Embedding as a tool for Language Comparison: On the CSP hierarchy

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]Centre for Mathematics and Computer Science,
P.O. Box 4079, 1009 AB Amsterdam, The Netherlands
email: katuscia@cwi.nl
and
Department of Computer Science, Utrecht University
P.O. Box 80089 3508 TB Utrecht, The Netherlands

Abstract

The concept of embedding has recently been introduced as a formal tool to study the relative expressive power of (concurrent) programming languages. We use the notion of "modular embedding" to compare various dialects of CSP and ACSP (Asynchronous CSP), which differ on the kind of communication primitives allowed in the guards: all, only input, or none. Concerning the synchronous paradigm, we show that CSP is strictly more powerful than $CSP_{\mathcal{I}}$ (the version of CSP with no output guards), and that $CSP_{\mathcal{I}}$ is strictly more powerful than $CSP_{\emptyset}$ (the version of CSP with no communication primitives in the guards). The first separation result does not hold in the asynchronous variants of these languages: since asynchronous output guards cannot be influenced by the environment (they can always proceed), it is irrelevant to have or not to have them in the language. Therefore, ACSP and $ACSP_{\mathcal{I}}$ are equivalent. Still, they are strictly more expressive than $ACSP_{\emptyset}$. Finally, we come to compare the synchronous and asynchronous paradigms. The asynchronous communication can be modeled synchronously by means of "buffer" processes. On the other hand, synchronous communication (when not fully used to control nondeterminism) can be modeled asynchronously by means of acknowledgement messages. As a consequence, $CSP_{\mathcal{I}}$, ACSP, and $ACSP_{\mathcal{I}}$ are equivalent. An interesting corollary of these results is that ACSP is strictly less powerful than CSP.

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 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

From a mathematical point of view, all "reasonable" programming languages are equivalent, since all of them can compute the same class of functions. Yet, it is common practice to speak about the "power" of a language on the basis of the expressibility or non expressibility of programming constructs. 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" [5, 10, 14, 16, 17, 18, 22]. 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" [10], i.e., compositionally. Of course, the translation must preserve

the meaning, at least in the weak sense of preserving termination [10]. In case of nondeterministic sequential languages, termination is usually dealt with existentially (i.e. failing or infinite computations are ignored, if there is at least a successfully terminating one). In this sense the nondeterministic Turing machines are equivalent to the deterministic ones.

When we move to the field of concurrent languages, the notion of termination must be reconsidered. This is because the nondeterminism plays an essentially different role: each possible computation represents a possible different evolution of a system of interacting processes, and must be considered independently from the other ones (backtracking is usually not possible, or to difficult to implement). Also, there is an additional termination case: *deadlock*, representing the situation in which each process is stuck waiting for some condition to be established by the others. Essentially due to this "richer" concept of termination, the models (and the notion of meaning itself) for concurrent languages are usually far more complicated. From the point of view of language comparison, this richer notion of meaning gives additional "freedom" for the definition of the notion of expressivity [20, 7, 8, 6, 21, 9, 1, 2, 15, 23].

In [23] the expressive power of a concurrent languages is investigated under various criteria, 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 (as contexts of the language).

While all "reasonable" languages are universal up to the first criterion, this is not the case for the other two. In particular, [23] shows that there cannot be a concurrent language (with an effective operational semantics) able to express all effective process graphs, up to trace equivalence. This generalizes a result of [1], proving the same but for (strong) bisimulation equivalence. Positive results for the second criterion include [1], showing the universality of ACP_τ (up to weak bisimulation), and [9], showing the universality of MEIJE-SCCS with unguarded recursion (up to strong bisimulation). This last result does not contradict the results of [1, 23], since unguarded recursion induces an infinitely branching (hence not effective) operational semantics. Even more surprisingly, [9] shows that MEIJE-SCCS are universal with respect to all the operators definable via transition rule of a certain, very general, format. In [15] an interesting result is shown for the class of finite process graphs: all of them can be expressed (up to every equivalence between traces and bisimulation, under τ abstraction) as parallel composition of a finite number of instances of three elementary processes (a three-ways synchronizer, an arbiter, and an alternator). As a consequence, it is shown that all finite-state asynchronous operators (including for instance the CSP external nondeterministic choice, parallel, and hiding) can be expressed as contexts in that simple language. With respect to the second criterion, [2] provides a detailed classification of various ACP sublanguages, showing how each operator contributes to increase the expressive power.

In this paper the notion of expressivity is considered only relatively, i.e. in the comparison of one language with respect to another. The method of comparison we adopt is, in part, based on the same idea of the third criterion illustrated above, in the sense that it requires (some) of the operators of a language to be expressible (as operators) in the other language. One difference is that we do not require a statement in a language and its correspondent in the other language to have the same semantics, even not up to isomorphism. We only require an injection (abstraction) from the second to the first. To our opinion, this second (more liberal) requirement better capture the intuition of expressing a language into another, since it allows to abstract from "details" that may be necessary for the implementation (like additional information about control).

We consider the notion of *modular embedding*, introduced in [7] as a method to compare the expressive power of concurrent logic languages. This notion has been refined in [8] and shown to be general enough to establish a hierarchy on a broader class of languages: the Concurrent Constraint family [19]. In this paper, we apply the method to compare various dialects of the CSP paradigm [4, 12]. We show that CSP is strictly more powerful than $CSP_{\mathcal{I}}$ (the sublanguage with no *output guards*), and that $CSP_{\mathcal{I}}$ is strictly more powerful than $CSP_{\emptyset}$ (the sublanguage with no *communication primitives* in the guards). This presents a strict analogy with the concurrent constraint case: the choices guarded by *tell* operations cannot be simulated by choices guarded by *ask* operations, and the latter cannot be simulated by the unguarded choice [8].

If we now consider the asynchronous variant of the CSP family, ACSP [12, 13], we may expect that these separation results are reflected there, but this is not the case. As we show in this paper, $\text{ACSP}_\emptyset$ is still less powerful, but ACSP and $\text{ACSP}_\mathcal{I}$ are equivalent. This is due to the asymmetric behavior of input and output guards in ACSP: an output can always proceed, independently from the actions performed by the other processes in the environment. Therefore output guards cannot control nondeterminism. They don't really increase the expressive power of the language. This is already suggested in [13], in fact output guards are not even considered there.

This asymmetry already suggests that shifting to asynchrony causes a loss of power, and, in fact, we show that ACSP is equivalent to $\text{CSP}_\mathcal{I}$, so proving that CSP is strictly more powerful than ACSP.

1.1 The method

We summarize here the method for language comparison, called modular embedding, as defined in [8].

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 behavior. This notion has recently become popular with the name of *embedding* [20, 7, 8, 21]. The basic definition of embedding, given by Shapiro [20], is the following. Consider two languages, L and L'. Assume given the semantic functions (*observation criteria*) $\mathcal{S} : L \to Obs$ and $\mathcal{S}' : L' \to Obs'$, where Obs, Obs' are some suitable domains. Then L *can embed* L' if there exists a mapping $\mathcal{C}$ (*compiler*) from the statements of L' to the statements of L, and a mapping $\mathcal{D}$ (*decoder*) from Obs to Obs' such that for every statement $A \in L'$, the following equation holds

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

In other words, the diagram of Figure 1 commutes.

