The failure of failures: towards a paradigm for asynchronous communication

F.S. de Boer, J.N. Kok, C. Palamidessi, J.J.M.M. Rutten

RUU-CS-90-40
December 1990

Utrecht University

Department of Computer Science

Padualaan 14, P.O. Box 80.089,
3508 TB Utrecht, The Netherlands,
Tel. : ... + 31 - 30 - 531454

The failure of failures: towards a paradigm for asynchronous communication

F.S. de Boer, J.N. Kok, C. Palamidessi, J.J.M.M. Rutten

Technical Report RUU-CS-90-40
December 1990

Department of Computer Science
Utrecht University
P.O.Box 80.089
3508 TB Utrecht
The Netherlands

ISSN:0924-3275

The Failure of Failures: Towards a Paradigm for Asynchronous Communication *

F.S. de Boer[1], J.N. Kok[2], C. Palamidessi[2,3], J.J.M.M. Rutten[3]

[1]Department of Computer Science, Technical University Eindhoven,
P.O. Box 513, 5600 MB Eindhoven, The Netherlands

[2]Department of Computer Science, Utrecht University,
P.O. Box 80089, 3508 TB Utrecht, The Netherlands

[3]Centre for Mathematics and Computer Science,
Kruislaan 413, 1098 SJ Amsterdam, The Netherlands

Abstract

We develop a general framework to study the variety of concurrent languages based on asynchronous communication, like data flow, concurrent logic, concurrent contraint languages and CSP with asynchronous channels. The main characteristic of this languages is that process interacts by reading and modifying the state of some common data structure. We abstract from the specific features of the various communication mechanisms by means of a uniform language where actions are interpreted as partial transformations on an abstract set of states. Suspension is modeled by an action being undefined in a state. The languages listed above can be seen as instances of our paradigm, obtained by fixing a specific set of states and interpretation of the actions.

The computational model of our paradigm is described by a transition system. We show that the traces generated by it already constitute a compositional semantics. This sharply contrasts with the synchronous case, where some additional branching information (like failure sets) is needed to describe deadlock. The specific features of a particular language come to surface when we consider the problem of full abstraction. These features are reflected by closure conditions on the set of states.

Finally, by means of a formal method for language comparison (embedding) we investigate the expressive power of our paradigm; we show that, under certain reasonable restrictions, it is not possible to embed CCS.

1 Introduction

In this paper we propose a general paradigm for asynchronously communicating processes. Such a paradigm should encompass such diverse systems as described by concurrent logic languages [Sha89], concurrent constraint languages [Sar89], imperative languages in which

*Part of this work was carried out in the context of ESPRIT Basic Research Action (3020) Integration. The research of F.S. de Boer was partially supported by the Dutch REX (Research and Education in Concurrent Systems) project and the Basic Research Action SPEC.

processes communicate by means of shared variables [HdBR90], or asynchronous channels [JJH90], and dataflow languages [Kah74].

These systems have in common that processes communicate via some shared data structure. The asynchronous nature of the communication consists in the way access to these shared data structure is modelled: the data structure is updated by means of write primitives which have free access whereas the read primitives may suspend in case the data structure does not contain the information required by it. The execution of the read and write primitives are independent in the sense that they can take place at different times. This marks an essential difference with synchronously communicating processes, like CSP [Hoa78], where reading from and writing to a channel has to take place at the same time.

Our paradigm consists of a concurrent language $\mathcal{L}$ which assumes given a set of basic (or atomic) actions. Statements are constructed from these actions by means of sequential composition, the plus operator for nondeterministic choice, and the parallel operator. (For simplicity, only finite behaviour is considered; recursion can be added straightforwardly.) Furthermore we assume given an abstract set of states. The basic actions are interpreted by means of an interpretation function I as partially defined state transformations. A pure read action a (like, e.g., a test) will have the property that $I(a)$ is a partial function; it suspends in a state in which $I(a)$ is undefined. A suspended process is forced to wait until actions of other processes produce a state in which it is enabled. A pure write action a is characterized by the fact that $I(a)$ is a totally defined function. It can always proceed autonomously. In general, an action can embody both a read and a write component. (See Example 1 of Section 2.)

Many languages for asynchronously communicating processes can be obtained as instances of our paradigm by choosing the appropriate set of actions, the set of states and the interpretation function for the basic actions. For example, the imperative language described in [HdBR90], based on shared variables, can be modelled by taking as states functions from variables to values, as actions the set of assignments, and then the usual interpretation of an assignment as a state transformation. Languages based on the *blackboard model* [EHLR80], like Linda [Gel86] and Shared Prolog [BC89] can be modelled analogously, by taking as states the configurations of a centralized data structure (the blackboard) and as actions checks and updates of the blackboard. Another example is the class of concurrent constraint languages [Sar89]. These are modelled by interpreting the abstract set of states as a constraint system and the actions as ask/tell primitives. Concurrent logic languages, like Flat Concurrent Prolog [Sha89], can be obtained by interpreting the states as the bindings established on the logical variables, and the actions as the unification steps. An asynchronous variant of CCS [Mil80, Mil83] is modelled by considering the state as a set (or a multi-set) of actions. Performing an action then corresponds to adding it to the set, while performing the complementary action corresponds to testing whether the action is already in the set. Finally, a variant of CSP [Hoa78], based on asynchronous channels (see also [JJH90]), can be obtained by taking as states the configurations of the channels and as actions the input-output primitives on these channels.

The basic computation model of the paradigm $\mathcal{L}$ is described by means of a labelled transition system. It specifies for every statement what steps it can take. Each step results in a state transformation, which is registered in the label: as labels we use pairs of states. Based on this transition system, various notions of observables for our language are defined. One of the results of this paper is a compositional characterization of these notions

of observables, defined independently of the particular choice for the sets of actions, the set of states, and the interpretation function. Thus a general compositional description of all the possible mechanisms for asynchronous communication is provided, so unifying the semantic work done in such apparently diverse fields as concurrent logic programming, dataflow, and imperative programming based on asynchronous communication.

The most striking feature of our compositional semantics is that it is essentially based on traces, namely sequences of pairs of states. A pair encodes the state transformation occurred during a transition step (initial state - final state). These sequences are not necessarily connected, i.e. the final state of a pair can be different from the initial state of the following pair. These "gaps" represent, in a sense, the possible steps made by the environment. Since there is no way to synchronize on actions, the behaviour of processes just depends upon the state. Therefore, such a set of sequences encodes all the information necessary for a compositional semantics. We do not need additional structures like failure sets, which are needed to describe deadlock in the case of synchronously communicating processes. We show that our model is more abstract than the classical failure set semantics, and it is argued that the latter model is not appropriate for describing our language and, therefore, not for describing asynchronous communication in general.

Another contribution of the paper is the identification of two classes of interpretations, for which we prove our semantics to be fully abstract. The first class, the elements of which are called *complete*, characterizes in an abstract setting the imperative paradigm. The second class characterizes asynchronous communication of the concurrent constraint paradigm. Such interpretations are called *monotonic*.

Having shown how most of the asynchronous systems can be seen as instances of our paradigm, it is natural to raise the question what languages can be modelled by it. For example, it would be interesting to see whether or not our paradigm is expressive enough to *simulate* synchronous communication mechanisms. The investigation of this problem requires the formalization of the concept of simulation. To this purpose we consider the notion of embedding, first proposed by Shapiro [Sha89], and extended in [dBP90a] to concurrent languages.

An embedding has two components: a compiler and a decoder [Sha89]. Without any restrictions on these components, any language can be embedded. In [dBP90a], a notion of embedding is provided which requires the compiler to be compositional and the decoder to preserve termination modes. We show that even under these conditions any language can be embedded into $\mathcal{L}$. We propose a third restriction to make the notion of embedding non trivial.

Additionally, an analysis is given under what further conditions CCS cannot be embedded. One of them is the requirement that basic actions are *extending*. Since both concurrent logic and constraint languages satisfy this requirement, this result suggests that those languages are not able to mimic CCS behaviour. We think that further development of this kind of argument will provide deeper insight into the differences and similarities between synchronous and asynchronous communication.

1.1 Comparison with related work

In spite of the general interest for asynchronous communication as the natural mechanism for concurrency in many different programming paradigms, not much work has been done in the past, neither for defining an uniform framework, nor for providing the ap-

propriate semantics tools for reasoning about such a mechanism in an abstract way. In most cases, asynchronous languages have been studied as special instances of the synchronous paradigm. For example, in [GCLS88, GMS89] the semantics of FCP is defined by using the failure set semantics of TCSP [BHR84], and [SR90] uses for a concurrent constraint language the bisimulation equivalence on trees of CCS [Mil80]. Only recently [dBP90b, dBP90c] it has been shown that concurrent logic and constraint languages do not need to code the branching structure in the semantic domain: linear sequences of assume/tell-constraints are sufficiently expressive for defining a compositional model (not only for the success, but also for the deadlock case). This model can be seen as an instance of ours, in fact, the tell-constraints correspond to the steps made by the process and coded by the pairs, while the assume-constraints represent the assumptions about the steps made by the environment, that, as explained above, are represented by the "gaps" between pairs.

In the field of data flow, compositional models based on linear sequences have been developed (for example in [Jon85]) and have been shown to be fully abstract (see [Kok87] for the first result not depending on fairness notions). For an overview consult [Kok89]. A related paradigm (abstract processes communicating via asynchronous channels) has been recently studied in [JHJ90]. Also in this paper the authors propose a linear semantics of input-output events, as opposed to the failure set semantics. Again, this model can be obtained as an instance of ours by interpreting the events as state transformations.

Finally, in [HdBR90], a semantics based on sequences of pair of states, similar to the one we study in this paper, has been developed for an imperative language and shown correct and fully abstract with respect to successful computations.

