1991

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

# The failure of failures in a paradigm for asynchronous communication

Computer Science/Department of Software Technology Report CS-R9147 October

# The Failure of Failures in 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]CWI,
Kruislaan 413, 1098 SJ Amsterdam, The Netherlands

e.mail: {frb, joost, katuscia, janr}@cwi.nl

**Abstract**

We develop a general framework for a variety of concurrent languages all based on asynchronous communication, like data flow, concurrent logic, concurrent constraint languages and CSP with asynchronous channels. The main characteristic of these languages is that processes interact 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 partially defined transformations on an abstract set of states. Suspension is modelled by an action being undefined in a state. The languages listed above can be seen as instances of our paradigm, and can be 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 in the style of Plotkin's SOS. A compositional model is presented that is based on traces (of pairs of states). This sharply contrasts with the synchronous case, where some additional branching information is needed to describe deadlock. In particular, we show that our model is more abstract than the standard failure set semantics (that is known to be fully abstract for the classical synchronous paradigms).

We also investigate the problem of full abstraction, with respect to various observation criteria. To tackle this problem, we have to consider the particular features of the specific languages. We study some cases, for which we give a fully abstract semantics.

*1985 Mathematics Subject Classification*: 68Q10, 68Q45.
*1986 Computing Reviews Categories*: D.3.1, F.3.2, F.3.3.
*Key Words and Phrases*: Paralellism, asynchrony, compositionality, labelled transition system, constraint languages, dataflow.
*Note*: 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

Report CS-R9147
ISSN 0169-118X
CWI
P.O. Box 4079, 1009 AB Amsterdam, The Netherlands

Basic Research Action SPEC. It appeared in the proceedings of CONCUR '91, LNCS 527.

## 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 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 this 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 in the style of Plotkin's SOS. 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 main results of this paper is a compositional characterization of these notions of observables, which is defined independently of the particular choice for the sets of actions and the set of states. 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.

The compositional model for our paradigm is so simple that it might look trivial. We should like to point out that it is not. Its interest is formed by the mere fact that it is as simple as it is. In particular, its definition is based on traces and does not need additional structures like failure sets, which are needed for describing deadlock in the case of synchronously communicating processes. We show that our model is more abstract than the classical failure set semantics, and for that reason more suitable for describing asynchronous communication.

Another contribution of the paper is the identification of two classes of interpretations, for which a full-abstraction result is obtained. 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*.

For complete interpretations it is shown that our model is fully abstract with respect to a notion of observables consisting of (sets of) sequences of states. For monotonic interpretations, full abstraction is considered with respect to another notion of observables, which abstracts from state repetitions ("finite stuttering"). This case is more complicated. The basic compositional model is not fully abstract. It is shown that a fully abstract model can be obtained by applying an abstraction operator to the first model. This abstraction consists of taking the closure of sets of sequences under a number of conditions. Interestingly, these closure conditions are reminiscent of the way one can view rooted tau-bisimulation [BK88] (or weak observational congruence, in the vocabulary of [Mil80]) as ordinary strong bisimulation, by adding certain rules to the transition system. In the present situation, a subtle refinement of that approach is required, due to the non-uniform nature of our models.

## 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 a uniform framework, nor for providing the appropriate 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, dBP91] 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 (both for the success and for the deadlock case). The paradigm and the semantics presented here can be seen as a generalization of this approach, abstracting from the specific details related to concurrent logic and constraint languages, and showing that it can be applied to a much wider range of languages. In fact, the models of [dBP90b, dBP91] can be obtained as special instances of our language: see Example 2 of Section 2.

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, dBP91, 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. Section 5 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

$$I : \ A \rightarrow \Sigma \rightarrow_{partial} \Sigma$$

that maps atomic actions to partially defined state transformations. If $I(a)(\sigma)$ is undefined, the action is blocked in the current state $\sigma$. This need not neccesarily lead to a global deadlock of the whole system, because some other component of the program may be enabled to take a next step. Note that therefore the operator plus models global nondeterminism rather than local choice: the choice can be influenced by the environment.

