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 e Scienze dell'Informazione,
Università di Genova,
Viale Benedetto XV, 3, 16132 Genova, Italy
email: catuscia@.di.unipi.it

Note: This work was developed during the stay of Catuscia Palamidessi at the Centre for Mathematics and Computer Science, with the support of the Italian CNR (Consiglio Nazionale delle Ricerche) and the ESPRIT project 3020 (Integration). The research of Frank S. de Boer was partially supported by the Dutch REX (Research and Education in Concurrent Systems) project.

Running head: On Language Comparison.

Catuscia Palamidessi
Dipartimento di Informatica e Scienze dell'Informazione,
Viale Benedetto XV, 3, 16132 Genova, Italy

Abstract

This paper addresses the problem of defining a formal tool to compare the expressive power of different concurrent constraint languages. We refine the notion of embedding by adding some "reasonable" conditions, suitable for concurrent frameworks. The new notion, called *modular embedding*, is used to define a preorder among these languages, representing different degrees of expressiveness. 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, while the converses hold.

$\mathcal{A}, \mathcal{C}, \mathcal{D}, \mathcal{G}, \mathcal{M}, \mathcal{O}, \mathcal{P}, \mathcal{R}, \mathcal{T}$: In calligraphic style.

ss, ff, dd: In slanted style.

$\Sigma, \Gamma, \vartheta, \sigma, \lambda, \psi, \tau, \alpha$.

$\bigcup, \cup, \cap, \emptyset, \in$

$\models, \not\models, \wedge, \exists$

$+, \|, \tilde{+}, \tilde{\|}, \rightarrow$

$\longrightarrow_W, \longrightarrow_\emptyset, \longrightarrow_W^*, \longrightarrow_\emptyset^*$

$\langle, \rangle, [\![,]\!], \lceil, \rceil$

$\circ, \perp, \leq, \not\leq$

All reasonable programming languages are equivalent, since they are Turing-complete. However, if the differences between languages were not material, we would not have invented so many of them.

Shapiro, (1989).

1 Introduction

From a computational point of view, all "reasonable" programming languages are equivalent, as they express the same class of functions: the recursive functions. On the other hand, in the practice of programming, certain languages are considered to be more "powerful" than others with respect to the capability to express control and data structures. In the field of sequential languages there has been already since a long time a line of research aiming to formalize the notion of "expressive power" (Landin, 1966; Reynolds, 1970; Paterson and Hewitt, 1970; Chandra and Manna, 1975; Steele and Sussman, 1976; Reynolds, 1981; and Felleisen, 1990). The various approaches agree in considering a language L more expressive than L' if the constructs of L' can be translated in L without requiring a "global reorganization of the entire program" (Felleisen, 1990), i.e., compositionally. Of course, the translation should allow to retrieve the meaning of the original program. In this respect, Felleisen (1990) argues that a basic requirement is the preservation of the (successful) termination.

Sequential languages are used to program *transformational systems*, namely systems which transform an input into an output, if deterministic, or into a set of outputs, if nondeterministic. The nondeterminism does not contribute substantially to enrich the above described notion of simulation, because in the transformational view only successfully terminating computations are considered, while failing computation are discarded. This makes sense because the system runs "in isolation", hence failing computations can be effectively discarded by means of a backtracking mechanism.

When we move to the field of concurrent languages, the notion of termination must be reconsidered. In fact these languages are used to program *interactive systems*, where nondeterminism plays an essentially different role: each possible computation represents a different evolution of a system of interacting processes. These interactions are based upon some synchronization mechanism which influences the choices of a process. As a consequence, it does not make sense for a process to backtrack, because the evolution of the environment might have changed the conditions upon which the choice has been made. For this reason we must take into account also non successful computations. Of particular importance is the situation in which all processes are suspended, which is called *deadlock*.

Because of the interaction mechanisms, models of concurrent languages require more sophisticated structures. In particular, modeling deadlock requires encoding the branching structures of a process (process graph). From the point of view of language comparison, this richer notion of model induces a reconsideration of the notion of expressiveness.

In the field of process algebra, Vaandrager (1992) classifies various notions of expressiveness, increasingly more refined:

- the capability to simulate Turing machines,
- the capability to specify effective process graphs, up to some notion of equivalence, and
- the capability to express effective operations on graphs by means of constructs of the language.

While all "reasonable" languages are universal according to the first criterion, this is not the case for the remaining ones. In particular, Baeten, Bergstra, and Klop (1987) show that no language with a recursive operational semantics is universal, namely able to express all effective process graphs, up to strong bisimulation equivalence, and Vaandrager (1992) generalizes this result to trace equivalence. Therefore it makes sense to characterize the expressive power of a language by considering the class of graphs which can be specified by it. Bergstra and Klop (1986) provide a detailed classification of various ACP sublanguages, showing the contribute of each operator to enlarge the set of expressible graphs.

However, universality is achieved for ACP_τ by considering a more abstract notion of equivalence called weak bisimulation congruence (Baeten, Bergstra, and Klop, 1987), and for MEIJE and SCCS with unguarded recursion up to strong bisimulation (de Simone, 1985), This last result does not contradict the results of Baeten, Bergstra, and Klop (1987) and Vaandrager (1992), because unguarded recursion induces an infinitely branching, possibly not recursive, operational semantics. Parrow (1989) shows a nice result for the class of finite process graphs. Every finite graph can be expressed as a parallel composition of three kinds of elementary processes: synchronizers, arbiters and alternators. This holds for all weak equivalences (i.e. abstracting from τ steps) between traces and bisimulation.