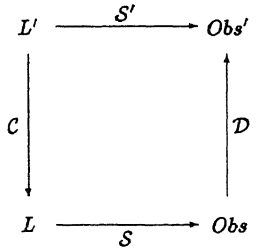

Figure 1: basic embedding.

This notion however is too weak (as Shapiro himself remarked) since the above equation is satisfied by any Turing-complete language. Actually, L does not even need to be Turing-complete, it is sufficient that it contains infinitely many observably different programs. We can then take a $\mathcal{C}$ such that $\mathcal{S} \circ \mathcal{C}$ does not identify more programs than $\mathcal{S}'$, and define $\mathcal{D}$ as the function such that the diagram of Figure 1 commutes. In [21] it is shown how finite automata can "embed" (in this weak sense) Turing machines.

In order to use the notion of embedding as a tool for comparison of concurrent languages we have therefore to add some restrictions on $\mathcal{C}$ and $\mathcal{D}$. 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 plays an important role, the domain of the observables (Obs) is in general a powerset (i.e. the elements S of Obs are sets). In fact, each element must represent the outcomes of all possible computations. 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 $\forall S \in Obs.\, \mathcal{D}(S) = \{\mathcal{D}_{el}(s) : s \in S\}$

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 $\mathcal{C}$ so that, for each statement A, each element of $\mathcal{S}[\![\mathcal{C}(A)]\!]$ encodes A, and then define an appropriate decoder. See [8, §4] for a concrete example of such an embedding.

Another observation is the following. When compiling a concurrent process, it might not be feasible 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.

Additionally, it is useful to compile a process in a compositional way with respect to the possible nondeterministic choices, so to have the possibility to add alternatives (for instance, communication offers) without the need of recompilation. These properties can be formulated as follows:

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 behavior of the original process with respect to deadlock (and/or failure) and success. Intuitively, a system which is not deadlock-free cannot be considered equivalent to a system which is. 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 $\mathcal{S}[\![\mathcal{C}(A)]\!]$ must correspond to a deadlock [failure] in $\mathcal{S}'[\![A]\!]$, and a success must correspond to a success. Formally

P3 $\forall S \in Obs. \forall s \in S.\, tm'(\mathcal{D}_{el}(s)) = tm(s)$

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** discussed above. (In the following we will omit the term "modular" when it is clear from the context.)

The existence of a modular embedding from L' into L will be denoted by $L' \leq L$. 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 functions, and the transitivity is shown by the commutative diagram in Figure 2.

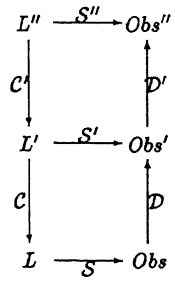

Figure 2: transitivity of $\leq$.

Note that, if $L' \subseteq L$, then $L' \leq L$. In fact, it is sufficient to define $\mathcal{C}$ and $\mathcal{D}$ as (the extension of) the identity functions.

1.2 Related works

Bougé [3] has presented similar separation results for three CSP-dialects which closely correspond to three of the languages we study: CSP, $\text{CSP}_{\mathcal{I}}$, and $\text{CSP}_{\emptyset}$. (The main difference is that he considers the original definition of CSP, as given in [11].)

His method is based on showing that, given some communication graph, 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. These results can be interpreted as stating the non-existence of a translation, when the compiler is required to preserve parallelism and the topology of the network.

The notion of embedding was first proposed by Shapiro as a method for language comparison in [20], and refined in various vays in [7, 8, 21, 6] by adding conditions so to make it non-trivial. As already explained, we follow here the approach of [7, 8], where the notion of modular embedding was first defined and applied to show separation results in the Concurrent Constraint family. Based upon the same notion we establish in this paper a hierarchy of the CSP languages similar to the one of the Concurrent Constraint family. This is quite surprising, because the Concurrent Constraint languages are based on asynchronous communication, so one would rather expect a similarity with the ACSP hierarchy.

In [21] three different notions of embedding are investigated. The first one (sound embedding) is equivalent to the original definition in [20] plus the requirement of compositionality with respect to the parallel operator. (Therefore it is equivalent to our notion, dropping the conditions on the decoder and the compositionality w.r.t. the choice operator. Notice that our notion of embedding thus leads to a subcategory of the category of sound embeddings.) The second one (faithful embedding) requires additionally the compiler to translate (observable) equivalent statements into equivalent ones. The third notion considered (fully abstract embedding) requires the compiler to preserve the (non-)equivalence of statements with respect to the fully abstract semantics. These two last notions are not comparable with our approach.

In [6] the notion of modular embedding is used to compare the expressive power of a general asynchronous paradigm $\mathcal{L}$ with respect to a synchronous paradigm (CCS). It is shown that CCS can be embedded into $\mathcal{L}$, by choosing an appropriate interpretation for the basic actions of $\mathcal{L}$ (CCS and $\mathcal{L}$ are *uniform* languages, i.e. the basic actions are uninterpreted). This is not in contrast with the non-embeddability of CSP into ACSP, because ACSP is only a particular instance of $\mathcal{L}$. In order to point out the differencies between synchronous and asynchronous forms of communication, it is investigated under which further conditions CCS cannot be embedded into $\mathcal{L}$.

1.3 Plan of the paper

This paper is organized as follows. Next section introduces the family $\mathrm{CSP}_\mathcal{G}$ as a class of CSP-like languages parametrized on $\mathcal{G}$, the set of communication primitives that can occur in the guards. The behavior of $\mathrm{CSP}_\mathcal{G}$ processes is specified via a transition system, from which we derive the notion of observables and a compositional semantics. In Section 3 we study the hierarchy on the members of this class: by using the compositional semantics we show that CSP cannot be embedded in $\mathrm{CSP}_\mathcal{I}$, and that $\mathrm{CSP}_\mathcal{I}$ cannot be embedded in $\mathrm{CSP}_\emptyset$. In Section 4 we introduce the asynchronous variant of this family, $\mathrm{ACSP}_\mathcal{G}$, and its compositional model. In Section 5 we show that ACSP can be embedded in $\mathrm{ACSP}_\mathcal{I}$, but not in $\mathrm{ACSP}_\emptyset$. Finally, the synchronous and the asynchronous CSP families are compared, and we discuss the scope of the results. Due to space limitations, some of the proofs are omitted.

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

In this section we present the $\mathrm{CSP}_\mathcal{G}$ family. The members of this class are simplified versions of CSP dialects [4, 12]. We abstract from some of the traditional CSP constructs, like renaming. Also, we don't consider recursion (we deal only with finite processes), but the results we present can be generalized to the full version of the languages.