The computational model of $\mathcal{L}$ is described by a *labelled transition system* $(\mathcal{L}, \mathit{Label}, \rightarrow)$. The set $(\lambda \in)\mathit{Label}$ of labels is defined by $\mathit{Label} = \Sigma \times \Sigma$. A label represents the state transformation caused by the action performed during the transition step.

Instead of pairs of states, one could also take functions from $\Sigma$ to $\Sigma$ as labels. We prefer to use pairs of states because in that way a semantics is obtained that is more abstract. This is due to the fact that using sets of (sequences of) pairs will yield more identifications of statements than using sets of (sequences of) functions. (The reader is invited to find a simple example.)

The transition relation $\rightarrow \subseteq \mathcal{L} \times \mathit{Label} \times \mathcal{L}$ 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. Given an initial state $\sigma$, the first observation criterion we consider, $\mathit{Obs}_1$, 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

[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]).

which the computation goes. (Such a sequence may end in $\delta$, 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 $\sigma_0 \cdot \sigma_1 \cdots \sigma_n \in Obs_1[\![s]\!](\sigma_0)$. 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.

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 $\delta$ for deadlocking computations.

**Definition 2.1** *Let $\Sigma^+$ denote the set of all finite non-empty sequences of states. We put $\Sigma^+_\delta = \Sigma^+ \cup \Sigma^+ \cdot \{\delta\}$ and $\Sigma_\delta = \Sigma \cup \{\delta\}$. Let $\mathcal{P}(\cdot)$ be the set of subsets of $(\cdot)$. 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 $i = 1, 2$,*

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

*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 $Obs_3 = \lambda s.\lambda\sigma.\gamma(Obs_1[\![s]\!](\sigma))$, 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 $A$, $\Sigma$ and $I$. We present a list of five examples.

**Example 1: An imperative language**
Let $A$ be the set of assignments $x := e$, where $x \in \mathit{Var}$ is a variable and $e \in \mathit{Exp}$ is an expression. Assume that the evaluation $\mathcal{E}(e)(\sigma)$ of expression $e$ in state $\sigma$ is simple in that it does not have side effects and is instantaneous. Let the set of states be defined by

$$\Sigma = \mathit{Var} \rightarrow \mathit{Val}$$

where $\mathit{Val}$ is some abstract set of values. Then define $I$ by

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

With this choice for $A$, $\Sigma$ and $I$, the models $\mathit{Obs}_1$ 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:

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

**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., $A = \{\mathit{ask}(\vartheta), \mathit{tell}(\vartheta) : \vartheta \in L\}$. 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 $\bar{a}$ action 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:

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

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

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

Note that a 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** *First a semantic domain* $(X, Y \in)P$ *is introduced. It is defined by*

$$P = \mathcal{P}_{\mathrm{ne}}(Q) \quad \textit{(non-empty subsets of } Q\textit{)}$$

$$Q = (\Sigma\times\Sigma)^* \cup (\Sigma\times\Sigma)^*\cdot(\Sigma\times\{\delta\})$$

*Next* $O : \mathcal{L} \to P$ *is defined 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 $(\sigma,\sigma')$ is that if the current state is $\sigma$, then the computation at hand can transform this state into $\sigma'$. 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 $\sigma_1'$ may be different from $\sigma_2$.

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.*

- $X_1;\,X_2 = \bigcup\{w_1\hat{;}\,w_2 :\ w_1 \in X_1 \wedge w_2 \in X_2\}$, *where* $\hat{;}\ : Q\times Q \to P$ *is defined by*

$$w_1\hat{;}\,w_2 = \begin{cases} \{w_1\} & \textit{if } w_1 = w\cdot(\sigma,\delta) \\ \{w_1\cdot w_2\} & \textit{otherwise} \end{cases}$$

- $X_1 + X_2 = ((X_1\cup X_2)\cap((\Sigma\times\Sigma)\cdot Q))\ \cup\ (X_1\cap X_2\cap(\Sigma\times\{\delta\}))$

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

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

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

The definition of the sequential composition ; is straightforward.