Concerning the third notion, de Simone (1985) characterizes a very general format of transition rules and shows that every operator defined by those rules can be expressed in MEIJE and SCCS. Parrow (1989) shows that the class of *finite-state, asynchronous* operators, which includes nondeterministic choice, hiding and merge, can be expressed as parallel composition of synchronizers, arbiters and alternators.

In the dataflow framework the expressiveness of various operators like fair merge, angelic merge and infinity-fair merge is investigated in (McAllester, Panangaden, and Shanbhogue, 1988; Panangaden and Stark, 1988; Stark, 1990; and Panangaden and Shanbhogue, 1992). The notion of expressiveness used is based on the capability to express certain properties of relations between input and output streams of data.

In this paper we study a *relative* notion of expressive power, namely we measure the expressive power of a language relatively to another language. The method of comparison we adopt is partly based on the third criterion illustrated above, in the sense that we consider whether the operators of a language can be expressed in the other language. However the "relative" version of the third criterion above would require a statement in the object language and the correspondent statement in the target language to have the same semantics. In our opinion this condition is too restrictive, because it does not allow to abstract from "implementation details". Therefore we relax this restriction and only require the existence of an abstraction from the observables of the second language to the observables of the first.

1.1 The method

A natural way to compare the expressive power of two languages (and their relative observation criteria) 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 (1989), 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* $\mathcal{O} : Prog_L \rightarrow Obs$ and $\mathcal{O}' : Prog_{L'} \rightarrow Obs'$, 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.

Figure 1: basic embedding.

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

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, in our opinion, are rather "reasonable" in a concurrent framework.

A first remark is the following. In a concurrent language, where nondeterminism plays an important role, the domain of the observables (Obs) is in general a powerset (i.e. the elements O of Obs 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 $\quad \forall O \in Obs.\, \mathcal{D}(O) = \{\mathcal{D}_{el}(o) \mid o \in O\}$

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 information 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 nondeterministic 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 nondeterministic choice operators in L and L' respectively.)

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

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

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 it is clear from the context.

1.2 The framework

In order to show the application of our method we consider a class of concurrent languages based on the same data structures and on the same choice and parallel operators. The differences are relative to the way 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.

The class we consider is the class of concurrent constraint languages (Maher, 1987; Saraswat, 1989; and Saraswat and Rinard, 1990). This class will be denoted by $\mathrm{CC}_{\mathcal{G}}$, where CC stands for Concurrent Constraint and $\mathcal{G}$ is a parameter which denotes the set of communication primitives which 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 constraint, 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 (Saraswat, 1989), and it includes a large number of flat concurrent logic languages. For instance, when the constraint system is the set of equations on the Herbrand universe, $\mathrm{CC}_{\mathcal{A}\cup\mathcal{T}}$ corresponds to Atomic Herbrand (Saraswat, 1989), to ccH (Gabbrielli and levi, 1990) and to the language of Gaifman, Maher, and Shapiro (1989), which is a refined version of Flat CP (Shapiro, 1989). The language $\mathrm{CC}_{\mathcal{A}}$ corresponds to Eventual Herbrand (Saraswat, 1989) and to Flat GHC in its earlier version (Ueda, 1987).

1.3 Technical results

We show that the definition of modular embedding "makes sense" by proving that $\mathrm{CC}_{\emptyset}$ cannot embed $\mathrm{CC}_{\mathcal{A}}$ (Flat GHC) and that $\mathrm{CC}_{\mathcal{A}}$ cannot embed $\mathrm{CC}_{\mathcal{A}\cup\mathcal{T}}$ (Flat CP) (*separation results*). As far as we know, these separation (i.e. non-embeddability) results have not been proved before, although they were believed to hold by researchers in the area.

1.4 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 Figures 1 and 2.

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

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

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

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

More specifically, we use the compositional operational semantics developed by de Boer and Palamidessi (1991a) for a similar class of languages. This model is rather simple, therefore it is suitable for the investigation on the properties of the different languages of this class. We prove that the semantics of $\mathrm{CC}_{\mathcal{A}}$ [$\mathrm{CC}_{\emptyset}$] satisfies a certain closure property that in general is not satisfied by $\mathrm{CC}_{\mathcal{A}\cup\mathcal{T}}$ [$\mathrm{CC}_{\mathcal{A}}$]. This closure property gives rise to an observable distinction that cannot be hidden by the decoder, when **P1**,**P2** and **P3** are required.

1.5 Related work

The expressiveness of various communication and synchronization constructs have been investigated on the basis of some specific analysis of the operational properties (Herlihy, 1988).

Bougé (1988a) has presented 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. These results are based on the capability to express of certain algorithms. Namely, he has shown that in certain communication graphs one dialect admits symmetric solutions to the *election problem*, while another dialect doesn't. These solutions are required to satisfy certain conditions about termination which closely correspond to our notion of termination invariance. In order to obtain separation results, Bougé introduced a notion of compiler preserving the parallel operator and the topology of the network, namely the structure of the communication graph. This last restriction would be too strong for concurrent constraint languages, and also difficult to formulate, since the communication network evolves dynamically. Later on, Bougé (1987) generalized the above results and he showed that similar techniques can be applied to Petri Nets (Bougé, 1988b).