2.1 The syntax

Let $(c, d, \ldots \in)$ *Chan* be a set of *channel names*. Let $(x, y, \ldots \in)$ *Var* be a set of *variables*, $(t, u, \ldots \in)$ *Term* an abstract set of terms (expressions) on these variables, and $(v, w, \ldots \in)$ *Val* an abstract set of values. The set of basic actions is given by $\mathcal{I} \cup \mathcal{O} \cup \mathcal{A}$, where

$$(i \in) \mathcal{I} = \{c?x : c \in \mathit{Chan}, x \in \mathit{Var}\}$$

are the *input* actions,

$(o \in)\ \mathcal{O} = \{c!t : c \in \mathit{Chan}, t \in \mathit{Term}\}$

are the *output* actions, and $(a \in)\ \mathcal{A}$ are internal actions. Note that we abstract from the structure of internal actions. We only assume the existence of an internal move, *Skip*, which does not have any observable effect.

The set $(g \in)\ \mathcal{G}$ specifies the *communication primitives* used in the guards. We restrict to the following cases: $\mathcal{G} = \emptyset$, $\mathcal{G} = \mathcal{I}$, or $\mathcal{G} = \mathcal{I} \cup \mathcal{O}$. The set of statements in $\mathrm{CSP}_{\mathcal{G}}$, with typical elements $A, B, \ldots$, is described by the following grammar

$$A ::= \mathit{Stop} \mid c!t; A \mid c?x; A \mid a; A \mid A \parallel A \mid G$$

$$G ::= g \rightarrow A \mid a \rightarrow A \mid G + G$$

The symbols ; and $\parallel$ represent the sequential and the parallel composition respectively. A process $g \rightarrow A$ first executes g (if possible) and then it behaves like A. The *guard* g belongs to $\mathcal{G}$ and its execution depends upon the actions performed by the external environment. The language presents two forms of nondeterministic choice $+$: the guarded (external) nondeterminism, like for instance in $(g_1 \rightarrow A_1) + (g_2 \rightarrow A_2)$, where only a branch whose guard is enabled can be selected, and the unguarded (internal) nondeterminism, like for instance in $(a_1 \rightarrow A_1) + (a_2 \rightarrow A_2)$, where a branch will be selected independently from the external environment.

Note that we don't impose explicitly one-to-one communication. All the results we present (apart from $\mathrm{CSP}_{\mathcal{I}} \leq \mathrm{ACSP}_{\mathcal{I}}$ and $\mathrm{CSP}_{\emptyset} \leq \mathrm{ACSP}_{\emptyset}$) do not depend on this assumption.

2.2 The operational model of $\mathrm{CSP}_{\mathcal{G}}$

The operational model of $\mathrm{CSP}_{\mathcal{G}}$ is uniformly described by a labeled transition system

$$T = (\mathit{Conf}, \bigcup_{e \in E} \xrightarrow{e})$$

where $(e \in)\ E$ denotes the set of events of the form $c?v, c!v$ (*external* or *visible* steps), with $v \in \mathit{Val}$, and τ (*internal* or *invisible* step),. The configurations *Conf* are pairs consisting of a statement A and a state σ $(\in \mathit{State})$, namely a mapping from *Var* to *Val*. The notion of state is introduced here just to assign a meaning to internal actions, it cannot be used to communicate (there is no shared memory). In order to simplify the description of the transition system, we assume that the components of a set of parallel processes do not share variables with the same name. This syntactical restriction on the language allows us to represent the collection of all local states by just one "global" state. The meaning of internal actions we assume to be given by an *interpretation function* I. Namely, $I(a)(\sigma)$ is the state resulting by executing a in σ. (Actually, the execution of a only depends upon and affects the component of σ local to the process executing a.) The interpretation of *Skip* is the identity, i.e. $I(\mathit{Skip})(\sigma) = \sigma$. Given a state σ and a term t, the value of t is denoted by $\sigma(t)$. The state resulting from σ by assigning v to x is denoted by $\sigma\{v/x\}$.

The transition relations are described in Table 1. The first rule describes the meaning of an internal action as a state transformation. The statement *Stop* denotes termination. An input action $c?y$ is described by means of a transition labeled by an event $c?v$, which records the fact that the value v has been received. Similarly, an output action $c!t$ is described by means of a transition labeled by an event $c!v$, where v is the value of the term t. The other rules are the usual rules for compound statements (note that parallelism is described as interleaving).

The result of a computation consists of the final state together with the termination mode, *ss* or *dd*, indicating successful termination or deadlock, respectively. This is formally represented by the notion of *observables*.

Definition 2.1 *The observables of the class* $\mathrm{CSP}_{\mathcal{G}}$ *are given by the function* $\mathcal{S}_{\mathcal{G}} : \mathrm{CSP}_{\mathcal{G}} \rightarrow \mathit{Obs}$ *with* $\mathit{Obs} = \mathit{State} \rightarrow (\mathit{State} \times \{ss, dd\})$.

$$\mathcal{S}_{\mathcal{G}}[\![A]\!](\sigma) = \{\langle \sigma', ss\rangle : \langle A, \sigma\rangle \xrightarrow{\tau}{}^{*} \langle \mathit{Stop}, \sigma'\rangle\} \cup \{\langle \sigma', dd\rangle : \langle A, \sigma\rangle \xrightarrow{\tau}{}^{*} \langle B, \sigma'\rangle \not\xrightarrow{\tau} \wedge\ B \neq \mathit{Stop}\}$$

Table 1: The Transition System T

R0	$\langle a; A, \sigma\rangle \xrightarrow{\tau} \langle A, I(a)(\sigma)\rangle$	
R1	$\langle c?x; A, \sigma\rangle \xrightarrow{c?v} \langle A, \sigma\{v/x\}\rangle$	where $v \in Val$
R2	$\langle c!t; A, \sigma\rangle \xrightarrow{c!v} \langle A, \sigma\rangle$	where $\sigma(t) = v$
R3	$\dfrac{\langle g; Stop, \sigma\rangle \xrightarrow{e} \langle Stop, \sigma'\rangle}{\langle g \to A, \sigma\rangle \xrightarrow{e} \langle A, \sigma'\rangle}$	where $g \in \mathcal{G} \cup \mathcal{A}$
R4	$\dfrac{\langle A, \sigma\rangle \xrightarrow{e} \langle A', \sigma'\rangle \mid \langle Stop, \sigma'\rangle}{\begin{array}{l}\langle A \parallel B, \sigma\rangle \xrightarrow{e} \langle A' \parallel B, \sigma'\rangle \mid \langle B, \sigma'\rangle \\ \langle B \parallel A, \sigma\rangle \xrightarrow{e} \langle B \parallel A', \sigma'\rangle \mid \langle B, \sigma'\rangle \\ \langle A+B, \sigma\rangle \xrightarrow{e} \langle A', \sigma'\rangle \mid \langle Stop, \sigma'\rangle \\ \langle B+A, \sigma\rangle \xrightarrow{e} \langle A', \sigma'\rangle \mid \langle Stop, \sigma'\rangle\end{array}}$	
R5	$\dfrac{\langle A, \sigma\rangle \xrightarrow{c?v} \langle A', \sigma'\rangle \quad \langle B, \sigma\rangle \xrightarrow{c!v} \langle B', \sigma\rangle}{\langle A \parallel B, \sigma\rangle \xrightarrow{\tau} \langle A' \parallel B', \sigma'\rangle}$	

where $\xrightarrow{\tau}{}^{*}$ denotes the reflexive and transitive closure of $\xrightarrow{\tau}$.