The definition of the choice operator $+$ is slightly more intricate. The value of $X_1 + X_2$ consists of all sequences of both sets that do not start with a deadlock pair (i.e., a pair of which the second element is $\delta$), together with those pairs $(\sigma, \delta)$ that occur in both sets. This is motivated by the fact that operationally, the nondeterministic composition of two statements will yield a deadlock only if both statements yield deadlock separately.

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: $w_1 \mathbin{|\!\lfloor} w_2$ and $w_2 \mathbin{|\!\lfloor} w_1$. In $w_1 \mathbin{|\!\lfloor} w_2$, every word starts with a step from the left component $w_1$. This definition of the merge ensures that deadlocking steps are delayed as long as possible. Thus we have, for instance,

$$\{(\sigma, \sigma)\} \parallel \{(\sigma, \delta)\} = \{(\sigma, \sigma) \cdot (\sigma, \delta)\}$$

Now we are ready for the following theorem.

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

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

Finally, it is shown that $O$ is *correct* with respect to the observation functions $\mathit{Obs}_i$, 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 show that $O$ is correct with respect to $\mathit{Obs}_1$, from which the other two cases will follow. The relation between $\mathit{Obs}_1$ and $O$ can be made precise using the following abstraction operator. Let $\alpha : P \rightarrow \Sigma \rightarrow \mathcal{P}(\Sigma_\delta^+)$ be defined by

$$\alpha(X)(\sigma) = \bigcup \{\alpha(x)(\sigma) : x \in X\}$$

where

$$\alpha(x)(\sigma) = \begin{cases} \{\sigma\sigma_1 \cdots \sigma_n\} & \textit{if } x = (\sigma, \sigma_1)(\sigma_1, \sigma_2) \cdots (\sigma_{n-1}, \sigma_n) \\ \{\sigma\sigma_1 \cdots \sigma_{n-1}\delta\} & \textit{if } x = (\sigma, \sigma_1)(\sigma_1, \sigma_2) \cdots (\sigma_{n-1}, \delta) \\ \emptyset & \textit{otherwise} \end{cases}$$

The operator $\alpha$ selects from a set $X$, given an initial state $\sigma$, all connected sequences starting with $\sigma$. (It should be noted that also this operator is defined "element-wise".)

**Theorem 2.5 (Correctness of $O$)** $\alpha \circ O = \mathit{Obs}_1$

As a corollary, the correctness of $\mathit{Obs}_2$ and $\mathit{Obs}_3$ is immediate:

$$\mathit{Obs}_2 = \beta \circ \mathit{Obs}_1 = \beta \circ \alpha \circ O$$

$$\mathit{Obs}_3 = \gamma \circ \mathit{Obs}_1 = \gamma \circ \alpha \circ O$$

## 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 $\mathcal{O}$. 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 ($\mathcal{O}$) 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 $\mathcal{O}$ based on streams of pairs of states that can possibly end in a pair $(\sigma, \delta)$. The correspondence with the failure set semantics (containing only one element) is given by interpreting such a pair $(\sigma, \delta)$ at the end of a stream as a failure set with one element, $\sigma$. If the environment offers the state $\sigma$, then the statement will deadlock.

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

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

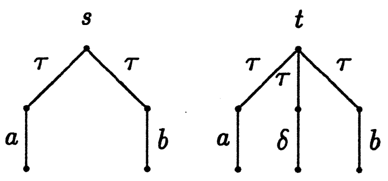

Figure 1: In case of asynchronous communication, $s$ and $t$ cannot be distinguished by any context.