The main contribution of this paper is the generalization of the results obtained in [dBP90b, dBP90c, JHJ90, HdBR90] to a paradigm for asynchronous communication, and thus providing a uniform framework for reasoning about this kind of concurrency.

1.2 Plan of the paper

In the next section we start by presenting our paradigm. We give the syntax and the computational model (characterizing the observational behaviour) of the language, and show that many formalisms for asynchronous processes can be obtained as instances of it. Next a compositional model is introduced that is correct with respect to the observational model. In Section 3 we compare this compositional semantics with the standard notion of failure set semantics. We show that our model is more abstract and we characterize the redundancy present in the latter (when applied to asynchronous languages). The issue of full abstraction is addressed in Section 4. We give conditions on the interpretations, corresponding to various classes of languages, under which a number of full abstraction results (possibly after some closure operation) can be obtained. Finally, in Section 5 we compare the expressive power of asynchronous and synchronous communication, by discussing some conditions under which CCS can or cannot be embedded into our paradigm. To this end, a formalization of the notion of embedding is provided. Section 6 briefly sketches some future research.

2 The language and its semantics

Let $(a \in)A$ be a set of *atomic actions*. We define the set $(s \in)\mathcal{L}$ of statements as follows:

$$s ::= a \mid s; t \mid s + t \mid s \parallel t$$

Moreover, $\mathcal{L}$ contains a special E, the terminated statement. The symbols ';', '+' and '$\parallel$' represent the sequential, the choice and the parallel operator respectively. Note that we do not include any constructs for recursion. In this paper, only finite behaviour is studied for the sake of simplicity. All the results that follow can be extended as to cover also infinite behavior[1].

The actions of our language are interpreted as transformations on a set $(\sigma \in)\Sigma$ of abstract *states*. We assume given an interpretation function I of type

$$I : A \to \Sigma \to_{partial} \Sigma$$

that maps atomic actions to partially defined state transformations. The computational model of $\mathcal{L}$ is described by a ***labelled transition system*** $(\mathcal{L}, Label, \to)$. The set $(\lambda \in)Label$ of labels is defined by $Label = \Sigma \times \Sigma$. A label represents the state tranformation caused by the action performed during the transition step. The transition relation $\to \subseteq \mathcal{L} \times Label \times \mathcal{L}$ is defined as the smallest relation satisfying the following axiom and rules:

- If $I(a)(\sigma)$ is defined then

$$a \xrightarrow{(\sigma, I(a)(\sigma))} E$$

- If $s \xrightarrow{\lambda} s'$ then

$$s; t \xrightarrow{\lambda} s'; t \quad s + t \xrightarrow{\lambda} s' \quad t + s \xrightarrow{\lambda} s' \quad s \parallel t \xrightarrow{\lambda} s' \parallel t \quad t \parallel s \xrightarrow{\lambda} t \parallel s'$$

  If $s' = E$ then read t for $s'; t$, $s' \parallel t$ and $t \parallel s'$ in the clauses above.

Note that the transition relation depends on I.

There are various reasonable notions of observables for $\mathcal{L}$ that we can derive from the transition system above described. Given an initial state σ, the first observation criterium we consider, Obs_1, assigns to a statement a set of sequences of states. Each such sequence represents a possible computation of the statement, listing *all* intermediate states through which the computation goes. (Such a sequence may end in δ, indicating *deadlock*, if no transitions are possible anymore and the end of the statement has not yet been reached.) As an example, we have that if

$$s \xrightarrow{(\sigma_0, \sigma_1)} s_1 \xrightarrow{(\sigma_1, \sigma_2)} \cdots \xrightarrow{(\sigma_{n-2}, \sigma_{n-1})} s_{n-1} \xrightarrow{(\sigma_{n-1}, \sigma_n)} s_n = E$$

then $\sigma_0 \cdot \sigma_1 \cdots \sigma_n \in Obs_1[\![s]\!](\sigma)$. Note that in the definition of Obs_1, only *connected* transition sequences are considered: the labels of subsequent transitions have the property that the last element of the first label equals the first element of the second.

[1] In particular, the co-domains of our semantic models should then be turned into *complete* spaces of some kind in order to obtain infinite behaviour as limit of a sequence of finite approximations. A suitable framework would be the family of complete metric spaces (see [dBZ82]).

The second observation criterion we consider, Obs_2, is slightly more abstract than Obs_1: it also yields sequences of intermediate states, but omits subsequent repetitions of identical states. Thus it abstracts from so-called *finite stuttering*.

The third observation criterion is the most abstract one: it only registers final states for successfully terminating computations, and a pair of the final state together with δ for deadlocking computations.

Definition 2.1 *Let* $\Sigma_\delta^* = \Sigma^* \cup \Sigma^* \cdot \{\delta\}$ *and* $\Sigma_\delta = \Sigma \cup \{\delta\}$. *We define*

$$Obs_1 : \mathcal{L} \to \Sigma \to \mathcal{P}(\Sigma_\delta^*)$$

$$Obs_2 : \mathcal{L} \to \Sigma \to \mathcal{P}(\Sigma_\delta^*)$$

$$Obs_3 : \mathcal{L} \to \Sigma \to \mathcal{P}(\Sigma_\delta)$$

as follows. We put, for $i = 1, 2$,

$$Obs_i[\![E]\!](\sigma) = \{\sigma\}$$

where ϵ *denotes the empty sequence, and for* $s \neq E$,

$$Obs_1[\![s]\!](\sigma) = \bigcup\{\sigma \cdot Obs_1[\![s']\!](\sigma') : s \xrightarrow{(\sigma,\sigma')} s'\} \cup \{\sigma \cdot \delta : \forall\sigma'\forall s', \neg(s \xrightarrow{(\sigma,\sigma')} s')\}$$

$$\begin{aligned} Obs_2[\![s]\!](\sigma) = {} & \bigcup\{\sigma \cdot Obs_2[\![s']\!](\sigma') : s \xrightarrow{(\sigma,\sigma')} s' \wedge \sigma \neq \sigma'\} \cup \\ & \bigcup\{Obs_2[\![s']\!](\sigma) : s \xrightarrow{(\sigma,\sigma)} s'\} \cup \{\sigma \cdot \delta : \forall\sigma'\forall s', \neg(s \xrightarrow{(\sigma,\sigma')} s')\} \end{aligned}$$

Finally, we put $Obs_3 = \gamma \circ Obs_1$, *with* $\gamma : \mathcal{P}(\Sigma_\delta^*) \to \mathcal{P}(\Sigma_\delta)$ *defined by*

$$\gamma(X) = \{\sigma : w \cdot \sigma \in X, \textit{ for some } w\} \cup \{\delta : w \cdot \delta \in X, \textit{ for some } w\}$$

Note that the definitions of Obs_1 and Obs_2 are recursive. Since we do not treat infinite behaviour, they can be justified by a simple induction on the structure of statements. The relation between Obs_1 and Obs_2 is immediate: If we define, for a word $w \in \Sigma_\delta^*$, $del(w)$ to be the word obtained from w by deleting all subsequent occurrences of identical states, and $\beta : \mathcal{P}(\Sigma_\delta^*) \to \mathcal{P}(\Sigma_\delta^*)$ by

$$\beta(X) = \{del(w) : w \in X\}$$

then we have: $Obs_2 = \beta \circ Obs_1$.

Now we illustrate the generality of the language $\mathcal{L}$ and its semantics presented above. It does not constitute one language with a fixed semantics but rather an entire family, each member of which depends on the particular choice of A, Σ and I. We present a list of five examples. (In Section 5, an even more general strategy of embedding other languages into $\mathcal{L}$ will be discussed.)

Example 1: An imperative language

Let A be the set of assignments $x := e$, where $x \in Var$ is a variable and $e \in Exp$ is an expression. Assume that the evaluation $E(e)(\sigma)$ of expression e in state σ is simple in that it does not have side effects and is instantaneous. Let the set of states be defined by

$$\Sigma = Var \to Val$$

where *Val* is some abstract set of values. Then define I by

$$I(x := e)(\sigma)(y) = \begin{cases} \sigma(y) & \text{if } y \neq x \\ E(e)(\sigma) & \text{if } y = x \end{cases}$$

With this choice for A, Σ and I, the models Obs_1 and O (to be introduced below) for $\mathcal{L}$ are essentially the same as the operational and denotational semantics presented for a concurrent language with assignment in [HdBR90]

One could include in this language a suspension mechanism by associating with each assignment a boolean expression, which must be true to enable the execution of the assignment (otherwise it suspends). A basic action is then an object of the form $b.x := e$. Its interpretation is:

$$I(b.x := e)(\sigma)(y) = \begin{cases} \sigma(y) & \text{if } E(b)(\sigma) \text{ and } y \neq x \\ E(e)(\sigma) & \text{if } E(b)(\sigma) \text{ and } y = x \\ \text{undefined} & \text{otherwise} \end{cases}$$

Example 2: Concurrent constraint languages

Constraint programming is based on the notion of computing with systems of partial information. The main feature is that the state is seen as a constraint on the range of values that variables can assume, rather than a function from variables to values (valuation) as in the imperative case. In other words, the state is seen as a (possibly infinite) set of valuations. Constraints are just finite representations of these sets. For instance, a constraint can be a first order formula, like $\{x = f(y)\}$, representing the set $\{\{y = a, x = f(a)\}, \{y = b, x = f(b)\}, \ldots\}$. As discussed in [Sar89, SR90], this notion of state leads naturally to a paradigm for concurrent programming. All processes share a common *store*, i.e. a set of variables and the constraints established on them until that moment. Communication is achieved by adding (telling) some constraint to the store, and by checking (asking) if the store entails (implies) a given constraint. Synchronization is based on a blocking ask: a process waits (suspends) until the store is "strong" enough to entail a certain constraint.