Note that the observables are defined in a uniform way for all the members of the class, and on the same domain *Obs*. In the following, we will omit the subscript $\mathcal{G}$ when it is clear from the context which language we are referring to.

On the basis of the transition system T we now define a compositional semantics $\mathcal{M}_{\mathcal{G}}$ for $\mathrm{CSP}_{\mathcal{G}}$. The semantical domain of $\mathcal{M}_{\mathcal{G}}$, denoted by $(p, q \in) P$, is the same for all $\mathcal{G}$'s and it is the set of synchronization trees with finite depth whose arcs are labeled on $State \times E \times State$. More formally, P is the smallest set that satisfies the following conditions:

- $\emptyset \in P$
- If $\phi_1, \ldots, \phi_n, \ldots \in State \times E \times State$, and $p_1, \ldots, p_n, \ldots \in P$ then $\{\phi_1 \cdot p_1, \ldots, \phi_n \cdot p_n, \ldots\} \in P$

Definition 2.2 *The semantics $\mathcal{M}_{\mathcal{G}} : \mathrm{CSP}_{\mathcal{G}} \to P$ is defined as*

$$\mathcal{M}_{\mathcal{G}}[\![A]\!] = \begin{cases} \emptyset & \text{if } A = Stop \\ \{\langle \sigma, e, \sigma'\rangle \cdot \mathcal{M}_{\mathcal{G}}[\![B]\!] : \langle A, \sigma\rangle \xrightarrow{e} \langle B, \sigma'\rangle\} & \text{otherwise} \end{cases}$$

Also for $\mathcal{M}_{\mathcal{G}}$ we have a uniform definition, and we will omit the subscript $\mathcal{G}$.

The correctness $\mathcal{M}$ is shown by defining the following abstraction operator $\mathcal{R}$, that extracts the observables of a process from its denotation in $\mathcal{M}$. This is done by collecting the final state plus termination mode of all those maximal paths labeled by τ steps and consecutive states. Such a path is maximal if either it ends in a leaf, in which case it represents a successful terminating execution, or it cannot be extended by a τ step, thus representing a deadlocking execution.

Definition 2.3

The operator $\mathcal{R} : P \to Obs$ is given by

$$\mathcal{R}(p)(\sigma) = \begin{cases} \{\langle \sigma, ss\rangle\} & \text{if } p = \emptyset \\ \{\langle \sigma, dd\rangle\} & \text{if } p \neq \emptyset \text{ and } \forall \sigma', q. \langle \sigma, \tau, \sigma'\rangle \cdot q \notin p \\ \bigcup\{\mathcal{R}(q)(\sigma') : \langle \sigma, \tau, \sigma'\rangle \cdot q \in p\} & \text{otherwise} \end{cases}$$

Theorem 2.4 (Correctness of $\mathcal{M}$) *The observables $\mathcal{S}$ can be obtained by $\mathcal{R}$-abstraction from $\mathcal{M}$, namely $\mathcal{S} = \mathcal{R} \circ \mathcal{M}$.*

The compositionality of $\mathcal{M}$ is shown by defining semantic operators corresponding to the operators of the language. The choice operator is just defined as union and the sequential operator as tree concatenation. The parallel operator $\tilde{\|}$ is defined as usual in terms of the left merge and the synchronization merge, where the synchronization merge transforms the arcs $\langle \sigma, c!v, \sigma \rangle$ and $\langle \sigma, c?v, \sigma' \rangle$ into $\langle \sigma, \tau, \sigma' \rangle$.

3 The CSP hierarchy

In this section we study the relations between the members of the CSP family. We consider $\mathrm{CSP}_\emptyset$, $\mathrm{CSP}_\mathcal{I}$ and $\mathrm{CSP}_{\mathcal{I} \cup \mathcal{O}}$. For the sake of convenience, the latter will be denoted by CSP.

Since the languages $\mathrm{CSP}_\emptyset$, $\mathrm{CSP}_\mathcal{I}$ and CSP are sublanguages of each other, we have the following proposition.

Proposition 3.1 $\mathrm{CSP}_\emptyset$ *can be embedded in* $\mathrm{CSP}_\mathcal{I}$ *and* $\mathrm{CSP}_\mathcal{I}$ *can be embedded in* CSP, *namely:* $\mathrm{CSP}_\emptyset \leq \mathrm{CSP}_\mathcal{I} \leq \mathrm{CSP}$.

The rest of the section will be devoted to show that this ordering is strict, namely $\mathrm{CSP} \not\leq \mathrm{CSP}_\mathcal{I} \not\leq \mathrm{CSP}_\emptyset$.

We will use the following properties on the compositional semantics and on the abstraction operator. They can easily be shown by definition of $\mathcal{R}$ and $\tilde{\|}$.

Proposition 3.2

1. $\forall p.\ \mathcal{R}(p)(\sigma) \neq \emptyset$
2. $\forall \sigma, \sigma', p, q.$ *if* $\langle \sigma, \tau, \sigma' \rangle \cdot p \in q$ *then* $\mathcal{R}(\langle \sigma, \tau, \sigma' \rangle \cdot p)(\sigma) \subseteq \mathcal{R}(q)(\sigma)$
3. $\forall p, q, q'.\ p \tilde{\|} (q \cup q') = p \tilde{\|} q \cup p \tilde{\|} q'$

The first property says that the abstraction operator $\mathcal{R}$ always delivers a non-empty set of results for any synchronization tree. The second property states that a subtree starting with a τ branch will deliver a subset of the results delivered by the complete tree. Note that this is not the case for subtrees starting with a communication action: they give a deadlock, whereas the full tree may offer other alternatives, so avoiding deadlock. Finally, the last property states the distributivity of $\tilde{\|}$ with respect to the union.

We first show that CSP cannot be embedded into $\mathrm{CSP}_\mathcal{I}$.

Theorem 3.3 $\mathrm{CSP} \not\leq \mathrm{CSP}_\mathcal{I}$.

Proof Assume, by contradiction, that $\mathrm{CSP} \leq \mathrm{CSP}_\mathcal{I}$ via $\mathcal{C}$ and $\mathcal{D}$. Let

$$A_1 = c_1!v_1 \rightarrow \mathit{Stop}, \quad A_2 = c_2?x_2 \rightarrow \mathit{Stop},$$

and

$$B_1 = c_1?x_1 \rightarrow \mathit{Stop}, \quad B_2 = c_2!v_2 \rightarrow \mathit{Stop}.$$

with $c_1 \neq c_2$. Let $A = A_1 + A_2$, and $B = B_1 + B_2$, and consider $A \parallel B$.