The notion of embedding was first proposed by Shapiro (1989) as a method for language comparison, and refined by de Boer and Palamidessi (1990) and by Shapiro (1991) so to make it non-trivial. De Boer and Palamidessi (1990) defined a preliminary version of the notion of modular embedding (called *uniform embedding*), and applied it to show separation results between Flat CP, Flat GHC and Oc. These separation results however depend upon a condition on the decoder (*consistency-preserving*), tailored for logic/constraint languages. The present paper makes the notion of modular embedding general and applicable to any concurrent language with explicit choice and parallel operators. Using this notion of modular embedding, de Boer and Palamidessi (1991b) have extablished the same separation results for the CSP dialects obtained by Bougé in his framework. Furthermore also Asynchronous CSP dialects have been compared, and shown to be less sensitive to the kind of guard: output guards do not increase the expressive power. Putting these results together it follows that Asynchronous CSP is less expressive than CSP. a have compared the Asynchronous CSP dialects and have showed that this class is less sensitive to the definition of the guards. Finally, they have proved that Asynchronous CSP is strictly less expressive than CSP.

Shapiro (1991) investigated three different notions of embedding. The first one (sound embedding) is equivalent to the original definition in (Shapiro, 1989) plus the requirement of compositionality with respect to the parallel operator. Therefore it is equivalent to our notion if we drop the conditions on the decoder and the compositionality w.r.t. the choice operator. The second one (faithful embedding) requires additionally the compiler to translate equivalent statements into equivalent ones. The third notion (fully abstract embedding) requires the compiler to preserve the non-equivalence of statements with respect to the fully abstract semantics. This notion of embedding has been applied to show various positive and negative results in (Shapiro, 1991) and (Shapiro, 1992).

1.6 Plan of the paper

This paper is organized as follows. Next section introduces the class $\mathrm{CC}_{\mathcal{G}}$ 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 $\mathrm{CC}_{\emptyset}$ 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 $\mathrm{CC}_{\mathcal{A}}$, $\mathrm{CC}_{\mathcal{A}}$ (Flat GHC), and $\mathrm{CC}_{\mathcal{A}\cup\mathcal{T}}$ (Flat CP). Finally, in Section 7 we discuss our results and future work.

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

In this section we present the class of languages $\mathrm{CC}_{\mathcal{G}}$, which is a variant of the languages presented in (Saraswat, 1989; Saraswat and Rinard, 1990; and de Boer and Palamidessi, 1991a). The main difference with (Saraswat and Rinard, 1990; and de Boer and Palamidessi, 1991a) is that we do not consider here an explicit hiding operator, because the results we present do not depend upon it. We refer to (Shapiro, 1989) for a detailed introduction to concurrent constraint languages.

2.1 The syntax

A constraint system consists of a set of elements representing items of partial information, together with two relations representing *consistency* and *entailment*. For the sake of simplicity we consider here systems based on first-order languages, however our results are independent from the choice of the constraint system. 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 $\Sigma = (Var, Fun, Pred_C)$. 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 $\mathrm{CC}_{\mathcal{G}}$ 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

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

and

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

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* $Proc_{\mathcal{G}}$ in $\mathrm{CC}_{\mathcal{G}}$, with typical elements $A, B, \ldots$, is described by the grammar in Table 1.

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 whether 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 $\|$ represents the parallel operator. A *guarded process* $g \to A$ 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 nondeterministic choice is represented by $+$ and it is guarded, namely, $+$ applies to guarded processes and selects those ones whose guard is

Table 1: The syntax of processes.

Processes	$A ::= ask(\vartheta) \mid tell(\vartheta) \mid A \parallel A \mid G \mid p(\vec{t}) \mid W; A$
Guarded statements	$G ::= g \rightarrow A \mid G + G$
Programs	$W ::= p(\vec{x}) \text{ :- } A \mid W, W$

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.* We assume that the variables which occur in W do not occur in A. This restriction, together with the dynamic instantiation mechanism (see next paragraph) is sufficient to deal correctly with the scope of the variables. The following example shows the necessity of this restriction. Consider $W = \{p(x) \text{ :- } \{q(x) \text{ :-} tell(true)\}; q(f(x))\}$ with procedure call $p(a)$. Replacing the formal parameter x by *a everywhere in W* would cause a failure at the moment of the internal procedure call.

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 $\mathrm{CC}_{\mathcal{G}}$ 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*, *ff* 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 2.

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 (Saraswat, and Rinard, 1990; and de Boer and Palamidessi, 1991a) 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.* In the sequel, $\mathcal{P}$ will denote the powerset operation.

Definition 2.1 *The observables of the class* $\mathrm{CC}_{\mathcal{G}}$ *are given by the function* $\mathcal{O} : Proc_{\mathcal{G}} \rightarrow Obs$, *where* $Obs = \mathcal{P}(Con \times \{ss, ff, dd\})$, *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 the relation* $\longrightarrow_{\emptyset}$ *in the transition system* T..