A typical example of a constraint system is a decidable first-order theory, a constraint in this case being simply a first-order formula. Given a first-order language L and a theory T in L define A to be the set of ask and tell primitives, i.e., $A = \{ask(\vartheta), tell(\vartheta) : \vartheta \in L\}$. Let the set of states be defined by

$$\Sigma = \{\vartheta : \vartheta \in L\}$$

The interpretation function I is given by

$$I(ask(\vartheta))(\vartheta') = \begin{cases} \vartheta' & \text{if } T \cup \vartheta' \vdash \vartheta \\ \text{undefined} & \text{otherwise} \end{cases}$$

and

$$I(tell(\vartheta))(\vartheta') = \vartheta \wedge \vartheta'$$

Example 3: Asynchronous CCS

Let $(a \in) Act$ be a set of atomic actions and $\bar{Act} = \{\bar{a} : a \in Act\}$ a set of actions corresponding with the ones in Act. Let

$$A = Act \cup \bar{Act}, \quad \Sigma = \mathcal{P}(A)$$

Define I by $I(a)(\sigma) = \sigma \cup \{a\}$ and

$$I(\bar{a})(\sigma) = \begin{cases} \sigma & \text{if } a \in \sigma \\ \text{undefined} & \text{if } a \notin \sigma \end{cases}$$

With this interpretation, an asynchronous variant of CCS is modelled in the following sense. Actions $a \in Act$ are viewed as send actions; they can always proceed. The actions $\bar{a} \in \bar{Act}$ are the corresponding receive actions: an action $\bar{a}$ can only take place when at least one a action has been performed.

Another interpretation would be to take as states *multisets* of actions rather than plain sets. Thus one could register the number of a actions that have been performed, and decrease this number each time an $\bar{a}$ is executed.

Example 4: Concurrent Prolog
For our fourth example we refer to [dBK90], where a mapping is defined from the language Concurrent Prolog [Sha83] to a language similar to ours.

Example 5: Asynchronous CSP
We consider an asynchronous version of CSP as described in [JJH90, JHJ90]. Let $(x \in) Var$ be a set of variables, $(h \in) IExp$ a set of integer expressions, $(b \in) BExp$ a set of Boolean expressions, and $(\alpha \in) CName$ a set of channel names. Let the set of atomic actions a be defined by

$$A = BExp \cup \{\alpha?x : \alpha \in CName,\ x \in Var\} \cup \{\alpha!h : \alpha \in CName,\ h \in IExp\}$$

Let Val be a set of values. We take as set of states

$$\Sigma = (Var \to Val) \times (CName \to Val^*)$$

The interpretation function is defined as follows:

$$I(b)(\sigma) = \begin{cases} \sigma & \text{if } E(b)(\sigma) = true \\ \text{undefined} & \text{otherwise} \end{cases}$$

$$I(\alpha?x)(\sigma) = \begin{cases} \sigma\langle head(\sigma(\alpha))/x, tail(\sigma(\alpha))/\alpha\rangle & \text{if } \sigma(\alpha) \textit{ is non empty} \\ \text{undefined} & \text{otherwise} \end{cases}$$

$$I(\alpha!h)(\sigma) = \sigma\langle \sigma(\alpha) \cdot E(h)(\sigma)/\alpha\rangle$$

Note thata guarded command of the form $g_1 \to S_1 \Box \ldots \Box g_n \to S_n$ $(g_1, \ldots g_n \in A)$ will be translated as $g_1; S_1 + \ldots + g_n; S_n$.

A similar construction can be made for static dataflow along the lines of [JK89], in which a state based model for dataflow nets is given. (Already referring to the discussion in the next section, we observe that in the world of dataflow, it is already known for a long time that trace models are compositional (and even fully abstract).)
End of examples.

Next we introduce a semantics O that describes the behaviour of $\mathcal{L}$ in a compositional manner. It is introduced using the transition system, and is later shown to be compositional. (This way of introducing O has the advantage that its *correctness* can be easily proved (see Theorem 2.5).)

Definition 2.2 *Let $(X, Y \in)P$ be the set of all non-empty subsets of finite sequences of pairs of states, possibly ending in deadlock, formally $P = \mathcal{P}_{\mathrm{ne}}(Q)$, where $Q = (\Sigma \times \Sigma)^* \cup (\Sigma \times \Sigma)^* \cdot (\Sigma \times \{\delta\})$. We define $O : \mathcal{L} \to P$ as follows. We put $O[\![E]\!] = \{\epsilon\}$ and, for $s \neq E$,*

$$O[\![s]\!] = \bigcup\{(\sigma,\sigma') \cdot O[\![s']\!] : \ s \xrightarrow{(\sigma,\sigma')} s'\} \cup \{(\sigma,\delta) : \ \forall\sigma'\forall s', \ \neg(s \xrightarrow{(\sigma,\sigma')} s')\}$$

The function O yields sets of sequences of *pairs* of states, rather than just states. The intuition behind such a pair (σ, σ') is that if the current state is σ, then the computation at hand can transform this state into σ'. For instance, if

$$s \xrightarrow{(\sigma_1,\sigma_1')} s_1 \xrightarrow{(\sigma_2,\sigma_2')} \cdots \xrightarrow{(\sigma_{n-1},\sigma_{n-1}')} s_{n-1} \xrightarrow{(\sigma_n,\sigma_n')} s_n = E$$

then $(\sigma_1, \sigma_1') \cdots (\sigma_n, \sigma_n') \in O[\![s]\!]$. An important difference between the functions Obs_i and O is that in the definition of the latter, the transition sequences need not be connected: for instance, in the above example σ_1' may be different from σ_2.

The main interest of O lies in the fact that it is *compositional*. This we show next. To this end, semantic interpretations of the operators $;, +, \|$, denoted by the same symbols, are introduced.

Definition 2.3 *Three operators $;, +, \|: P \times P \to P$ are introduced as follows.*

- $X_1; X_2 = \{w_1 \hat{;} w_2 : \ w_1 \in X_1 \wedge w_2 \in X_2\}$,
  where $w_1 \hat{;} w_2 = \begin{cases} w_1 & \text{if } w_1 = w \cdot (\sigma, \delta) \\ w_1 \cdot w_2 & \text{otherwise} \end{cases}$

- $X_1 + X_2 = rem(X_1 \cup X_2)$,
  where $rem(X) = X \setminus \{(\sigma,\delta) : \ \exists\sigma', \ X_{(\sigma,\sigma')} \neq \emptyset\}$,
  and $X_{(\sigma,\sigma')} = \{w : \ w \in (\Sigma \times \Sigma_\delta)^* \wedge (\sigma,\sigma') \cdot w \in X\}$.

- $X_1 \parallel X_2 = \{w_1 \hat{\|} w_2 : \ w_1 \in X_1 \wedge w_2 \in X_2\}$,
  where $\hat{\|}, \mathbin{\hat{\perp\!\!\!\perp}} : Q \times Q \to P$ *are defined by induction on the length of words* $w_1, w_2 \in Q$ *as follows:*

  - $w_1 \hat{\|} w_2 = w_1 \mathbin{\hat{\perp\!\!\!\perp}} w_2 \cup w_2 \mathbin{\hat{\perp\!\!\!\perp}} w_1$

  - $w_1 \mathbin{\hat{\perp\!\!\!\perp}} w_2 = \begin{cases} (\sigma_1,\sigma_2) \cdot (w_1' \hat{\|} w_2) & \text{if } w_1 = (\sigma_1,\sigma_2) \cdot w_1' \\ (\sigma,\delta) & \text{if } w_1 = w_2 = (\sigma,\delta) \\ w_2 & \text{if } w_1 = \epsilon \\ \emptyset & \text{otherwise} \end{cases}$

The sequential compositon $;$ of X_1 and X_2 appends to the sequences of X_1 not ending in deadlock the sequences of X_2. The definition of the choice operator $+$ makes use of a function *rem* that removes those initial occurrences of deadlock (σ, δ) for which there is an alternative non-deadlocking pair present, that is, a pair (σ, σ'). The motivation is that if a process is in state σ and has the choice between a deadlocking step and a successful step, it will never choose the former. Note that this analysis of δ removal takes place only at the level of *initial* steps. The parallel composition is defined as interleaving: the merge operator $\|$, applied to two sets of sequences X_1 and X_2, takes all the possible interleavings

$w_1 \hat{\|} w_2$ of words $w_1 \in X_1$ and $w_2 \in X_2$. The set $w_1 \hat{\|} w_2$ is the union of two so-called left merges: $w_1 \mathbin{\mathbb{L}} w_2$ and $w_2 \mathbin{\mathbb{L}} w_1$. In $w_1 \mathbin{\mathbb{L}} w_2$, every word starts with a step from the left component w_1. This can be a delta step (σ, δ) only in the case that also w_2 starts with the same (σ, δ). If w_2 offers an alternative (σ, σ') as its first step, then the initial deadlock of w_1 is neglected: this is expressed by

$$\begin{aligned}(\sigma,\delta)\cdot w_1 \hat{\|} (\sigma,\sigma')\cdot w_2 &= (\sigma,\delta)\cdot w_1 \mathbin{\mathbb{L}} (\sigma,\sigma')\cdot w_2 \cup (sigma,\sigma')\cdot w_2 \mathbin{\mathbb{L}} (\sigma,\delta)\cdot w_1 \\ &= \emptyset \cup (\sigma,\sigma')\cdot (w_2 \hat{\|} (\sigma,\delta)\cdot w_1).\end{aligned}$$