We have that

$$\mathcal{S}[\![A \parallel B]\!](\epsilon) = \{(\{v_1/x_1\}, ss), (\{v_2/x_2\}, ss)\} \tag{1}$$

where ϵ denotes the empty (initial) state. Because of **P2** and the compositionality of $\mathcal{M}$, we have

$$\mathcal{M}[\![\mathcal{C}(A \parallel B)]\!] = (\mathcal{M}[\![\mathcal{C}(A_1)]\!] \cup \mathcal{M}[\![\mathcal{C}(A_2)]\!]) \tilde{\|} (\mathcal{M}[\![\mathcal{C}(B_1)]\!] \cup \mathcal{M}[\![\mathcal{C}(B_2)]\!]) \tag{2}$$

Because of **P3** and the fact that in $\mathrm{CSP}_\mathcal{I}$ the semantics of a choice cannot contain an initial node labeled by an output event (a choice cannot be guarded by an output event) we then have

$$\exists \sigma, p.\ \langle \epsilon, \tau, \sigma \rangle \cdot p \in \mathcal{M}[\![\mathcal{C}(A_1)]\!] \cup \mathcal{M}[\![\mathcal{C}(A_2)]\!] \cup \mathcal{M}[\![\mathcal{C}(B_1)]\!] \cup \mathcal{M}[\![\mathcal{C}(B_2)]\!]$$

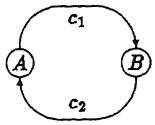

Figure 3: $A \parallel B$

We consider the following case (the other ones being analogous):

$$\langle \epsilon, \tau, \sigma \rangle \cdot p \in \mathcal{M}[\![\mathcal{C}(A_1)]\!]. \tag{3}$$

We show that there exists an element $\langle \vartheta, dd \rangle \in \mathcal{S}[\![\mathcal{C}(A \parallel B)]\!](\epsilon)$, that, together with (1), contradicts **P3**.
By Proposition 3.2(1),

$$\exists \langle \vartheta, \alpha \rangle \in \mathcal{R}(\langle \epsilon, \tau, \sigma \rangle \cdot p \,\tilde{\|}\, \mathcal{M}[\![\mathcal{C}(B_2)]\!])(\epsilon)$$

where $\langle \epsilon, \tau, \sigma \rangle \cdot p$ is the element considered in (3). By Proposition 3.2(2),

$$\mathcal{R}(\langle \epsilon, \tau, \sigma \rangle \cdot p \,\tilde{\|}\, \mathcal{M}[\![\mathcal{C}(B_2)]\!])(\epsilon) \subseteq \mathcal{R}(\mathcal{M}[\![\mathcal{C}(A_1)]\!] \,\tilde{\|}\, \mathcal{M}[\![\mathcal{C}(B_2)]\!])(\epsilon)$$

Since $A_1 \parallel B_2$ always deadlock, we have by **P1**, **P2** and **P3** that $\alpha = dd$. Therefore

$$\begin{aligned}
\langle \vartheta, dd \rangle &\in \mathcal{R}(\langle \epsilon, \tau, \sigma \rangle \cdot p \,\tilde{\|}\, \mathcal{M}[\![B_2]\!])(\epsilon) \\
&\subseteq \mathcal{R}(\mathcal{M}[\![\mathcal{C}(A \parallel B)]\!])(\epsilon) && \text{(by Proposition 3.2 (2 and 3) and by (2))} \\
&= \mathcal{S}[\![\mathcal{C}(A \parallel B)]\!](\epsilon) && \text{(by Theorem 2.4 (correctness))}
\end{aligned}$$

□

In a similar way we can prove that $\mathrm{CSP}_{\mathcal{I}}$ cannot be embedded in $\mathrm{CSP}_{\emptyset}$:

Theorem 3.4 $\mathrm{CSP}_{\mathcal{I}} \not\lesssim \mathrm{CSP}_{\emptyset}$.

4 The class $\mathrm{ACSP}_{\mathcal{G}}$

In this section we present the $\mathrm{ACSP}_{\mathcal{G}}$ family. The members of this class are simplified versions of ACSP dialects [13] with the difference that we consider, for the sake of uniformity with the synchronous case, also output guards (not present in [13]).

The syntax of the languages of the $\mathrm{ACSP}_{\mathcal{G}}$ family is defined in the same way as for $\mathrm{CSP}_{\mathcal{G}}$ but for the communication actions: outputting a value v along a channel c is now indicated by $c!!v$, and receiving a value by $c??x$.

$$\begin{array}{|l|}
\hline
A ::= \mathit{Stop} \mid c!!t; A \mid c??x; A \mid a; A \mid A \parallel A \mid G \\
\\
G ::= g \rightarrow A \mid a \rightarrow A \mid G + G \\
\hline
\end{array}$$

Semantically, we do not require synchronization on complementary events: outputting a value v along a channel c does not need to take place at the same time as a corresponding input action $c??x$. The only restriction is that an input action on a channel c can take place only if at least one output action on c has been performed (by another process), and the corresponding value has not yet been "consumed" by another input action on c. In other words, a channel is seen as a buffer. Outputting a value corresponds to adding it to the buffer, and inputting a value corresponds to reading and retrieving a value present in the buffer. There are various design options concerning the buffers. First of all, a buffer can be *ordered* or *unordered*. In the first case there are various strategies, FIFO, LIFO etc. The most "natural" is the FIFO: an input

reads and consumes the first value sent and not yet consumed. This is the solution adopted in standard ACSP [12, 13]. For the sake of simplicity, we consider here unordered buffers (bags). However, the results we present can be adapted to the ordered buffers (the only non-obvious case is $\mathrm{ACSP}_{\mathcal{I}} \leq \mathrm{CSP}_{\mathcal{I}}$, for which we have to require explicitly that communication is one-to-one).

The state of a channel modeled as a bag is a multiset of values. An output action adds the corresponding value, whereas an input action reads and retrieves a value nondeterministically selected . An input action suspends in case the channel is empty.

The operational model of $\mathrm{ACSP}_{\mathcal{G}}$ is based upon a transition system T' that is defined in the same way as T (the transition system for $\mathrm{CSP}_{\mathcal{G}}$) but for the last rule (synchronous communication) which is omitted. Formally:

$$T' = (Conf, \bigcup_{e \in E} \xrightarrow{e})$$

where *Conf* is defined as in T and the transition relations $\xrightarrow{e}$ are defined by the rules **R0-R4** of Table 1. (Replacing, of course, $c?$ by $c??$ and $c!$ by $c!!$.)