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

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 $(2,2)$):

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

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

(Note that elements like $(*,*)\emptyset$ are present in *failure*$(s)$ and *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 the component $\tau;\, \delta$ in $t$, which does not occur in $s$: This explains the presence of $(*,*)\{1,2\}$ in *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 form of $(*,*)(1,1)$ and $(*,*)(2,2)$. Therefore $s$ and $t$ need not be distinguished. As we can see, $\mathcal{O}$ does not distinguish between the two statements:

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

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 $\mathcal{O}$: 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 $\mathcal{O}$, 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 $\sigma$ then the action $a(\sigma, \sigma')$ changes it to $\sigma'$; 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 $(\sigma, \mu)$ and $(\mu, \sigma')$ 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 instantaneously 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 $y \in \mathit{Var}$ 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 $v \in \mathit{Val}$ different from $\sigma(y)$ and $\sigma'(y)$. (Both $\mathit{Var}$ and $\mathit{Val}$ 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 $(\sigma, \tau)$ and $(\tau, \sigma')$ 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 modelled by implication, also the successful execution of ask depends monotonically on the store, i.e., if a certain ask is defined on $\sigma$, it will be defined in all the constraints greater than $\sigma$. Both ask and tell are *extending*, i.e. their action on the store can only increase the store. Another characteristic is that both ask and tell are *strongly idempotent*: if their execution results in a certain store $\sigma$, to execute them again in the same store $\sigma$, or in a greater store, will have no effect. These properties easily derive from the definition of ask and tell (see Section 2). Notice that a consequence of them is that the store will monotonically increase during the computation.

We now formalize these features in our framework. Given a set of states $\Sigma$, where a state represents a store (constraint), and a partial order $\sqsubseteq$ on $\Sigma$, where $\sigma \sqsubseteq \sigma'$ should be read as "$\sigma'$ encodes more information than (or the same as) $\sigma$", 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 $\sigma$ and $\sigma'$. (Note that these notions are independent.)

Furthermore, for the construction of the fully 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, dBP91]) 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 transformation (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, two full abstraction results are presented. The first one concerns the full abstraction of $\mathcal{O}$ with respect to $Obs_1$. Next we consider the second observation function $Obs_2$. For this, $\mathcal{O}$ is not fully abstract. We develop a more abstract compositional semantics $\mathcal{D}$ by applying certain closure operators to $\mathcal{O}$, 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 $\mathcal{O}$ with respect to $Obs_1$ for complete interpretations

In this section we state the full abstraction of $\mathcal{O}$ for complete interpretations.

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

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

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

## 4.4 Full abstraction with respect to $Obs_2$

It is not difficult to see that $O$ 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 $O$, closing it under the following conditions:

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

$$\begin{array}{llcll}
\textbf{C1} & w_1 \cdot w_2 \in X & \Rightarrow & w_1 \cdot (\sigma,\sigma) \cdot w_2 \in X & (w_1 \neq \epsilon) \\
\textbf{C2} & w \in X & \Rightarrow & (\sigma,\sigma) \cdot w \in X & ((\sigma,\delta) \notin X) \\
\textbf{C3} & w_1 \cdot (\sigma,\sigma) \cdot (\sigma,\sigma') \cdot w_2 \in X & \Rightarrow & w_1 \cdot (\sigma,\sigma') \cdot w_2 \in X & \\
\textbf{C4} & w_1 \cdot (\sigma,\sigma') \cdot (\sigma',\sigma') \cdot w_2 \in X & \Rightarrow & w_1 \cdot (\sigma,\sigma') \cdot w_2 \in X & \\
\textbf{C5} & w_1 \cdot (\sigma,\sigma) \cdot (\sigma,\delta) \in X & \Rightarrow & w_1 \cdot (\sigma,\delta) \in X & (w_1 \neq \epsilon)
\end{array}$$

*Furthermore, let $Close(X)$ denote the smallest set containing $X$ and closed under the conditions* **C1**, **C2**, **C3**, **C4** *and* **C5**.