Theorem 2.4 (Compositionality of O) *For all $s, t \in \mathcal{L}$, $* \in \{;, +, \|\}$,*

$$O[\![s * t]\!] = O[\![s]\!] * O[\![t]\!]$$

Proof

1. The case that $*$ is $;$ is trivial.

2. $$\begin{aligned} O[\![s+t]\!] = \quad & \bigcup\{(\sigma,\sigma')\cdot O[\![s']\!] : s \xrightarrow{(\sigma,\sigma')} s'\} \\ \cup \quad & \bigcup\{(\sigma,\sigma')\cdot O[\![t']\!] : t \xrightarrow{(\sigma,\sigma')} t'\} \\ \cup \quad & \{(\sigma,\delta) : \forall\sigma'\forall u,\ \neg(s+t \xrightarrow{(\sigma,\sigma')} u)\} \\ \\ = \quad & \bigcup\{(\sigma,\sigma')\cdot O[\![s']\!] : s \xrightarrow{(\sigma,\sigma')} s'\} \\ \cup \quad & \bigcup\{(\sigma,\sigma')\cdot O[\![t']\!] : t \xrightarrow{(\sigma,\sigma')} t'\} \\ \cup \quad & (\{(\sigma,\delta) : \forall\sigma'\forall s',\ \neg(s \xrightarrow{(\sigma,\sigma')} s')\} \cap \{(\sigma,\delta) : \forall\sigma'\forall t',\ \neg(t \xrightarrow{(\sigma,\sigma')} t')\}) \\ \\ = \quad & (\bigcup\{(\sigma,\sigma')\cdot O[\![s']\!] : s \xrightarrow{(\sigma,\sigma')} s'\} \cup \{(\sigma,\delta) : \forall\sigma'\forall s',\ \neg(s \xrightarrow{(\sigma,\sigma')} s')\}) \\ + \quad & (\bigcup\{(\sigma,\sigma')\cdot O[\![t']\!] : t \xrightarrow{(\sigma,\sigma')} t'\} \cup \{(\sigma,\delta) : \forall\sigma'\forall t',\ \neg(t \xrightarrow{(\sigma,\sigma')} t')\}) \\ \\ = \quad & O[\![s]\!] + O[\![t]\!] \end{aligned}$$

3. We prove now that $O[\![s \parallel t]\!] = O[\![s]\!] \parallel O[\![t]\!]$. Let's first introduce some notation: we write $s \overset{w}{\Rightarrow} s'$ for

$$\begin{aligned}&\exists s_1, \ldots s_n \in \mathcal{L}, \sigma_1, \sigma_1', \ldots \sigma_{n-1}, \sigma_{n-1}' \in \Sigma, \\ &s = s_1 \xrightarrow{(\sigma_1,\sigma_1')} s_2 \xrightarrow{(\sigma_2,\sigma_2')} s_3 \cdots \xrightarrow{(\sigma_{n-1},\sigma_{n-1}')} s_n = s' \ \wedge\ w = (\sigma_1,\sigma_1')\cdots(\sigma_{n-1},\sigma_{n-1}').\end{aligned}$$

   By the definition of O and $\|$ it is sufficient to consider only words of the form $w \in (\Sigma \times \Sigma)^*$ and $w \cdot (\sigma, \delta) \in (\Sigma \times \Sigma)^* \cdot (\Sigma \times \{\delta\})$. We have the following equivalent statements.

   Case 1: $w \in (\Sigma \times \Sigma)^*$

$$\begin{aligned} w \in O[\![s \parallel t]\!] &\Leftrightarrow s \parallel t \overset{w}{\Rightarrow} E \\ &\Leftrightarrow \exists w_1, w_2 : s \overset{w_1}{\Rightarrow} E \ \wedge\ t \overset{w_2}{\Rightarrow} E \ \wedge\ w \in w_1 \hat{\|} w_2 \\ &\Leftrightarrow \exists w_1, w_2 : w_1 \in O[\![s]\!] \ \wedge\ w_2 \in O[\![t]\!] \ \wedge\ w \in w_1 \hat{\|} w_2 \\ &\Leftrightarrow w \in O[\![s]\!] \parallel O[\![t]\!]. \end{aligned}$$

Case 2: $w \cdot (\sigma, \delta) \in (\Sigma \times \Sigma)^* \cdot (\Sigma \times \{\delta\})$

$$
\begin{aligned}
w \cdot (\sigma,\delta) \in O[\![s \parallel t]\!] \quad \Leftrightarrow \quad & \exists \bar{s} \in \mathcal{L}, \forall \sigma' \in \Sigma, u \in \mathcal{L} : \\
& s \parallel t \overset{w}{\Rightarrow} \bar{s} \ \wedge \ \neg(\bar{s} \overset{(\sigma,\sigma')}{\Rightarrow} u) \ \wedge \ \bar{s} \neq E \\
\\
\Leftrightarrow \quad & \exists w_1, w_2, \exists s', t', \forall \sigma \in \Sigma, u \in \mathcal{L} : \\
& s \overset{w_1}{\Rightarrow} s' \ \wedge \ t \overset{w_2}{\Rightarrow} t' \ \wedge \ w \in w_1 \hat{\|} w_2 \ \wedge \\
& (\textit{if } s' \neq E \textit{ then } \neg(s' \overset{(\sigma,\sigma')}{\Rightarrow} u)) \ \wedge \\
& (\textit{if } t' \neq E \textit{ then } \neg(t' \overset{(\sigma,\sigma')}{\Rightarrow} u)) \ \wedge \\
& (s' \neq E \ \vee \ t' \neq E) \\
\\
\Leftrightarrow \quad & (\textit{treating only the most interesting case} : \\
& s' \neq E \ \wedge \ t' \neq E) \\
& \exists w_1, w_2 : w_1 \cdot (\sigma,\delta) \in O[\![s]\!] \ \wedge \\
& w_2 \cdot (\sigma,\delta) \in O[\![t]\!] \ \wedge \\
& w \in w_1 \hat{\|} w_2 \\
\\
\Leftrightarrow \quad & \exists w_1, w_2 : w_1 \cdot (\sigma,\delta) \in O[\![s]\!] \ \wedge \\
& w_2 \cdot (\sigma,\delta) \in O[\![t]\!] \ \wedge \\
& w \cdot (\sigma,\delta) \in w_1 \cdot (\sigma,\delta) \hat{\|} w_2 \cdot (\sigma,\delta) \\
\\
\Leftrightarrow \quad & w \cdot (\sigma,\delta) \in O[\![s]\!] \parallel O[\![t]\!].
\end{aligned}
$$

□

Finally, it is shown that O is *correct* with respect to the observation functions Obs_i, for $i = 1, 2, 3$. That is, if two statements are distinguished by any of these functions, then O should distinguish them as well. We start by showing that O is correct with respect to Obs_1, from which the other two cases will follow. The relation between Obs_1 and O can be made precise using the following abstraction operator. Let $\alpha : P \to \Sigma \to \mathcal{P}(\Sigma_\delta^*)$ be defined by

$$
\alpha(X)(\sigma) = \begin{cases} \{\sigma\} & \textit{if } X = \{\epsilon\} \\ \bigcup\{\sigma \cdot \alpha(X_{(\sigma,\sigma')})(\sigma') : \ X_{(\sigma,\sigma')} \neq \emptyset\} \cup \{\sigma \cdot \delta : \ (\sigma,\delta) \in X\} & \textit{otherwise} \end{cases}
$$

The operator α picks from a set X, given an initial state σ, all connected sequences; for each such sequence, α delivers the sequence of the second elements of the pairs occurring in that sequence.

Theorem 2.5 (Correctness of O) $\alpha \circ O = Obs_1$

Proof

$$
\begin{aligned}
\alpha(O[\![s]\!])(\sigma) &= \alpha(\bigcup\{(\sigma_1,\sigma_2) \cdot O[\![s']\!] : \ s \overset{(\sigma_1,\sigma_2)}{\longrightarrow} s'\} \cup \{(\sigma_1,\delta) : \ \forall \sigma_2 \forall s', \ \neg(s \overset{(\sigma_1,\sigma_2)}{\longrightarrow} s')\})(\sigma) \\
&= \bigcup\{\sigma \cdot \alpha(O[\![s']\!])(\sigma') : \ s \overset{(\sigma,\sigma')}{\longrightarrow} s'\} \cup \{\delta : \ \forall \sigma' \forall s', \ \neg(s \overset{(\sigma,\sigma')}{\longrightarrow} s')\} \\
&= (\textit{induction}) \ \bigcup\{\sigma \cdot Obs_1[\![s']\!](\sigma') : \ s \overset{(\sigma,\sigma')}{\longrightarrow} s'\} \cup \{\delta : \ \forall \sigma' \forall s', \ \neg(s \overset{(\sigma,\sigma')}{\longrightarrow} s')\} \\
&= Obs_1[\![s]\!](\sigma)
\end{aligned}
$$

□

As a corollary, the correctness of Obs_2 and Obs_3 is immediate:

$$Obs_2 = \beta \circ Obs_1 = \beta \circ \alpha \circ O \quad Obs_3 = \gamma \circ Obs_1 = \gamma \circ \alpha \circ O$$

3 Comparison with failure semantics

The reader familiar with the various semantic models for synchronous concurrency (as exemplified by languages like, e.g., CCS [Mil80] and TCSP [BHR84]) might be surprised and might even be made somewhat suspicious by the simplicity of the model O. It is well known that the most abstract semantics for CCS, called *fully* abstract, that is still compositional and correct (with respect to the standard trace semantics) is *failure* semantics, and that the trace semantics for CCS is *not* compositional. (See [BHR84] and [BKO88].) How come that for the present language failure semantics is not needed and already a trace-like model (O) is compositional?

Failure semantics assigns to statements sets of streams of actions possibly ending in a so-called failure set of actions. The intended meaning of a stream of actions ending in a failure set is the following: after having performed the actions in the stream, the statement can refuse the actions present in the failure set. That is, if the environment offers one or more (possibly all) of these actions, then the parallel composition of the environment and this statement will deadlock. Note that the environment can offer (by means of the plus operator for nondeterministic choice) more than one action at the same time.