The semantics $\mathcal{M}$ defined for $\mathrm{CSP}_{\mathcal{G}}$ can be adopted also for $\mathrm{ACSP}_{\mathcal{G}}$ because T' is just a sub-transition system of T. (Of course, $c?$ and $c!$ have to be replaced by $c??$ and $c!!$. The new model will be denoted by $\mathcal{M}'$.) The compositionality of $\mathcal{M}'$ is shown in a similar way, the only difference being the definition of the parallel operator, which is defined only in terms of the leftmerge, i.e., we do not have the synchronization merge. To extract the observables from the denotation of a program we have to modify the definition of the abstraction operator $\mathcal{R}$. For this purpose we add a parameter which assigns to each channel a multiset of values. The execution of an output action then will consists in adding the value sent to the channel, and the execution of an input action in retrieving a value from the channel. The set of assignments of multisets to channels is denoted by $(b \in)B = Chan \to \mathcal{P}_m(Val)$, where $\mathcal{P}_m(Val)$ denotes the set of multisets of values. Given a multiset m and a value v the multisets $m \setminus v$ and $m \cup v$ denote the result of deleting a copy of v from m, and adding a copy of v, respectively.

Definition 4.1

The operator $\mathcal{R}' : P \to B \to State \to (State \times \{ss, dd\})$ *is given by*

$$\begin{aligned}
\mathcal{R}'(p)(b)(\sigma) = \; & \{\mathcal{R}'(q)(b)(\sigma') : \langle \sigma, \tau, \sigma' \rangle \cdot q \in p\} \\
& \cup \\
& \{\mathcal{R}'(q)(b\{b(c) \cup v/c\})(\sigma') : \langle \sigma, c!!v, \sigma' \rangle \cdot q \rangle \in p\} \\
& \cup \\
& \{\mathcal{R}'(q)(b\{b(c) \setminus v)/c\})(\sigma') : \langle \sigma, c??v, \sigma' \rangle \cdot q \in p \wedge v \in b(c)\} \\
& \cup \\
& \{\langle \sigma, dd \rangle : \{\langle \sigma, e, \sigma' \rangle : \langle \sigma, e, \sigma' \rangle \cdot q \in p \textit{ and if } e = c??v \textit{ then } v \in b(c)\} = \emptyset\}
\end{aligned}$$

Here p is assumed to be not empty, for $p = \emptyset$ we define $\mathcal{R}'(\emptyset)(b)(\sigma) = \{\langle \sigma, ss \rangle\}$.

Definition 4.2 *The observables of the class $ACSP_{\mathcal{G}}$ are given by the function $\mathcal{S}'$:$ACSP_{\mathcal{G}} \to State \to (State \times \{ss, dd\})$ defined as follows:*

$$\mathcal{S}' = \mathcal{R}' \circ \mathcal{M}'(\emptyset)$$

where $\emptyset$ represents the function that assigns to every channel a empty multiset (at the beginning of the computation, all buffers are assumed to be empty).

5 The ACSP hierarchy

In this section we study the relation between $\mathrm{ACSP}_{\emptyset}$, $\mathrm{ACSP}_{\mathcal{I}}$ and $\mathrm{ACSP}_{\mathcal{I} \cup \mathcal{O}}$ (denoted by ACSP).

As in the synchronous class, we have the obvious embeddings due to the subset relations:

Proposition 5.1 $\mathrm{ACSP}_{\emptyset} \leq \mathrm{ACSP}_{\mathcal{I}} \leq \mathrm{ACSP}$.

The relation between $\mathrm{ACSP}_{\emptyset}$ and $\mathrm{ACSP}_{\mathcal{I}}$ is strict, namely:

Theorem 5.2 $\mathrm{ACSP}_{\mathcal{I}} \not\leq \mathrm{ACSP}_{\emptyset}$

Proof Consider the processes $A_1 = c_1??x_1 \rightarrow Stop$, $A_2 = c_2??x_2 \rightarrow Stop$, and $B = c_1!!v; Stop.$ with $c_1 \neq c_2$, and let $A = A = A_1 + A_2$. We have that

$$\mathcal{S}[\![A \parallel B]\!](\epsilon) = \{\langle\{v_1/x_1\}, ss\rangle\}. \tag{4}$$

The rest of the proof proceeds as in Theorem 3.4, by showing that there exists an element $\langle\vartheta, dd\rangle \in \mathcal{S}[\![\mathcal{C}(A \parallel B)]\!](\epsilon)$, which, together with (4), contradicts **P3**. □

One might now expect a separation between ACSP and $\mathrm{ACSP}_{\mathcal{I}}$, like the one for CSP and $\mathrm{CSP}_{\mathcal{I}}$, but this is not the case. In fact, there is an asymmetry between ACSP and CSP: the output actions, in ACSP, do not synchronize with the corresponding input actions, they can always proceed independently from the other processes. As a consequence, a choice guarded by output actions is an *internal choice*. This suggests how to compile ACSP into $\mathrm{ACSP}_{\mathcal{I}}$: it is sufficient to transform the output guarded statements in sequential statements guarded by a *Skip* action. All the others operators are left unchanged.

Definition 5.3 (A compiler $\mathcal{C}$ from ACSP into $\mathrm{ACSP}_{\mathcal{I}}$) . *The only non-trivial case is given by the translation of the statement* $c!! \rightarrow A$*:*

$$\mathcal{C}(c!!t \rightarrow A) = Skip \rightarrow c!!t; \mathcal{C}(A)$$

The compiler $\mathcal{C}$, and the decoder $\mathcal{D}$ defined as the identity function, obviously satisfy the conditions **P1-P3**. Furthermore, they constitute an embedding of ACSP into $\mathrm{ACSP}_{\mathcal{I}}$.

Theorem 5.4 $\mathrm{ACSP} \leq \mathrm{ACSP}_{\mathcal{I}}$.

The proof is based on the following lemma, which shows the correctness of the compiler defined above.

Lemma 5.5 *For every statement* A *in* ACSP, $b \in Chan \rightarrow \mathcal{P}_m(Val)$, *state* σ, *we have*

$$\mathcal{R}'(\mathcal{M}'[\![A]\!])(b)(\sigma) = \mathcal{R}'(\mathcal{M}'[\![\mathcal{C}(A)]\!])(b)(\sigma)$$

Proof Induction on the structural complexity of A. □

6 Comparing CSP and ACSP

The main result of this section is the separation between CSP and ACSP. This result follows from the relations established so far and the the fact that $\mathrm{ACSP}_{\mathcal{I}}$ can be embedded into $\mathrm{CSP}_{\mathcal{I}}$, that we show now.

The only non-trivial case, in the definition of a compiler, is the translation of communication actions. The asynchronism between an output action and the corresponding input action can be simulated, in the synchronous framework, in the following way. For every process which performs an output action on c we create a parallel process which behaves like a sort of one-position buffer: it inputs the value and outputs it later on. In order to prevent the sender to communicate directly with the receiver (that would cause the buffer to become blocked) the original output and the input of the buffer will take place on a different channel b_c.