The conditions **C1** and **C2** allow for the addition of silent steps, whereas **C3**, **C4** and **C5** allow for the removal of silent steps. Note that we do not allow the addition of $(\sigma,\sigma)$ 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 $(\sigma,\delta)$ would be removed because of the presence of $(\sigma,\sigma) \cdot w$.

It is interesting to note that it is possible to express these closure conditions inside the transition system itself. For instance, the counterpart for **C3** would be a transition rule of the form

$$\text{If } s \xrightarrow{(\sigma,\sigma)} s' \text{ and } s' \xrightarrow{(\sigma,\sigma')} s'' \text{ then } s \xrightarrow{(\sigma,\sigma')} s''$$

This is reminiscent of the way one can mimic rooted tau-bisimulation [BK88] (or weak observational congruence, in the vocabulary of [Mil80]) with ordinary strong bisimulation by adding rules like, for instance,

$$\text{If } s \xrightarrow{\tau} s' \text{ and } s' \xrightarrow{a} s'' \text{ then } s \xrightarrow{a} s''$$

(See [BK88, vG87].) The present case is by its non-uniform nature essentially more intricate. We intend to investigate this point further in the future.

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

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

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

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

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

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

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 $O$ and $\mathcal{D}$ to increasing sequences. In fact, since during a computation the state increases monotonically, only increasing sequences can be combined so to give connected results.

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

**Proposition 4.6** *For monotonic interpretations we have*

$$\textbf{C6} \quad w_1 \cdot (\sigma_1, \sigma_2) \cdot w_2 \in \mathcal{D}[\![s]\!] \quad \Rightarrow \quad w_1 \cdot (\sigma, \sigma) \cdot w_2 \in \mathcal{D}[\![s]\!]$$

*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 $C(u)$.)*

## 5 Future work

As was shown in Section 3, the equivalence induced by the compositional semantics for asynchronous processes is coarser than the equivalence associated with 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.

With respect to the issue of full abstraction, a number of questions remain to be answered. For instance, we have not yet investigated full abstraction for $Obs_3$, in which only the initial and final states are visible. For complete interpretations, this is expected to be an easy extension; for monotonic ones, some new closure conditions will have to be applied.

Our language $\mathcal{L}$ does not have a construct for recursion; this should be added. We should also like to investigate the extension of the language with an operator for hiding in combination with the possibility of distinguishing between local and global states.

**Acknowledgements** We would like to thank Jan-Friso Groote, Peter Knijnenburg, Alban Ponse, Fer-Jan de Vries and Jeroen Warmerdam for their detailed comments on preliminary versions of this paper. We thank the members of the Amsterdam Concurrency Group, including Jaco de Bakker, Franck van Breughel, Arie de Bruin, Peter Knijnenburg, Erik de Vink and Jeroen Warmerdam for helpful comments and stimulating discussions.

## References

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

[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.

[BK88] J.A. Bergstra and J.W. Klop. A complete inference system for regular processes with silent moves. In F.R. Drake and J.K. Truss, editors, *Proceedings Logic Colloquium 1986*, pages 21–81, Hull, 1988. North-Holland.

[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.

[dBP91] F.S. de Boer and C. Palamidessi. A fully abstract model for concurrent constraint programming. In S. Abramsky and T.S.E. Maibaum, editors, *Proc. of TAPSOFT/CAAP*, volume 493 of *Lecture Notes in Computer Science*. Springer-Verlag, 1991. Revised and Extended version in Technical Report CS-9110, Department of Computer Science, University of Utrecht, The Netherlands.

[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. To appear in Information and Computation.

[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)*, number 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.

[vG87] R.J. van Glabbeek. Bounded nondeterminism and the approximation induction principle in process algebra. In F.J. Brandenburg, G. Vidal-Naquet, and M. Wirsing, editors, *Proceedings STACS 1987*, volume 247 of *Lecture Notes in Computer Science*, pages 336–347. Springer-Verlag, 1987.