For our language $\mathcal{L}$, one could try to mimic this approach and define a failure semantics as well. The first thing to be observed is that it would not make sense to have sets of *actions* for the failure sets, because an action may or may not deadlock depending on the current state ($I(a)$ is in general a *partial* function). Therefore it would be better to take sets of *states*, namely those that would yield a deadlock when generated by the environment. Now the main difference between the synchronous and the asynchronous case is essentially that, in the asynchronous case, there is a fundamental difference between the role of the actions, on the one hand, and the states, on the other. Consider a statement s in an environment. At any moment in the computation the way in which this environment can influence the possible behaviour of s is not so much determined by the fact that the environment can choose among a set of actions. What *is* relevant is the fact that corresponding to each such choice there will be a new state, which *does* influence the activity of s: in this new state, s may or may not be able to perform some action that is enabled. In other words, although the environment can perform different *actions*, it cannot simultaneously offer different *states*. At every moment in the computation there is only one relevant state (the "current" state).

The above should, at least at an intuitive level, make clear that it is sufficient to use failure sets that contain only one element. This, however, means abandoning the idea of failure *sets* altogether. The result is our semantic model O based on streams of pairs of states that can possibly end in a pair (σ, δ). The correspondence with the failure set semantics (containing only one element) is given by interpreting such a pair (σ, δ) at the end of a stream as a failure set with one element, σ. If the environment offers the state σ, then the statement will deadlock.

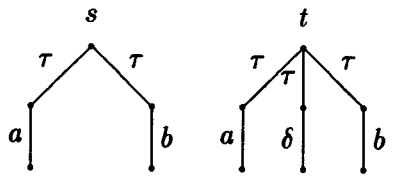

Figure 1: In case of asynchronous communication, s and t cannot be distinguished by any context.

Let us inspect a brief example to make this point more clear. Suppose that

$$A = \{a, b, \tau, \delta\}, \quad \Sigma = \{1, 2\}$$

and that $I : A \to \Sigma \to_{partial} \Sigma$ is defined by

$$I(a)(i) = \begin{cases} 1 & \text{if } i = 1 \\ \text{undefined} & \text{if } i = 2 \end{cases} \qquad I(b)(i) = \begin{cases} 2 & \text{if } i = 2 \\ \text{undefined} & \text{if } i = 1 \end{cases}$$

$$I(\tau)(i) = i, \text{ for } i = 1, 2, \quad I(\delta)(i) = \text{ undefined, for } i = 1, 2$$

Next consider two statements

$$s = \tau; a + \tau; b, \quad t = \tau; a + \tau; b + \tau; \delta$$

(see Figure 1).

Without giving any formal definitions, the failure semantics for s and t would be as follows (using $(*, *)$ to indicate either $(1, 1)$ or $(2, 2)$):

$$\mathit{failure}(s) = \{\emptyset,\ (*,*)(1,1)\emptyset,\ (*,*)(2,2)\emptyset,\ (*,*)\{1\},\ (*,*)\{2\},\ (*,*)\emptyset\}$$

$$\mathit{failure}(t) = \{\emptyset,\ (*,*)(1,1)\emptyset,\ (*,*)(2,2)\emptyset,\ (*,*)\{1\},\ (*,*)\{2\},\ (*,*)\{1,2\},\ (*,*)\emptyset\}$$

(Note that elements like $(*, *)\emptyset$ are present in $\mathit{failure}(s)$ and $\mathit{failure}(t)$ because these are closed, as in the standard failure semantics, under taking arbitrary subsets of failure sets: if wX is an element and $Y \subseteq X$, then also wY is an element.)

Clearly, the two statements are distinguished because of of the component $\tau; \delta$ in t, which does not occur in s: This explains the presence of $(*, *)\{1, 2\}$ in $\mathit{failure}(t)$. However, a moment's thought is sufficient to see that both s and t will have the same deadlock behaviour in all possible environments. For, any of the deadlock possibilities represented in t by $(*, *)\{1, 2\}$ occurs also in the meaning of s, namely in the shape of $(*, *)(1, 1)$ and $(*, *)(2, 2)$. Therefore s and t need not be distinguished. As we can see, O does not distinguish between the two statements:

$$O[\![s]\!] = O[\![t]\!] = \{(*,*)(1,1),\ (*,*)(2,2),\ (*,*)(1,\delta),\ (*,*)(2,\delta)\}$$

A remedy suggested by the above example would be to allow, in addition to the usual closure under arbitrary subsets, also the closure under taking arbitrary unions of failure

sets. It can be easily shown formally that such a closure condition would yield a correct compositional model. In fact, the resulting model would be isomorphic to our semantics O: allowing arbitrary unions of failure sets is equivalent to having one-element failure sets. Thus we find back the observation made above at an intuitive level.

We can conclude that for our asynchronous language $\mathcal{L}$ the failure model is not abstract enough (let alone be *fully* abstract), since it distinguishes more statements that necessary. Instead, a stream-like model O, which is also compositional but more abstract, is more suited. The question of a fully abstract model for $\mathcal{L}$ that is correct with respect to Obs will be treated in the next section.

4 Full abstraction

In this section we discuss the problem of the full abstraction of the compositional semantics O (with respect to the different notions of observability) for two classes of interpretations.

4.1 Complete interpretations

The first class is intended to characterize the asynchronous communication of the imperative paradigm. Such interpretations are called *complete* and satisfy the following two properties:

- For every $\sigma, \sigma' \in \Sigma$ there exists an atomic action $a \in A$ (also denoted by $a(\sigma, \sigma')$) such that

$$I(a)(\sigma^*) = \begin{cases} \sigma' & \text{if } \sigma^* = \sigma \\ \text{undefined} & \text{if } \sigma^* \neq \sigma \end{cases}$$

  If the current state is σ then the action $a(\sigma, \sigma')$ changes it to σ'; otherwise the actions blocks.

- For every $s, t \in \mathcal{L}$ and every $\sigma, \sigma' \in \Sigma$ there exists a state $\mu \in \Sigma$ (also denoted by $\mu(\sigma, \sigma')$) such that (σ, μ) and (μ, σ') do not occur in $O[\![s]\!] \cup O[\![t]\!]$.

Example We consider a variant of Example 1 of Section 2 with multiple assignment. Let A be the set of atomic actions that can istantaneously test and perform multiple assignment. It is easy to see that the first property is satisfied. For the second one, consider statements s and t. Let $y \in Var$ be a variable that does not occur in either s or t. (Thus s and t do not change the value of y.) Let $\sigma, \sigma' \in \Sigma$. Choose $v \in Val$ different from $\sigma(y)$ and $\sigma'(y)$. (Both Var and Val are assumed to be infinite.) Then define

$$\tau(\sigma, \sigma')(x) = \begin{cases} \sigma(x) & \text{if } x \neq y \\ v & \text{if } x = y \end{cases}$$

Now (σ, τ) and (τ, σ') do not occur in $O[\![s]\!] \cup O[\![t]\!]$.
End of example.

4.2 Monotonic interpretations

The second class we consider here is intended to characterize the asynchronous communication of the concurrent constraint paradigm. (For a short explanation see Example 2 of Section 2.) The elements of a constraint system are naturally ordered with respect to (reverse) logical implication. The effect of both the ask and tell primitives is *monotonic* with respect to the store, in fact tell only adds constraints to the store, while ask does not modify it. Since it is modeled by implication, also the successfull execution of ask depends monotonically on the store, i.e., if a certain ask is defined on σ, it will be defined in all the constraints bigger than σ. Both ask and tell are *extending*, i.e. their action on the store can only increase the store. An other characteristic is that both ask and tell are *strongly idempotent*: if their execution results in a certain store σ, to execute them again in the same store σ, or in a bigger store, will have no effect. This properties can be proved easily by the formalization of ask and tell (see Section 2). Notice that a consequence of them is that the store will monotonically increase during the execution.

We now formalize these features in our framework. Given a set of states Σ, where a state represents a store (constraint), and a partial order $\sqsubseteq$ on Σ, where $\sigma \sqsubseteq \sigma'$ should be read as "σ' encodes more information than (or the same as) σ", the behaviour of the ask and tell primitives is characterized by the following requirements on the interpretation function I:

- $\sigma \sqsubseteq \sigma' \Rightarrow I(a)(\sigma) \sqsubseteq I(a)(\sigma')$ (monotonicity)
- $I(a)(\sigma) \sqsubseteq \sigma' \Rightarrow I(a)(\sigma') = \sigma'$ (strong idempotency)
- $\sigma \sqsubseteq I(a)(\sigma)$ (extension)

for every action a, states σ and σ'. (Note that these notions are independent.)

Furthermore, for the construction of the full abstract semantics, we require the following assumption on the expressiveness of the atomic actions:

> **(Assumption)** For every $\sigma \sqsubseteq \sigma' \in \Sigma$ there exists an atomic action a such that
>
> 1. $I(a)(\sigma) = \sigma'$, and
> 2. for every $\sigma'' \sqsubseteq \sigma$ $(\sigma'' \neq \sigma)$, $I(a)(\sigma'')$ is undefined.

This assumption will be used to construct a distinguishing context in the full abstraction proof. Actually, what we exactly need, is the language to allow the construction of a compound statement having the same semantics $\mathcal{D}$ (see Section 4.4) as the action a above. In concurrent constraint languages, the above assumption corresponds to requiring the atomic actions to contain both the tell (1) and the ask (2) component (like, for instance, the guards of the language in [dBP90a]). Some concurrent constraint languages (see, for instance, the languages in [SR90, dBP90c]) do not allow this. However, it is possible there to construct a statement of two consecutive actions performing the check for entailment (ask) and the state transforamation (tell), and we can prove that it has the same semantics $\mathcal{D}$ as if it were atomic.

An interpretation satisfying the above requirements is called *monotonic*.