Note that this solution would work also in case of (recursively defined) infinite processes, but we must then assume that communication is many-to-many (since many parallel one-position buffer processes inputting on b_c or outputting on c can be created), and it works only for bag-like channels (since the parallel one-position buffers may interleave the output actions in arbitrary ways). However, if we assume one-to-one communication, there is an alternative way to simulate asynchronous communication in $\mathrm{CSP}_{\mathcal{I}}$ by means of ordered buffers (see [4, 12]). In case of infinite processes the buffers will have to be unbounded (i.e. the corresponding processes can be infinite).

Definition 6.1 (A compiler $\mathcal{C}$ from $\mathrm{ACSP}_{\mathcal{I}}$ into $\mathrm{CSP}_{\mathcal{I}}$) .

- $\mathcal{C}(Stop) = Stop$
- $\mathcal{C}(c!!t; A) = (b_c!t; \mathcal{C}(A)) \parallel (b_c?x \rightarrow c!x)$ *where* b_c *is a new channel not occurring in* $\mathcal{C}(A)$.

- $\mathcal{C}(c??x;A) = c?x;\mathcal{C}(A)$
- $\mathcal{C}(a;A) = a;\mathcal{C}(A)$
- $\mathcal{C}(A_1 \parallel A_2) = \mathcal{C}(A_1) \parallel \mathcal{C}(A_2)$
- $\mathcal{C}(A_1 + A_2) = \mathcal{C}(A_1) + \mathcal{C}(A_2)$
- $\mathcal{C}(c??x \to A) = c?x \to \mathcal{C}(A)$
- $\mathcal{C}(a \to A) = a \to \mathcal{C}(A)$

The compiler $\mathcal{C}$, and the decoder $\mathcal{D}$ defined as the identity function, obviously satisfy the conditions **P1-P3**. Furthermore, they constitute an embedding of $\mathrm{ACSP}_{\mathcal{I}}$ into $\mathrm{CSP}_{\mathcal{I}}$.

Note that it is not necessary to use a input guard in the definition of the buffer process, we could just use a sequential statement, namely $\mathcal{C}(c!!t;A) = (b_c!t;\mathcal{C}(A)) \parallel (b_c?x;c!x)$. This leads immediately to the definition of a compiler of $\mathrm{ACSP}_{\emptyset}$ in $\mathrm{CSP}_{\emptyset}$.

Theorem 6.2 $\mathrm{ACSP}_{\mathcal{I}} \leq \mathrm{CSP}_{\mathcal{I}}$, *and* $\mathrm{ACSP}_{\emptyset} \leq \mathrm{CSP}_{\emptyset}$.

The proof is based on the following lemma, which shows the correctness of the compiler(s) defined above.

Lemma 6.3 *For every statement A of* $\mathrm{ACSP}_{\mathcal{I}}$ *(*$\mathrm{ACSP}_{\emptyset}$*), $b \in Chan \to \mathcal{P}_m(Val)$, state σ, we have*

$$\mathcal{R}'(\mathcal{M}'[\![A]\!])(b)(\sigma) = \mathcal{R}(\mathcal{M}[\![\mathcal{C}(A) \parallel A_b]\!])(\sigma)$$

where $A_b = \|_c\|_{v \in b(c)}\, c!v$.

Proof Induction on the structural complexity of A. □

Now we can prove the following theorem which separates CSP from ACSP:

Theorem 6.4 $\mathrm{CSP} \not\leq \mathrm{ACSP}$.

Proof Suppose that $\mathrm{CSP} \leq \mathrm{ACSP}$. Since $\mathrm{ACSP} \leq \mathrm{ACSP}_{\mathcal{I}} \leq \mathrm{CSP}_{\mathcal{I}}$, we would have that $\mathrm{CSP} \leq \mathrm{CSP}_{\mathcal{I}}$, which contradicts Theorem 3.3. □

Note that on the other hand we have $\mathrm{ACSP} \leq \mathrm{CSP}$.

There are still two questions: at which level $\mathrm{ACSP}_{\mathcal{I}}$ and $\mathrm{ACSP}_{\emptyset}$ are situated, with respect to $\mathrm{CSP}_{\mathcal{I}}$ and $\mathrm{CSP}_{\emptyset}$? The answer to these questions depends on the assumption that communication is one-to-one. If this is the case, then $\mathrm{ACSP}_{\mathcal{I}}$ can embed $\mathrm{CSP}_{\mathcal{I}}$, and $\mathrm{ACSP}_{\emptyset}$ can embed $\mathrm{CSP}_{\emptyset}$. In fact, synchronous communication can be simulated asynchronously via acknowledgement messages [21]: every process which performs an input action will send an acknowledgement, and each process which performs an output action will wait for the corresponding acknowledgement. Of course, this technique does not work when the communication is many-to-many: one sender could "steal" the acknowledgement directed to an other process.

Definition 6.5 (A compiler $\mathcal{C}$ from $\mathrm{CSP}_{\mathcal{I}}$ into $\mathrm{ACSP}_{\mathcal{I}}$) .

- $\mathcal{C}(Stop) = Stop$
- $\mathcal{C}(c!t;A) = c!!t;\, ack_c??x;\mathcal{C}(A)$
- $\mathcal{C}(c?x;A) = c??x;\, ack_c!!ok;\mathcal{C}(A)$
- $\mathcal{C}(a;A) = a;\mathcal{C}(A)$
- $\mathcal{C}(A_1 \parallel A_2) = \mathcal{C}(A_1) \parallel \mathcal{C}(A_2)$
- $\mathcal{C}(A_1 + A_2) = \mathcal{C}(A_1) + \mathcal{C}(A_2)$
- $\mathcal{C}(c?x \to A) = c??x \to ack_c!!ok;\mathcal{C}(A)$

- $\mathcal{C}(a \rightarrow A) = a \rightarrow \mathcal{C}(A)$

The compiler $\mathcal{C}$ from $\mathrm{CSP}_\emptyset$ into $\mathrm{ACSP}_\emptyset$ can be obtained just by dropping the last but one line in the previous definition.

The compiler $\mathcal{C}$, and the decoder $\mathcal{D}$ defined as the identity function, obviously satisfy the conditions **P1-P3**. Furthermore, they constitute an embedding of $\mathrm{ACSP}_\mathcal{I}$ into $\mathrm{CSP}_\mathcal{I}$ ($\mathrm{CSP}_\emptyset$ in $\mathrm{ACSP}_\emptyset$).

Theorem 6.6 *If the communication is one-to-one, then* $\mathrm{CSP}_\mathcal{I} \leq \mathrm{ACSP}_\mathcal{I}$, *and* $\mathrm{CSP}_\emptyset \leq \mathrm{ACSP}_\emptyset$.

The proof is based on the following lemma, which shows the correctness of the compiler(s).