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

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

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

Table 2: The Transition System T. Note that $A, A', \ldots B, B'$ range over processes.

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	$\langle tell(\vartheta), \sigma \rangle \longrightarrow_W \langle ff, \sigma \rangle$	**if** $\not\models \exists(\sigma \wedge \vartheta)$
R5	$\langle p(\vec{t}), \sigma \rangle \longrightarrow_W \langle A, \sigma \rangle$	**if** $p(\vec{t})$:- $A \in Inst(W)$
R6	$\langle p(\vec{t}), \sigma \rangle \longrightarrow_W \langle ff, \sigma \rangle$	**if** $p(\vec{t})$:- $\ldots \notin Inst(W)$
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** $\alpha \in \{ff, dd\}$
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	$\dfrac{\langle A, \sigma \rangle \longrightarrow_W \langle dd, \sigma \rangle \quad \langle B, \sigma \rangle \longrightarrow_W \langle \alpha, \sigma \rangle}{\begin{array}{c} \langle A \parallel B, \sigma \rangle \longrightarrow_W \langle \alpha, \sigma \rangle \\ \langle B \parallel A, \sigma \rangle \longrightarrow_W \langle \alpha, \sigma \rangle \\ \langle A+B, \sigma \rangle \longrightarrow_W \langle dd, \sigma \rangle \\ \langle B+A, \sigma \rangle \longrightarrow_W \langle dd, \sigma \rangle \end{array}}$	**if** $\alpha \in \{ff, dd\}$
R11	$\dfrac{\langle A, \sigma \rangle \longrightarrow_W \langle ff, \sigma \rangle}{\begin{array}{l} \langle A \parallel B, \sigma \rangle \longrightarrow_W \langle ff, \sigma \rangle \\ \langle B \parallel A, \sigma \rangle \longrightarrow_W \langle ff, \sigma \rangle \end{array}}$	
R12	$\dfrac{\langle A, \sigma \rangle \longrightarrow_W \langle ff, \sigma \rangle \quad \langle B, \sigma \rangle \longrightarrow_W \langle ff, \sigma \rangle}{\begin{array}{c} \langle A+B, \sigma \rangle \longrightarrow_W \langle ff, \sigma \rangle \\ \langle B+A, \sigma \rangle \longrightarrow_W \langle ff, \sigma \rangle \end{array}}$	

developed by de Boer and Palamidessi (1990) and (1991a) 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 more details we refer to (de Boer and Palamidessi, 1991a)

The behaviour of a process is described as a sequence of *interactions* with its environment (the other parallel processes). Interactions are modelled 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* $Con_T = \{\vartheta^T \mid \vartheta \in Con\}$.
- *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 $T^C = (Conf^C, \longrightarrow^C_W)$. 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* : $Con^*_{AT} \to Con$ *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 3 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 auxiliary termination mode, $\perp$, that represents an "unfinished" computation. See (de Boer and Palamidessi, 1991a) for a more detailed explanation of the reasons to introduce $\perp$.

We denote the set of sequences as $Seq = Con^*_{AT}.\{ss, ff, dd, \perp\}$, the typical element will be represented by s. The semantical domain of $\mathcal{M}$, $\mathcal{P}(Seq)$, will be denoted by P.

Definition 3.3 *The semantics* $\mathcal{M} : Proc_{\mathcal{G}} \to P$ *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 from its denotation. This mapping is usually not injective, i.e. its application abstracts from some of the information encoded in the semantical domain. For this reason it is called an *abstraction operator*.

To show the correctness of $\mathcal{M}$, let's first consider which information (encoded by $\mathcal{M}$) is 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 labelled 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 $\mathcal{R}$.

Definition 3.4

Table 3: The Transition System T^C. Note that $A, A', \ldots B, B'$ range over processes.

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	$\dfrac{\langle A, c\rangle \longrightarrow_W \langle dd, c\rangle \quad \langle B, c\rangle \longrightarrow_W \langle \alpha, c\rangle}{\begin{array}{c}\langle A \parallel B, c\rangle \longrightarrow_W \langle \alpha, c\rangle \\ \langle B \parallel A, c\rangle \longrightarrow_W \langle \alpha, c\rangle \\ \langle A{+}B, c\rangle \longrightarrow_W \langle dd, c\rangle \\ \langle B{+}A, c\rangle \longrightarrow_W \langle dd, c\rangle\end{array}}$	**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	$\dfrac{\langle A, c\rangle \longrightarrow_W \langle ff, c\rangle \quad \langle B, c\rangle \longrightarrow_W \langle ff, c\rangle}{\begin{array}{c}\langle A{+}B, c\rangle \longrightarrow_W \langle ff, c\rangle \\ \langle B{+}A, c\rangle \longrightarrow_W \langle ff, c\rangle\end{array}}$	
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** $\models \exists(c \wedge \vartheta)$

- *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*

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

Note that $\mathcal{R}$ preserves the termination mode. The 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*

$$\exists c' \ \langle A, c \rangle \longrightarrow_W \langle A', c' \rangle \ \textit{ is a } \mathbf{Ci} \textit{ transition step in } T^C\textit{, and } store(c') = \sigma$$

iff

$$\langle A, store(c) \rangle \longrightarrow_W \langle A', \sigma \rangle \ \textit{ is a } \mathbf{Ri} \textit{ transition step in } T.$$