In the rest of this section we present two full abstraction results. The first one concerns the full abstraction of O with respect to Obs_1 Next we consider the second observation

function Obs_2. For this, O is not fully abstract. We develop a more abstract compositional semantics $\mathcal{D}$ by applying certain closure operators to O, and we show that $\mathcal{D}$ is fully abstract with respect to Obs_2 both for complete and monotonic interpretations.

4.3 Full abstraction of O with respect to Obs_1 for complete interpretations

In this section we prove the full abstraction of O for complete interpretations.

Theorem 4.1 *For complete interpretations I, O is fully abstract with respect to Obs_1. That is, for all $s, t \in \mathcal{L}$,*

$$O[\![s]\!] = O[\![t]\!] \Leftrightarrow \forall C(\cdot),\ Obs_1[\![C(s)]\!] = Obs_1[\![C(t)]\!]$$

(Here $C(\cdot)$ is a unary context, which yields for each statement u a statement $C(u)$.)

Proof The arrow from left to right is immediate from the correctness of O with respect to Obs_1: consider $s, t \in \mathcal{L}$ with $O[\![s]\!] = O[\![t]\!]$, and let $C(\cdot)$ be an arbitrary context. Then

$$\begin{aligned} Obs_1[\![C(s)]\!] &= \alpha \circ O(C(s)) \\ &= \alpha \circ O(C(t)) \text{ (by the compositionality of } O \text{ and } O[\![s]\!] = O[\![t]\!]) \\ &= Obs_1[\![C(t)]\!] \end{aligned}$$

We prove the arrow from right to left by contradiction. So consider two statements $s, t \in \mathcal{L}$ such that $O[\![s]\!] \neq O[\![t]\!]$. We show the existence of a context $C(\cdot)$ with $Obs_1[\![C(s)]\!] \neq Obs_1[\![C(t)]\!]$. Assume that there exists $w \in O[\![s]\!] \setminus O[\![t]\!]$, say

$$w = (\sigma_1, \sigma_1'), \ldots, (\sigma_n, \sigma_n')$$

for some $n \geq 1$. (The case that the last pair is of the form (σ_n, δ) can be treated similarly.) Let the state μ_i, for $i = 1, \ldots, n-1$, be defined by

$$\mu_i = \mu(\sigma_i', \sigma_{i+1})$$

(The first requirement of complete interpretations ensures the existence of such states.) Next a context $C(\cdot) = \cdot \parallel u$ is defined by putting for the statement u

$$u = a(\sigma_1', \mu_1);\ a(\mu_1, \sigma_2);\ \cdots;\ a(\sigma_{n-1}', \mu_{n-1});\ a(\mu_{n-1}, \sigma_n)$$

(These actions exist by the second requirement on complete interpretations.) We show that $Obs_1[\![C(s)]\!] \neq Obs_1[\![C(t)]\!]$. Since $w \in O[\![s]\!]$ we have by the definition of O that

$$s \xrightarrow{(\sigma_1, \sigma_1')} \cdots \xrightarrow{(\sigma_n, \sigma_n')} E$$

Thus

$$s \parallel u \xrightarrow{(\sigma_1, \sigma_1')(\sigma_1', \mu_1)(\mu_1, \sigma_2)} \cdots \xrightarrow{(\mu_{n-1}, \sigma_n)(\sigma_n, \sigma_n')} E$$

Hence

$$\sigma_1' \mu_1 \sigma_2 \sigma_2' \cdots \mu_{n-1} \sigma_n \sigma_n' \in Obs_1[\![s \parallel u]\!](\sigma_1)$$

Suppose that also

$$\sigma'_1\mu_1\sigma_2\sigma'_2\cdots\mu_{n-1}\sigma_n\sigma'_n \in Obs_1[\![t \parallel u]\!](\sigma_1)$$

This would imply, by the definition of Obs_1, that

$$t \parallel u \xrightarrow{(\sigma_1,\sigma'_1)}\xrightarrow{(\sigma'_1,\mu_1)}\xrightarrow{(\mu_1,\sigma_2)} \cdots \xrightarrow{(\mu_{n-1},\sigma_n)}\xrightarrow{(\sigma_n,\sigma'_n)} E$$

Since for all i, (σ'_i,μ_i) and (μ_i,σ_i+1) do not occur in $O[\![t]\!]$, this implies

$$t \xrightarrow{(\sigma_1,\sigma'_1)} \cdots \xrightarrow{(\sigma_n,\sigma'_n)} E$$

Hence $w \in O[\![t]\!]$. Contradiction. □

4.4 A fully abstract semantics for state transitions

It is not difficult to see that $\mathcal{O}$ is not fully abstract with respect to Obs_2, in fact O encodes the states encountered step by step in the computation, (so, also the repetitions of the same state), while in Obs_2 only the transitions between two different states are visible. To obtain a fully abstract semantics we have therefore to abstract from "silent steps", namely those steps that do not cause any change in the state. We do so by saturating the semantics $\mathcal{O}$, closing it under the following conditions:

Definition 4.2 (Closure conditions) *Let X be a set of sequences of pairs of states.*

C1	$w_1 \cdot w_2 \in X$	$\Rightarrow$	$w_1 \cdot (\sigma,\sigma) \cdot w_2 \in X$	$(w_1 \neq \epsilon)$
C2	$w \in X$	$\Rightarrow$	$(\sigma,\sigma) \cdot w \in X$	$((\sigma,\delta) \notin X)$
C3	$w_1 \cdot (\sigma,\sigma) \cdot (\sigma,\sigma') \cdot w_2 \in X$	$\Rightarrow$	$w_1 \cdot (\sigma,\sigma') \cdot w_2 \in X$	
C4	$w_1 \cdot (\sigma,\sigma') \cdot (\sigma',\sigma') \cdot w_2 \in X$	$\Rightarrow$	$w_1 \cdot (\sigma,\sigma') \cdot w_2 \in X$	

Furthermore, let $Close(X)$ denote the smallest set containing X and closed under the conditions **C1**, **C2**, **C3** *and* **C4**.

The conditions **C1** and **C2** allow for the addition of silent steps, whereas **C3** and **C4** allow for the removal of silent steps. Note that we do not allow to add silent steps at the beginning of the sequence (σ',δ) because the processes δ and $\tau;\delta$ can be distinguished by a plus context (δ denotes the action which is nowhere defined and τ denotes the identity). Furthermore we do not allow the addition of (σ,σ) at the beginning of a sequence w in case $(\sigma,\delta) \in X$ because this would yield incorrect results with respect to a plus context: the deadlock possibility (σ,δ) would be removed because of the presence of $(\sigma,\sigma) \cdot w$.

Definition 4.3 *We define the semantics $\mathcal{D}$ as follows:* $\mathcal{D}[\![s]\!] = Close(\mathcal{O}[\![s]\!])$

First we note that $\mathcal{D}$ is correct:

Theorem 4.4 (Correctness of $\mathcal{D}$) $\alpha \circ \mathcal{D} = Obs_2$.

Proof

$$\alpha(\mathcal{D}[\![s]\!])(\sigma) = \alpha(Close(\mathcal{O}[\![s]\!]))(\sigma) = \alpha(\mathcal{O}[\![s]\!])(\sigma) = Obs_2[\![s]\!]$$

The crucial point is that the closure conditions do not give rise to additional connected sequences, which justifies the second equality. □

The following theorem states the compositionality of $\mathcal{D}$.

Theorem 4.5 (Compositionality of $\mathcal{D}$) *We have*

$$\begin{array}{lcl} \mathcal{D}[\![s_1; s_2]\!] & = & \mathit{Close}(\mathcal{D}[\![s_1]\!]; \mathcal{D}[\![s_2]\!]) \\ \mathcal{D}[\![s_1 + s_2]\!] & = & \mathcal{D}[\![s_1]\!] + \mathcal{D}[\![s_2]\!] \\ \mathcal{D}[\![s_1 \parallel s_2]\!] & = & \mathit{Close}(\mathcal{D}[\![s_1]\!] \parallel \mathcal{D}[\![s_2]\!]) \end{array}$$

Proof We treat the cases of the plus and the parallel operator, the case of the sequential operator being straightforward: First we treat the case of the plus operator:

$$\begin{array}{lcl} \mathcal{D}[\![s_1 + s_2]\!] & = & \mathit{Close}(\mathcal{O}[\![s_1 + s_2]\!]) \\ & = & \mathit{Close}(\mathit{rem}(\mathcal{O}[\![s_1]\!] \cup \mathcal{O}[\![s_2]\!])) \\ & = & \mathit{rem}(\mathit{Close}(\mathcal{O}[\![s_1]\!]) \cup \mathit{Close}(\mathcal{O}[\![s_2]\!])) \\ & = & \mathcal{D}[\![s_1]\!] + \mathcal{D}[\![s_2]\!] \end{array}$$

The first equation follows from the definition of $\mathcal{D}$, the second equation is justified by the compositionality of $\mathcal{O}$ and the definition of the interpretation of the plus operator. The third equation follows from the observation

$$\exists \sigma', w[(\sigma, \sigma') \cdot w \in X] \Leftrightarrow \exists \sigma', w[(\sigma, \sigma') \cdot w \in \mathit{Close}(X)]$$

for arbitrary σ and set of sequences X.

Next we treat the parallel operator:

$$\begin{array}{lcl} \mathcal{D}[\![s_1 \parallel s_2]\!] & = & \mathit{Close}(\mathcal{O}[\![s_1 \parallel s_2]\!]) \\ & = & \mathit{Close}(\mathcal{O}[\![s_1]\!] \parallel \mathcal{O}[\![s_2]\!]) \\ & = & \mathit{Close}(\mathit{Close}(\mathcal{O}[\![s_1]\!]) \parallel \mathit{Close}(\mathcal{O}[\![s_2]\!])) \\ & = & \mathit{Close}(\mathcal{D}[\![s_1]\!] \parallel \mathcal{D}[\![s_2]\!]) \end{array}$$

The first equation holds by definition of $\mathcal{D}$. For the second equation the compositionality of $\mathcal{O}$ is used. For the third equation it is sufficient to prove that for arbitrary sequences w, w' $\mathit{Close}(w) \parallel \mathit{Close}(w') \subseteq \mathit{Close}(w \parallel w')$, which is shown by an induction on the number of applications of the closure conditions. Finally, the last equation follows by definition of $\mathcal{D}$. □

The proof of the full abstraction of $\mathcal{D}$ with respect to Obs_2, for complete interpretations, essentially follows the same line of reasoning as presented above (for the full abstraction of O with respect to Obs_1).