Lemma 6.7 *For every statement* A *of* $\mathrm{CSP}_\mathcal{I}$ *(*$\mathrm{CSP}_\emptyset$*) and state* σ *we have*

$$\mathcal{R}(\mathcal{M}[\![A]\!])(\sigma) = \mathcal{R}'(\mathcal{M}'[\![\mathcal{C}(A)]\!])(\emptyset)(\sigma)$$

Proof Induction on the structural complexity of A. □

7 Conclusions and future work

We have applied our notion of embedding to establish a hierarchy between various sublanguages of CSP and ACSP, see Figure 4.

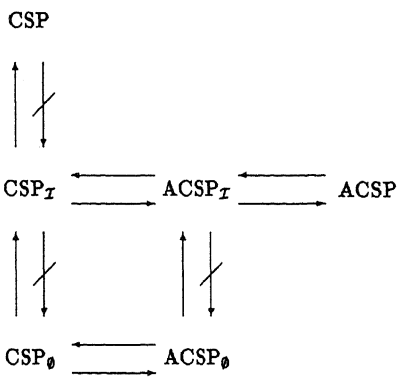

Figure 4: The CSP-ACSP hierarchy.

It is worthwhile to remark that the diagram actually should hold for every "reasonable" notion of embedding. In particular, $\mathrm{CSP}_\emptyset \leq \mathrm{CSP}_\mathcal{I} \leq \mathrm{CSP}$ and $\mathrm{ACSP}_\emptyset \leq \mathrm{ACSP}_\mathcal{I} \leq \mathrm{ACSP}$ hold for every notion of embedding subsuming the sublanguage relation, and $\mathrm{CSP} \not\leq \mathrm{CSP}_\mathcal{I} \not\leq \mathrm{CSP}_\emptyset$ is justified by the existence of algorithms expressible in one language and not in the other [3]. Furthermore, $\mathrm{ACSP} \leq \mathrm{ACSP}_\mathcal{I}$ holds because the local nature of asynchronous output guards makes them superfluous [13]. Finally, $\mathrm{ACSP}_\mathcal{I}$ can be implemented in $\mathrm{CSP}_\mathcal{I}$ by means of buffering techniques [12]. Thus, the difference in the expressivity of CSP and ACSP seems to be a general result.

If we compare the results of this paper with the ones in [8], 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 ACSP family! To our opinion, this is due to the fact that Concurrent Constraint has the possibility to express choices guarded by tell primitives, which enforce a test for consistency, so depending upon the previous tell actions done by the environment. This mutual dependency of the same kind of action cannot be expressed in any way in ACSP (input only depends on output, and output does not depend on any other action). In CSP we have an

indirect mutual dependency (input on output and output on input). Why exactly this mutual dependency determinates a growth in the expressive power, has still to be understood.

The characteristic features of our notion of embedding are compositionality of the compiler and termination invariance of the decoder. Actually, the requirement **P2** on the compiler is actually more restrictive than simple compositionality w.r.t. $+$ and $\|$ (i.e., the translation of these operators in a combination of operators), and it can be justified as follows. Since the languages we study extensions of each 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,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 in [10]). 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**.

It would be interesting to study the consequences of adopting a weaker notion of compositionality. More generally a systematic study on the kind of restrictions imposed by the different requirements seems to be of interest. To this end several case studies can be considered. For example, is there (and should there be) a difference in expressive power between many-to-many channels and one-to-one channels, and how does the expressive power of CSP relate to other asynchronous languages (like the concurrent logic languages)?

Acknowledgements We thank Jan-Willem Klop, Joost Kok, Jan Rutten and Ehud Shapiro for stimulating discussions and encouragements. We thank the members of the C.W.I. concurrency group, J.W. de Bakker, F. Breugel, A. de Bruin, J.M. Jacquet, P. Knijnenburg, J. Kok, J. Rutten, E. de Vink and J. Warmerdam for their comments on preliminary versions of this paper. We acknowledge the department of Software Technology of CWI, and the Department of Computer Science at Utrecht University, for providing a stimulating working environment. Finally, we thank Krzysztof Apt for having suggested us relevant literature on the subject.

References

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

[2] J.A. Bergstra and J.-W. Klop. Process algebra: specification and verification in bisimulation semantics. In *Mathematics and Computer Science II*, CWI Monographs, pages 61 – 94. North-Holland, 1986.

[3] L. Bougé. On the existence of symmetric algorithms to find leaders in networks of communicating sequential processes. *Acta Informatica*, 25:179–201, 1988.

[4] S.D. Brookes, C.A.R. Hoare, and W. Roscoe. A theory of communicating sequential processes. *Journal of ACM*, 31:499–560, 1984.

[5] A.K. Chandra and Z. Manna. The power of programming features. *J. Computer Languages*, 1:219–232, 1975.

[6] F.S. de Boer, J.N. Kok, C. Palamidessi, and J.J.M.M. Rutten. The failure of failures: Towards a paradigm for asynchronous communication. Technical Report RUU-CS-90-40, Department of Computer Science, University of Utrecht, 1990. A short version of this paper will appear in Proc. of CONCUR 91.

[7] 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.

[8] F.S. de Boer and C. Palamidessi. Embedding as a tool for language comparison. Technical Report CS-R9102, Centre for Mathematics and Computer Science (CWI), Amsterdam, 1991.

[9] R. de Simone. Higher-level synchronising devices in MEIJE-SCCS. *Theoretical Computer Science*, 37(3):245–267, 1985.

[10] M. Felleisen. On the expressive power of programming languages. In N. Jones, editor, *Proc. of the European Symposium on Programming*, volume 432 of *Lecture Notes in Computer Science*, pages 134 – 151. Springer-Verlag, 1990. Full version to appear in *Science of Computer Programming*.

[11] C.A.R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, 1978.

[12] He Jifeng, M.B. Josephs, and C.A.R. Hoare. A theory of synchrony and asynchrony. In *Proc. of IFIP Working Conference on Programming Concepts and Methods*, pages 459–478, 1990.

[13] M.B. Josephs, C.A.R. Hoare, and He Jifeng. A theory of asynchronous processes. Technical report, Oxford University Computing Laboratories, 1990.

[14] P.J. Landin. The next 700 programming languages. *Communications of the ACM*, 3(9):157–166, 1966.

[15] J. Parrow. The expressive power of parallelism. In *Proc. of PARLE 89*, volume 366 of *Lecture Notes in Computer Science*, pages 389–405. Springer-Verlag, 1989. Revised and extended version in SICS Research Report R90016.

[16] M.S. Paterson and C.E. Hewitt. Comparative schematology. In *Conf. Rec. ACM Conference on Concurrent Systems and Parallel Computation*, pages 119–127, 1970.

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

[18] J.C. Reynolds. The essence of Algol. In J. de Bakker and van Vliet, editors, *Algorithmic Languages*, pages 345–372. North-Holland, Amsterdam, 1981.

[19] 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.

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

[21] E.Y. Shapiro. Separating concurrent languages with categories of language embeddings. In *Proc. of STOCS*, 1991. To appear.

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

[23] F. Vaandrager. Expressiveness results for process algebra. Technical report, MIT Lab. for Comp. Sci., Cambridge, USA, 1991.