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. Let c contain only tell constraints and let $\alpha \in \{ss, ff, dd\}$. Then

$$c.\alpha \in \mathcal{M}[\![A]\!]$$

iff 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 the rule **C13** is never used,
iff (by Lemma 3.5 there exists in T a derivation

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

iff

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

□

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 by Gerth *et al.* (1988), 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 familiar interleaving operator, the difference is that it applies to sequences containing already all the information concerning the way in which processes interleave (the assumptions specify "where" and "what"). Hence the application of $\tilde{\|}$ amounts to verifying 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 labelled by tell whenever they are labelled 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 *operator* $\tilde{\|} : Seq \times Seq \to Seq$ *and the function* $\tilde{+} : Seq \times Seq \to P$ *are defined as follows*
  - $s_1 \tilde{\|} s_2 = s_2 \tilde{\|} s_1$
  - $(\vartheta^A.s_1)\tilde{\|}(\vartheta^\ell.s_2) = \vartheta^\ell.(s_1\tilde{\|}s_2)$
  - $ss\tilde{\|}\alpha = \alpha$
  - $f\!f\tilde{\|}\alpha = f\!f$
  - $dd\tilde{\|}dd = dd$
  - $\perp \tilde{\|} \perp = \perp$
  - $s_1\tilde{+}s_2 = s_2\tilde{+}s_1$
  - $\vartheta^T.s_1 \in (\vartheta^T.s_1)\tilde{+}s_2$
  - $(\vartheta^A.s_1)\tilde{+}(\vartheta^A.s_2) = \{\vartheta^A.s \mid s \in s_1\tilde{+}s_2\}$
  - $ss \in ss\tilde{+}\alpha$
  - $\alpha \in f\!f\tilde{+}\alpha$
  - $dd\tilde{+}dd = \{dd\}$
  - $\perp \tilde{+} \perp = \{\perp\}$
- *The operators on sets of sequences* $\tilde{\|}, \tilde{+} : P \times P \to P$ *are defined by:*

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

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

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

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

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 proof proceeds by an 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*

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

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

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

Consider now the program

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

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

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

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

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

4 Basic embedding

In this section we discuss the notion of *embedding* proposed by Shapiro (1989). We rephrase his notion in order to deal with processes rather than with programs.

Definition 4.1 (Embedding (Shapiro, 1989)) *Let L, L' be two (concurrent) languages, let $Proc_L$, $Proc_{L'}$ be the respective sets of processes, and let $\mathcal{O} : Proc_L \rightarrow Obs$, $\mathcal{O}' : Proc_{L'} \rightarrow Obs'$ be their observation criteria. The language L embeds L' iff there exists a compiler $\mathcal{C} : Proc_{L'} \rightarrow Proc_L$, and a decoder $\mathcal{D} : Obs \rightarrow Obs'$ such that, for every $A \in Proc_{L'}$, the following equation holds*

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

or, in other words, the diagram of Figure 1 in Section 1.1 (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).

Figure 4: transitivity of $\leq$.

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

Remark 4.2 *If $L' \subseteq L$ then $L' \leq L$. Therefore, $\mathrm{CC}_{\emptyset} \leq \mathrm{CC}_{\mathcal{A}} \leq \mathrm{CC}_{\mathcal{A}\cup\mathcal{T}}$. Note that the notion of observables (cfr. Definition 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 $\mathrm{CC}_{\mathcal{G}}$ 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 constraint languages. This will be done by requiring $\mathcal{C}$ and $\mathcal{D}$ to satisfy certain properties, specific for concurrency.

As discussed in Section 1.1, it is reasonable to require $\mathcal{D}$ to be defined *elementwise*. Therefore the first requirement is the existence of a partial mapping $\mathcal{D}_{el} : Obs \rightarrow Obs$ 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 *Assume that the underlying constraint system is an Herbrand system, i.e. it includes an equality theory which coincide with syntactical identity, and that it contains at least one non-constant symbol. Then all languages of the class* $\mathrm{CC}_{\mathcal{G}}$ *can be embedded into* $\mathrm{CC}_{\emptyset}$ *(with a decoder satisfying* **P1***).*

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

Define the compiler $\mathcal{C} : Proc_{\mathcal{G}} \rightarrow \mathrm{CC}_{\emptyset}$ as follows.

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

where

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

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

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

□

5 Modular embedding

In this section we further refine the notion of embedding. Besides **P1**, we require the compiler and the decoder to satisfy the properties discussed in Section 1.1, namely *compositionality* with respect to $\parallel$ and $+$, and preservation of the termination mode (*termination invariance*).

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

P2 $\forall A, B \in Proc_{\mathcal{G}}.\; \mathcal{C}(A \parallel B) = \mathcal{C}(A) \parallel \mathcal{C}(B)$ 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 $\parallel$ (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 $\mathrm{CC}_{\mathcal{G}}$. 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 $\mathrm{CC}_{\emptyset} \leq \mathrm{CC}_{\mathcal{A}} \leq \mathrm{CC}_{\mathcal{A}\cup\mathcal{T}}$. In the next section, we will show that this ordering is strict.

