. de Boer, C. Palamidessi, and E. Shapiro. Comparing the languages of the FCP family. In preparation.

[7] M. Gabbrielli and G. Levi. Unfolding and fixpoint semantics for concurrent constraint logic programs. In Springer-Verlag, editor, *Proc. of the Second Int. Conf. on Algebraic and Logic Programming*, Lecture Notes in Computer Science, Nancy, France, 1990.

[8] H. Gaifman, M. J. Maher, and E. Shapiro. Reactive Behaviour semantics for Concurrent Constraint Logic Programs. In E. Lusk and R. Overbeck, editors, *North American Conference on Logic Programming*, 1989.

[9] R. Gerth, M. Codish, Y. Lichtenstein, and E. Shapiro. Fully abstract denotational semantics for Concurrent Prolog. In *Proc. of the Third IEEE Symposium on Logic In Computer Science*, pages 320–335. IEEE Computer Society Press, New York, 1988.

[10] M. J. Maher. Logic semantics for a class of committed-choice programs. In Jean-Louis Lassez, editor, *Proc. of the Fourth International Conference on Logic Programming*, Series in Logic Programming, pages 858-876, Melbourne, 1987. The MIT Press.

[11] V.A. Saraswat. *Concurrent Constraint Programming Languages.* PhD thesis, Carnegie-Mellon University, january 1989. Published by The MIT Press, U.S.A., 1990.

[12] V.A. Saraswat and M. Rinard. Concurrent constraint programming. In *Proc. of the seventeenth ACM Symposium on Principles of Programming Languages*, pages 232–245. ACM, New York, 1990.

[13] E.Y. Shapiro. The family of concurrent logic programming languages. *ACM Computing Surveys*, 21(3):412–510, 1989.

[14] K. Ueda. Guarded Horn Clauses. In E. Y. Shapiro, editor, *Concurrent Prolog: Collected Papers.* The MIT Press, 1987.