For monotonic interpretations, we first notice that we can, without loss of generality, restrict the the domain of $\mathcal{O}$ and $\mathcal{D}$ to increasing sequences. In fact, since during a computation the state increases monotonically, only increasing seqences can be combined so to give connected results.

Next, we show that $\mathcal{D}$ satisfy the following additional closure property:

Proposition 4.6 *For monotonic interpretations we have*

$$\mathbf{C4} \quad w_1 \cdot (\sigma_1, \sigma_2) \cdot w_2 \in \mathcal{D}[\![s]\!] \;\Rightarrow\; w_1 \cdot (\sigma, \sigma) \cdot w_2 \in \mathcal{D}[\![s]\!]$$

for every s, with $\sigma_2 \sqsubseteq \sigma$.

Proof By the requirement of (strong) idempotency of atomic actions. □

We are now ready to prove the full abstraction of $\mathcal{D}$ for monotonic interpretations.

Theorem 4.7 *For monotonic interpretations I, $\mathcal{D}$ is fully abstract with respect to Obs_2. That is, for all $s, t \in \mathcal{L}$,*

$$\mathcal{D}[\![s]\!] = \mathcal{D}[\![t]\!] \Leftrightarrow \forall C(.),\ Obs_2[\![C(s)]\!] = Obs_2[\![C(t)]\!].$$

(Here $C(.)$ is a unary context, which yields for each statement u the statement $C(u)$.)

Proof The arrow from left to right is immediate from the correctness of $\mathcal{D}$ with respect to Obs_2 and the compositionality of $\mathcal{D}$.

The other direction we prove by contradiction: Let $w \in \mathcal{D}[\![s]\!] \backslash \mathcal{D}[\![t]\!]$. Let w' fill the gaps of w (so $w \parallel w'$ is a connected sequence), say $w' = (\sigma_0, \sigma_1), \ldots, (\sigma_{n-1}, \sigma_n)$. Let, for $1 \leq i \leq n$, a_i be such that $I(a_i)(\sigma_{i-1}) = \sigma_i$ and for $\sigma \sqsubseteq \sigma_{i-1}$ $(\sigma \neq \sigma_{i-1})$ $I(a_i)(\sigma)$ is undefined. Note that such actions exists according to the assumption on the expressiveness of I. Let $u = a_1; \ldots; a_n$. Note that $w' \in \mathcal{D}[\![u]\!]$, and thus by the correctness and compositionality of $\mathcal{D}$ we have $\alpha(w \parallel w')(\sigma) \in Obs_2[\![s \parallel u]\!](\sigma)$, where σ denotes the first state of w.

Assume that also $\alpha(w \parallel w')(\sigma) \in Obs_2[\![t \parallel u]\!](\sigma)$. We derive a contradiction. So there exist $w_0 \in \mathcal{D}[\![t]\!]$ and $w_1 \in \mathcal{D}[\![u]\!]$ such that $\alpha(w \parallel w')(\sigma) = \alpha(w_0 \parallel w_1)(\sigma)$.

First we show that $w_1 \in Close(w')$: By definition of $\mathcal{D}$ there exists $w'' \in \mathcal{O}[\![u]\!]$ such that $w_1 \in Close(w'')$. Thus it suffices to show that $w'' \in Close(w')$. From the definition of u and $\alpha(w \parallel w')(\sigma) = \alpha(w_0 \parallel w_1)$ it is not difficult to derive that w_1 and thus w'' does *not* end in deadlock. A straighforward induction on the length of w' then establishes that for $w'' \in \mathcal{O}[\![u]\!]$, not ending in deadlock, we have $w'' \in Close(u)$.

Finally, from $\alpha(w \parallel w')(\sigma) = \alpha(w_0 \parallel w_1)(\sigma)$ and by induction on the number of applications of the closure conditions **C1**, ..., **C4** in the derivation of w_1 from w', it is straightforward to prove that $w \in Close(w_0)$. By the definition of $\mathcal{D}$ it then follows that $w \in \mathcal{D}[\![t]\!]$, so contradicting our initial assumption. □

5 On the embedding of languages in $\mathcal{L}$

The language $\mathcal{L}$ is a general paradigm for asynchronous communication, and hence it is an interesting question to see whether other languages can be embedded into it. Probably the most interesting results are the negative ones: proving that some languages cannot be embedded into $\mathcal{L}$.

In this section, a general notion of embedding is discussed. An embedding has two components: a compiler and a decoder [Sha89]. Without any restrictions on these components, any programming language can be embedded into any other language: one can define a decoder which is strong enough to do the simulation itself. This is not what we want: the simulation should be performed at the level of the target language. In [dBP90a], a notion of embedding is provided which requires the compiler to be compositional and the decoder to preserve termination modes. We show that even under these conditions any language can be embedded into $\mathcal{L}$. Additionally, an analysis is given under what further conditions CCS cannot be embedded. One of them is requirement that basic actions are *extending*.

We start by giving a general definition of what it means to embed a language L into the language $\mathcal{L}$. We assume that there is an observation criterion for the language L, say a mapping Obs_L from L to some domain D_L. We then say that we can embed L into $\mathcal{L}$ if we can find a mappings $\mathcal{C}$ ("the compiler"), a mapping d ("the decoder") from the set of observables D_i of $\mathcal{L}$ to the set of observables D_L of L, and if we can make appropriate

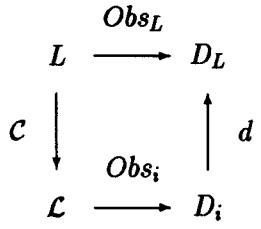

Figure 2: Embedding

choices for the set of states Σ and the interpretation function I such that the diagram of Figure 2 commutes. (In Figure 2 D_i stands for the domain reached by Obs_i, for example $D_3 = \Sigma \to \mathcal{P}(\Sigma_\delta)$.) Note that for different i and different observation criteria on the L level we can have different $\mathcal{C}$ and d. It might even be the case that for some criteria we can find an embedding and for other ones not.

In the rest of this section we restrict ourselves to the case of Obs_3.

A trivial embedding would be the following: map every statement of L to a different atomic action, and use in the interpretation I the function Obs_L. This is clearly to general. Every language could be embedded into $\mathcal{L}$.

The first restriction we make is that the translation $\mathcal{C}$ from L to $\mathcal{L}$ should be compostional and context free in the following sense:

> *Restriction* 1 **(Compositionality)**: Each atomic action $a \in A_L$ of L should be mapped to a statement $s \in \mathcal{L}$ and an operator of L should be associated with a context of $\mathcal{L}$, that is, if an operator has n operands, then it will be mapped to a context with n holes, and each of the holes is associated with one of the operands.

This first restriction, however, does not prevent us from encoding the syntax of L in the states of $\mathcal{L}$, as is shown in the following construction:

Take

$$A = \{op \ : \ op \in L\} \cup \{a \ : \ a \in A_L\}$$

$$\mathcal{C}(a) = a \quad \mathcal{C}(s_1 op \ s_2) = \mathcal{C}(s_1); op; \mathcal{C}(s_2)$$

$$\Sigma = \textit{prefixes of statements of } L$$

$$I(a)(\sigma) = \sigma a \quad I(op)(\sigma) = \sigma op$$

$$d(\sigma) = Obs_L(\sigma)$$

If the program is terminated, then we have coded the syntax of the L program in the final state and we can apply (as our decoder d) the Obs_L operator on this state.

If we look at this embedding, we observe that all the work is done in the decoder d. This, of course, does not fit with our intention: we want the simulation of the computations

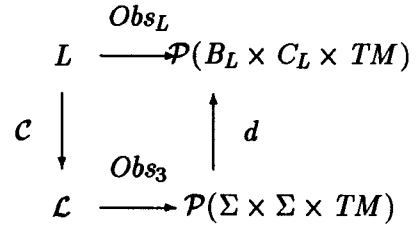

Figure 3: Embedding with termination modes

at the object level (L) to be accomplished by the target language ($\mathcal{L}$), while the decoder should serve just to fill the possible gaps at the level of the observables, due, for instance, to different formalisms.

Another point is that, in the simulation of concurrent languages, it seems natural to require the target system not to introduce additional deadlock possibilities that were not present in the original system.

In order to cope with these requirements we add the following restriction: a terminating computation on the L level should correspond to a terminating computation on the $\mathcal{L}$ level and the same applies to deadlocking and diverging computations.

Note that the construction above does not satisfy this requirement, since all the computation at the $\mathcal{L}$ level terminate. To formulate this restriction, we first observe that $\Sigma \to \mathcal{P}(\Sigma_\delta^*)$ is isomorphic to $\mathcal{P}(\Sigma \times \Sigma_\delta^*)$. In this way, we can view the domain as a set of computations. The set of *termination modes* is defined by $TM = \{\delta, \surd\}$ (where $\surd$ stands for normal termination). As a domain for the observables we take

$$\mathcal{P}(\Sigma \times \Sigma^* \times TM)$$

Also on the L level we should be able to take an isomorphic domain of the form

$$\mathcal{P}(B_L \times C_L \times TM)$$

for which B_L should be the initial observations (before the computation) and C_L should be what can be observed from the computation, given that the state of the system before execution satisfies B_L. Then the second restriction can be formulated as follows:

> *Restriction* 2 **(Preservation of the termination modes)**: The decoder d should be a mapping from computations on the $\mathcal{L}$ level (i.e., elements of $\Sigma \times \Sigma^* \times TM$) to the computations on the L level (i.e., elements of $B_L \times C_L \times TM$) that preserves termination modes.