6 Separation results

In this section we prove $\mathrm{CC}_{\mathcal{A}\cup\mathcal{T}} \not\leq \mathrm{CC}_{\mathcal{A}} \not\leq \mathrm{CC}_{\emptyset}$. 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 $c \in Con_A^*$ 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 $\mathrm{CC}_{\mathcal{A}\cup\mathcal{T}} \not\leq \mathrm{CC}_{\mathcal{A}}$

In this section we prove that $\mathrm{CC}_{\mathcal{A}\cup\mathcal{T}} \not\leq \mathrm{CC}_{\mathcal{A}}$. As explained in Section 1.4, the essence of the proof is based on the following property $\boldsymbol{\pi}$, that is satisfied by the semantics of $Proc_{\mathcal{A}}$, and not by the one of $Proc_{\mathcal{A}\cup\mathcal{T}}$. The property $\boldsymbol{\pi}$ 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:*

$\boldsymbol{\pi}$) *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 $c.\sigma_2^A.\mathit{ff} \in \mathcal{M}[\![A]\!]$

Proof Let $A, c, \sigma_1, s, \sigma_2$ be such that the premises of the proposition hold. Let A' be the process to which A has reduced after the sequence of transitions corresponding to c. We observe that $\sigma_1 \neq \mathit{true}$ holds, because $\models \exists(c \wedge \sigma_2)$ and $\not\models \exists(c \wedge \sigma_1 \wedge \sigma_2)$. Therefore σ_1^T must be generated by an application of one of the rules **C9** and **C3** in Table 3. In case **C3** is applied, then A' is of the form $tell(\sigma_1)$. 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 $Proc_{\mathcal{A}}$ cannot contain $tell(\sigma_1)$ 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 $tell(\sigma_1)$ or $B_1 \parallel B_2$. Applying recursively this reasoning, we deduce that A' is of the form $(\ldots(tell(\sigma_1) \parallel \ldots) \parallel \ldots)$, and that the transition step is caused by an application of **C3** to $tell(\sigma_1)$, 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 $tell(\sigma_1)$, thus 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 $\mathrm{CC}_{\mathcal{A}\cup\mathcal{T}}$ to $\mathrm{CC}_{\mathcal{A}}$.

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

If $A_i = tell(\vartheta_i) \rightarrow tell(true)$ 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

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

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, to obtain a contradiction, that $\models \exists(\sigma_1 \wedge \sigma_2)$. By Proposition 6.1(4) we have

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

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 $\mathrm{CC}_{\mathcal{A}\cup\mathcal{T}} \not\leq \mathrm{CC}_{\mathcal{A}}$

Proof Assume, to obtain a contradiction, that $\mathrm{CC}_{\mathcal{A}\cup\mathcal{T}} \leq \mathrm{CC}_{\mathcal{A}}$ 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

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

Since $\mathcal{O} = \mathcal{R} \circ \mathcal{M}$, we have that there exist $s_1, s_2 \in \mathcal{M}[\![\mathcal{C}(A)]\!] \cap (Con_T^* \times \{ss\})$ 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

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

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)

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

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

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

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

$$\langle store(c_1.c_2.\psi_2), ff\rangle \in \mathcal{O}[\![\mathcal{C}(A) \parallel \mathcal{C}(A)]\!] = \text{ (by } \mathbf{P2}\text{) } \mathcal{O}[\![\mathcal{C}(A \parallel A)]\!]. \tag{4}$$

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

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

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

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

In this section we prove that $\mathrm{CC}_{\mathcal{A}} \not\leq \mathrm{CC}_{\emptyset}$. Again, the essence of the proof is based on a property ($\boldsymbol{\pi}$) satisfied by the semantics of $Proc_{\emptyset}$ and not by the one of $Proc_{\mathcal{A}}$. This property says that if a process can deadlock, then also the nondeterministic choice between this process and another one may deadlock. Intuitively, this holds in $Proc_{\emptyset}$ (and not in $Proc_{\mathcal{A}}$) 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 $Proc_{\emptyset}$ satisfies the following closure property:*

$\boldsymbol{\pi}$) *For each $G \in Proc_{\emptyset}$*

if $c.dd \in \mathcal{M}[\![G]\!]$ then for each $G' \in Proc_{\emptyset}$ we have $c.dd \in \mathcal{M}[\![G+G']\!]$.

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

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

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

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

holds, hence, by Proposition 6.1(2):

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

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

$$\begin{aligned}
c.dd &= \sigma_1^A. \dots .\sigma_k^A.true^T.c'.dd \\
&= \sigma_1^A. \dots .\sigma_k^A.(true^T.c'.dd \mathbin{\tilde{+}} \lambda.\perp) \\
&= \sigma_1^A. \dots .\sigma_k^A.true^T.c'.dd \mathbin{\tilde{+}} \sigma_1^A. \dots .\sigma_k^A.\perp \\
&\in \mathcal{M}[\![G]\!] \tilde{+} \mathcal{M}[\![G']\!] \\
&= \mathcal{M}[\![G{+}G']\!] \quad \text{(by Theorem 3.8).}
\end{aligned}$$

□

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

Proof Assume, to obtain a contradiction, that $\mathrm{CC}_{\mathcal{A}} \leq \mathrm{CC}_{\emptyset}$ 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 $A_1, A_2, A_3 \in Proc_{\mathcal{A}}$ be defined as follows:

$$A_1 = ask(\vartheta_1) \rightarrow tell(true)$$

$$A_2 = ask(\vartheta_2) \rightarrow tell(true)$$

$$A_3 = tell(\vartheta_3).$$

It is easy to see that

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

and

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

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

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

Since $\mathcal{O} = \mathcal{R} \circ \mathcal{M}$, for appropriate $c_1.\alpha_1 \in \mathcal{M}[\![\mathcal{C}(A_1)]\!]$ and $c_2.\alpha_2 \in \mathcal{M}[\![\mathcal{C}(A_3)]\!]$ we have

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

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

case 1) $\alpha_1 = dd$, or

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

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

7 Conclusions and future work

We have applied our notion of embedding to establish a hierarchy between various languages of the concurrent constraint paradigm.

If we compare this paper with the work of de Boer and Palamidessi (1991b) , it is quite surprising to see that the concurrent constraint hierarchy strictly corresponds to the CSP hierarchy. Since concurrent constraint is an asynchronous paradigm, we would rather expect a correspondence with the Asynchronous CSP family, where output guards do not increase the expressive power. In our opinion this difference arises because in concurrent constraint a choice guarded by tell primitives enforces consistency check, thus it depends upon the previous tell actions performed by the environment. This mutual dependency of actions cannot be expressed in Asynchronous CSP: input only depends on output, and output does not depend on any other action. In CSP, on the other hand, we have that input depend on output and output on input.

The characteristic features of our notion of embedding are compositionality of the compiler with respect to $\|$ and $+$ and termination invariance of the decoder. Actually, compositionality with respect to an operator $op \in Op$ would only require its translation into a combination of operators, namely

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

for every $A_1, \ldots, A_n$ in L'. (Here $c_{op}[\,]$ represents an n-ary context in L.) Our requirement **P2** is more restrictive since it requires translation into *one* operator, and can be justified as follows. Since the languages we study are extensions of each other, and the differences between them consist of the kind of 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, the question is 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 $c_g[\,]$. A similar formalization of language comparison is also studied by Felleisen (1990) . This amounts to require the existence of a translation that only transforms the guard operators and is invariant with respect to $+$ and $\|$. Such a translation can be seen as a particular case of a compiler that satisfies **P2**.

However, it would be interesting to study the consequences of adopting the more general notion of compositionality, as it seems natural to use basic constructs for implementing more complex ones. This would lead, of course, to a more liberal notion of embedding. Perhaps too liberal, since it seems not possible in this context to obtain the results of Bougè (1988). On the other hand, this might be compensated by the natural requirement of compositionality with respect to all the operators of the language.

The method of comparison proposed in this paper can be applied to many questions of interest in the field of concurrency. For example, the difference between many-to-many channels and one-to-one channels, the relation between concurrent constraint languages and other paradigms like CSP, etc.

Acknowledgements We thank Ehud Shapiro for stimulating discussions and encouragements. We acknowledge the department of Software Technology of CWI for providing a stimulating working environment, and to Krzysztof Apt for having suggested us relevant literature on the subject. Finally, we thank the anonymous referees for their helpful comments. The example of the program W in Section 2.1, and the condition on the variables of $W; A$, was suggested by one of them.

References

Baeten, J.C.M. , Bergstra, J.A., and Klop, J.W. (1987), On the consistency of Koomen's fair abstraction rule, *Theoretical Computer Science* **51** (1,2), 129–176.

Bergstra, J.A., and Klop, J.W. (1986), Process algebra: specification and verification in bisimulation semantics, *in* "Mathematics and Computer Science II", CWI Monographs, North-Holland, 61–94.

Bougé, L. (1987), On the existence of generic broadcast algorithms in networks of Communicating Sequential processes, in "Proceedings, Second International Workshop on Distributed Algorithms" Technical Report RUU-CS-87-10, Dept. of Computer science, University of Utrecht.

Bougé, L. (1988a), On the existence of symmetric algorithms to find leaders in networks of communicating sequential processes, *Acta Informatica* **25**, 179–201.

Bougé, L. (1988b), Symple nets: when symmetry meets simplicity, Technical Report, LIFO, Université d'Orléans, also in "Petri Nets Newsletters".

CHANDRA, A.K., AND MANNA, Z. (1975), The power of programming features, *J. Computer Languages* **1**, 219–232.

DE BOER, F.S., AND PALAMIDESSI, C. (1990), Concurrent logic languages: Asynchronism and language comparison, *in* "Proceedings, North American Conference on Logic Programming", (S. Debray and M. Hermenegildo, Eds.), Series in Logic Programming, The MIT Press, pp. 175–194.

DE BOER, F.S., AND PALAMIDESSI, C. (1991a), A fully abstract model for concurrent constraint programming, *in* "Proceedings, Theory and Practice of Software Development", (S. Abramsky and T.S.E. Maibaum, Eds.), Lecture Notes in Computer Science, Vol. 493, Springer-Verlag, pp. 296–319.