If we add this restriction, then the construction above does not work anymore because on the $\mathcal{L}$ level every computation terminates.

Despite the restrictions, however, we are still able code a compostional model for L in the states and circumvent the idea that computations on the L level should be simulated on the $\mathcal{L}$ level. We give only an outline of this construction. We can build up in our states a compostional model for L. This can be done without violating the compostionality of $\mathcal{C}$.

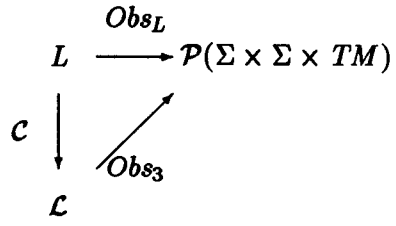

Figure 4: Embedding with restrictions on states

We also have to respect the termination mode. For this we make sufficiently many copies of each computation. At a certain moment we have built up in the state the complete model for the whole statement and then we decide what to do: to terminate or to deadlock (depending in which copy of the computation we are). In this way we get a lot of similar computations on the $\mathcal{L}$ level, which only differ in their last step (in which they decide which "path" they take and to terminate or to deadlock).

Therefore, it seems reasonable to put a restriction on the kind of states we allow.

Restriction 3. Take as states only those things we can observe, i.e.

$$\Sigma = C_L = B_L.$$

Moreover, we require d to be the identity.

Next we give an outline of the proof of a non-embeddability result. Consider a finite version of CCS, in which we have the following statements: (let $a \in A$, $c \in C$, $A \cap C = \emptyset$)

$$s ::= a \mid c \mid \bar{c} \mid s_1; s_2 \mid s_1 + s_2 \mid s_1 \parallel s_2$$

The important difference with the ACCS introduced before is that we have here synchronous communication. Actions a can move on their own, while both c and $\bar{c}$ have to find a communication partner $\bar{c}$ or c before they can proceed.

We take the following set of states:

$$\Sigma = (A \cup T)^* \cup (A \cup T)^* \cdot (C \cup \bar{C})$$

where

$$T = C \cdot \bar{C} \cup \bar{C} \cdot C$$

We do not use τ to denote the successful communication between two processes. Instead, pairs of actions representing the two participating communications are used. Moreover, in the case of deadlock, we do not deliver simply a δ but register the communication action for which no matching partner can be found. We assume that we have the standard mapping Obs_{CCS} from the language to $\mathcal{P}(\Sigma \times \Sigma \times TM)$.

If we put two more restrictions:

- The interpretation function I should be extending: For all σ and a we have $\sigma \leq I(a)(\sigma)$
- All computations on the $\mathcal{L}$ level should start from the empty word (the initial state is the empty word)

then we can prove the following theorem:

Theorem 5.1 *The CCS like language described above cannot be embedded in the language $\mathcal{L}$.*

The key lemma's of the proof are:

Lemma 5.2 *Let $s, t \in \mathcal{L}$. If we can make a sequence of connected transition steps from t in starting in state σ then we can make the same sequence of steps from t; s, $t \parallel s$, $s \parallel t$, $t + s$ and $s + t$ starting in state σ.*

A transition sequence is called complete if it ends in the empty statement E.

Lemma 5.3 *Let $s \in \mathcal{L}$. Let γ be a complete connected transition sequence from $C(s)$ ($C(\cdot)$ is a context). If in γ one transition is caused by the component s, then there is a subsequence of γ of which the labels correspond to the labels of a complete transition sequence from s.*

The proof sketch proceeds as follows. First we show that if we want to check if the compiled version of statement s deadlocks, then we have to make with our $\mathcal{L}$ transition system at least two steps. Hence if we want to give a context for $+$ on the $\mathcal{L}$ level, this context needs to check if two steps are possible of its components. We are then, by the lemma above, forced to execute the whole statement, and this leads to a contradiction.

Intuitively, if we want to implement synchronous communication in $\mathcal{L}$, we need at least a two step procedure: a send and an acknowledgement. In order to make the right choices for the translated $+$ we need a plus2 context: we can choose one of the sides if we can do two steps of a side. This context cannot be made in $\mathcal{L}$.

6 Future work

As we shown in Section 3, the equivalence induced by the compositional semantics for asynchronous processes is coarser than the equivalence associated to failure set semantics, and, a fortiori, the equivalences associated to the various notions of bisimulation. It would be interesting to investigate if it is possible to give an axiomatic characterization of this equivalence, and in this case, which axioms must be added to the standard ones of the process algebra so to obtain it.

Another interesting topic is the extension of our approach so to cope with distributed systems (for which, the notion of global state is not adequate), and, possibly, to model *true concurrency* and related notions of observables.

Finally, the issue of embedding seems to be quite promising. The content of Section 5 of this paper is not yet conclusive and we hope to return to this subject in a future paper.

References

[BC89] A. Brogi and P. Ciancarini. The concurrent language Shared Prolog. Technical Report TR-8/89, Dipartimento di Informatica, Università di Pisa, 1989. To be published in *ACM TOPLAS*.

[BHR84] S.D. Brookes, C.A.R. Hoare, and W. Roscoe. A theory of communicating sequential processes. *J. Assoc. Comput. Mach.*, 31:560–599, 1984.

[BKO88] J.A. Bergstra, J.W. Klop, and E.-R. Olderog. Readies and failures in the algebra of communicating systems. *SIAM J. Comp.*, 17(6):1134–1177, 1988.

[dBK90] J.W. de Bakker and J.N. Kok. Comparative metric semantics for Concurrent Prolog. *Theoretical Computer Science*, 75:15–43, 1990.

[dBP90a] 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.

[dBP90b] F.S. de Boer and C. Palamidessi. On the asynchronous nature of communication in concurrent logic languages: A fully abstract model based on sequences. In J.C.M. Baeten and J.W. Klop, editors, *Proc. of Concur 90*, volume 458 of *Lecture Notes in Computer Science*, pages 99–114, The Netherlands, 1990. Springer-Verlag. Full version available as report at the Technische Universiteit Eindhoven.

[dBP90c] F.S. de Boer and C. Palamidessi. A fully abstract model for concurrent constraint languages. *Proc. of TAPSOFT 91*, 1991. Also available as technical report, Centre for Mathematics and Computer Science (CWI), Amsterdam.

[dBZ82] J.W. de Bakker and J.I. Zucker. Processes and the denotational semantics of concurrency. *Information and Control*, 54:70–120, 1982.

[EHLR80] D. Erman, F. HayesRoth, V. Lesser, and D. Reddy. The Hearsay2 speech understanding system: Integrating knowledge to resolve uncertainty. *ACM Computing Surveys*, 12:213–253, 1980.

[GCLS88] R. Gerth, M. Codish, Y. Lichtenstein, and E. Shapiro. Fully abstract denotational semantics for Concurrent Prolog. In *Proc. of the Third IEEE Symposium on Logic In Computer Science*, pages 320–335. IEEE Computer Society Press, New York, 1988.

[Gel86] D. Gelenter. Generative communication in Linda. *ACM TOPLAS*, 7(1):80–112, 1986.

[GMS89] H. Gaifman, M. J. Maher, and E. Shapiro. Reactive Behaviour semantics for Concurrent Constraint Logic Programs. In E. Lusk and R. Overbeck, editors, *North American Conference on Logic Programming*, 1989.

[HdBR90] E. Horita, J.W. de Bakker, and J.J.M.M. Rutten. Fully abstract denotational models for nonuniform concurrent languages. Technical Report CS-R9027, Centre for Mathematics and Computer Science, Amsterdam, 1990.

[Hoa78] C.A.R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, 1978.

[JHJ90] M.B. Josephs, C.A.R. Hoare, and He Jifeng. A theory of asynchronous processes. Technical report, Oxford University Computing Laboratories, 1990.

[JJH90] 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.

[JK89] B. Jonsson and J.N. Kok. Comparing two fully abstract dataflow models. In *Proc. Parallel Architectures and Languages Europe (PARLE)*, volume 379 in Lecture Notes in Computer Science, pages 217–235, 1989.

[Jon85] B. Jonsson. A model and a proof system for asynchronous processes. In *Proc. of the 4th ACM Symp. on Principles of Distributed Computing*, pages 49–58, 1985.

[Kah74] G. Kahn. The semantics of a simple language for parallel programming. In *Information Processing 74: Proc. of IFIP Congress*, pages 471–475, New York, 1974. North-Holland.

[Kok87] J.N. Kok. A fully abstract semantics for data flow nets. In J.W. de Bakker, A.J. Nijman, and P.C. Treleaven, editors, *Proc. Parallel Architectures and Languages Europe (PARLE)*, volume 259 of *Lecture Notes in Computer Science*, pages 351–368. Springer Verlag, 1987.

[Kok89] J.N. Kok. Traces, histories and streams in the semantics of nondeterministic dataflow. In *Proceedings Massive Parallellism: Hardware, Programming and Applications*, 1989. Also available as report 91, Abo Akademi, Finland, 1989.

[Mil80] R. Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer-Verlag, New York, 1980.

[Mil83] Robin Milner. Calculi for synchrony and asynchrony. *Theoretical Computer Science*, 25:267–310, 1983.

[Sar89] V.A. Saraswat. *Concurrent Constraint Programming Languages*. PhD thesis, january 1989. To be published by the MIT Press.

[Sha83] E.Y. Shapiro. A subset of concurrent prolog and its interpreter. Technical Report TR-003, ICOT, 1983.

[Sha89] E.Y. Shapiro. The family of concurrent logic programming languages. *ACM Computing Surveys*, 21(3):412–510, 1989.

[SR90] 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.