DE BOER, F.S., AND PALAMIDESSI, C. (1991b), Embedding as a tool for language comparison: On the CSP hierarchy, *in* "Proceedings, CONCUR 91", (J.C.M. Baeten and J.F. Groote, Eds.), Lecture Notes in Computer Science, Vol. 527, Springer-Verlag, pp. 127–141.

DE SIMONE, R. (1985), Higher-level synchronising devices in MEIJE-SCCS, *Theoretical Computer Science*, **37** (3), 245–267.

FELLEISEN, M. (1990), On the expressive power of programming languages, *in* "Proceedings, European Symposium on Programming", (N. Jones, Ed.), Lecture Notes in Computer Science, Vol. 432, Springer-Verlag, pp. 134 – 151.

GABBRIELLI, M., AND LEVI, G. (1990), Unfolding and fixpoint semantics for concurrent constraint logic programs, *in* "Proceedings, Second Int. Conf. on Algebraic and Logic Programming", (H. Kirchner and W. Wechler, Eds.), Lecture Notes in Computer Science, Vol. 463, Springer-Verlag, pp. 204–216.

GAIFMAN, H., MAHER, M.J., AND SHAPIRO, E. (1989), Reactive behaviour semantics for concurrent constraint logic programs. *in* "Proceedings, North American Conference on Logic Programming", (E. Lusk and R. Overbeck, Eds.), Series in Logic Programming, The MIT Press.

GERTH, R., CODISH, M., LICHTENSTEIN, Y., AND SHAPIRO, E. (1988), Fully abstract denotational semantics for Concurrent Prolog, *in* "Proceedings, Third IEEE Symposium on Logic in Computer Science" , IEEE Computer Society Press, New York, pp. 320–335.

HERLIHY, M.P. (1988), Impossibility and universality results for wait-free synchronization, *in* "Proceedings, Sixth Annual ACM Symposium on Principles of Distributed Computing", pp. 276–290.

LANDIN, P.J. (1966), The next 700 programming languages, *Communications of the ACM*, **3** (9), 157–166.

MAHER, M.J. (1987), Logic semantics for a class of committed-choice programs, *in* "Proceedings, Fourth International Conference on Logic Programming", (J.-L. Lassez, Ed.), Series in Logic Programming, The MIT Press, pp. 858–876.

MCALLESTER, M., PANANGADEN, P., AND SHANBHOGUE, V. (1988), Nonexpressibility of fairness and signaling, *in* "Proceedings, IEEE Foundations of Computer Science",

PANANGADEN, P., AND SHANBHOGUE, V. (1992), *Information and Computation*, **98** (1), 99–131.

PANANGADEN, P., AND STARK, E.W. (1988), Computations, residuals, and the power of indeterminacy, *in* "Proceedings, Fifteenth International Colloquium on Automata, Languages and Programming", (T. Lepistö and A. Salomaa, Eds.), Lecture Notes in Computer Science, Vol. 317, Springer-Verlag, pp. 439–454.

PARROW, J. (1989), The expressive power of parallelism, *in* "Proceedings, PARLE 89", Lecture Notes in Computer Science, Vol. 366, Springer-Verlag, pp. 389–405.

PATERSON, M.S., AND HEWITT, C.E. (1970), Comparative schematology, *in* "Proceedings, ACM Conference on Concurrent Systems and Parallel Computation", pp. 119–127.

REYNOLDS, J.C. (1970), GEDANKEN - a symple typeless language based on the principle of completeness and the reference concept, *Communications of the ACM*, **5** (13), 308–319.

REYNOLDS, J.C. (1981), The essence of Algol, *in* Algorithmic Languages, (J.W. de Bakker and van Vliet, Eds.), North-Holland, Amsterdam, pp. 345–372.

SARASWAT, V.A. (1989), *Concurrent Constraint Programming*, PhD thesis, Carnegie-Mellon University, January 1989, Published by The MIT Press, U.S.A., 1992.

SARASWAT, V.A., AND RINARD, M. (1990), Concurrent constraint programming, *in* "Proceedings, Seventeenth ACM Symposium on Principles of Programming Languages", New York, pp. 232–245.

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

Shapiro, E.Y. (1991), Separating concurrent languages with categories of language embeddings, *in* "Proceedings, 23^{rd} Annual ACM Symposium on Theory of Computing", pp. 198–208.

Shapiro, E.Y. (1992), Embeddings among concurrent programming languages, *in* "Proceedings, CONCUR 92", (W.R. Cleaveland, Ed.), Lecture Notes in Computer Science, Vol. 630, Springer-Verlag, pp. 486–503.

Stark, E.W. (1990), On the relations computed by a class of concurrent automata, *in* "Proceedings, Seventeenth ACM Symposium on Principles of Programming Languages", New York, pp. 329–340.

Steele, G.L.Jr., and Sussman, G.J. (1976), Lambda: The ultimate imperative, Technical Report Memo 353, MIT AI Lab.

Ueda, K. (1987), Guarded Horn Clauses, *in* "Concurrent Prolog: Collected Papers", (E. Y. Shapiro, Ed.), Series in Logic Programming, The MIT Press.

Vaandrager, F. (1992), Expressiveness results for process algebras, *in* "Proceedings, REX Workshop on 'Semantics: Foundations and Applications' " Lecture Notes in Computer Science, Springer-Verlag.