A Paradigm for Asynchronous Communication and its Application to Concurrent Constraint Programming

Frank S. de Boer * *Joost N. Kok* † *Catuscia Palamidessi* ‡
Jan J.M.M. Rutten §

Abstract

We develop a general semantic theory of asynchronous communication in concurrent logic and concurrent constraint languages. The main characteristic of these languages, from the point of view of the communication mechanism, is that processes interact by querying and updating some common data structure. We abstract from the specific features of the underlying data structure by means of a uniform language of which the actions are interpreted as transformations on an abstract set of states. This approach shows that there exists a basic similarity between concurrent logic (constraint) languages and other languages based on asynchrononous communication, like dataflow and asynchronous CSP. Actually, our intention is to capture languages based on asynchronous communication as instances of this uniform language, such an instance being determined by 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 reactive sequences, i.e., sequences of pairs of states.

1 Introduction

In this chapter, we propose a general semantic theory of asynchronous communication as present in concurrent constraint and concurrent logic languages. These languages have in common that processes communicate via some shared data structure. In the case of concurrent constraint languages the data structure is given by the underlying constraint system, while in concurrent logic languages it is given by the bindings established on the logical variables. However, in both cases the data structure is updated by means of operations which have free access (provided some consistency requirements are met) whereas it is queried by operations that may suspend in case the data structure does not entail the required information. The asynchronous nature of the communication stems from the independency of the update and query operations in the sense that they can take place at different times. This marks an essential difference with languages like CSP [Hoa78], where processes communicate by means of a "handshaking" mechanism, i.e., the simultaneous execution of (complementary) actions.

Our paradigm consists of a concurrent language $\mathcal{L}$ which assumes as 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. Furthermore we assume

*Department of Mathematics and Computing Science, Eindhoven University of Technology, P.O. Box 513, 5600 MB Eindhoven, The Netherlands. Email: frb@info.win.tue.nl.

†Department of Computer Science, Utrecht University, P.O. Box 80089, 3508 TB Utrecht, The Netherlands. Email: joost@cs.ruu.nl.

‡Department of Computer Science, Utrecht University, P.O. Box 80089, 3508 TB Utrecht, The Netherlands and Department of Software Technology, Centre for Mathematics and Computer Science, P.O. Box 4079, 1098 SJ Amsterdam, the Netherlands. Email: katuscia@cwi.nl.

§Department of Software Technology, Centre for Mathematics and Computer Science, P.O. Box 4079, 1098 SJ Amsterdam, the Netherlands. Email: Jan.Rutten@cwi.nl.

given an abstract set of states, which includes some special states indicating suspension and failure. The basic actions are interpreted as state transformations. A pure query action (like, e.g., a test) will have the property that it will suspend in those states that do not contain the required information (in those cases a special state representing suspension is delivered). On the other hand, for those states which do contain the required information a query action succeeds; the state remains unaltered. A suspended process is forced to wait until actions of other processes produce a state in which it is enabled. A pure update action is characterized by the fact that it will never suspend. In case the result of the update gives rise to an inconsistency, a special state representing failure is delivered. In general, an action can embody both an update and a query component.

The concurrent constraint languages [Sar89] are modeled 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 as instances of $\mathcal{L}$ by interpreting the proper states (excluding suspension and failure) as the bindings established on the logical variables, and the actions as the unification steps (see [dBK90]).

Apart from the concurrent constraint languages, many languages for asynchronously communicating processes can be obtained as instances of $\mathcal{L}$ by choosing the appropriate set of actions, the set of (proper) states and the interpretation of the basic actions. For example, the imperative language described in [HdBR90], based on shared variables, can be modeled 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. An asynchronous version of CSP, where processes communicate via asynchronous channels (see [JJH90]), can be obtained by taking as states the configurations of the channels and as actions the input-output primitives on these channels. Other interesting instances of $\mathcal{L}$ are IO-automata and data flow.

The basic computation model of the language $\mathcal{L}$ is described by means of a labeled 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 chapter 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, unifying the semantic work done in such apparently diverse fields as concurrent logic programming, data flow, and imperative programming based on asynchronous communication.

The most striking feature of our compositional semantics is that it is based on reactive sequences, i.e., sequences of pairs of states. A pair encodes the state transformation caused by 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 immediately following pair. These "gaps" represent, in a sense, the possible steps made by the environment. Thus such a sequence describes the behaviour of a process in terms of its interaction with the environment. Since there is no way to synchronize on actions, the behaviour of a process solely depends upon the current state. Therefore, such a set of sequences encodes all the information necessary for a compositional semantics.

The naturalness of the compositional model for our paradigm derives from the fact that its definition is based on, essentially, sequences, and does not need additional structures like failure sets, which are needed for describing deadlock in the case of synchronously communicating processes. In [dBKPR91] we showed that our model is more abstract than the classical failure set semantics, and for that reason more suitable for describing asynchronous communication.

1.1 Comparison with related work

In spite of the general interest for asynchronous communication as a natural mechanism for concurrency in many different programming styles, not much work has been done in the past, neither for defining a uniform framework, nor for providing the appropriate semantic 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 [GMS89] the semantics of FCP is defined by using an adaptation of 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 [dBP91] it has been shown that for concurrent logic and constraint languages reactive 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 the field of data flow, compositional models based on an appropriate notion of sequences called *quiescent traces* have been developed (for example in [Jon85]) and have been shown to be fully abstract. For an overview consult [Kok89]. Related languages (receptive processes communicating via asynchronous channels) have been recently studied in [JHJ90] and [Jos90]. In these papers as well, the authors propose a semantics which is based on sequences of input-output events.

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

The main contribution of both this chapter and [dBKPR91] is the generalization of the results obtained in [dBP90, dBP91, JHJ90, HdBR90] to a general language for asynchronous communication, in this way providing a uniform framework for reasoning about this kind of concurrency. However, in this chapter we elaborate in more detail on how one can obtain various languages as instances of this general paradigm. Specifically, we emphasize the case of concurrent constraint languages (the concurrent logic languages are viewed as instances of these), and that of IO-automata. Additionally, we give a characterization in terms of transition rules for a notion of observables which abstracts from finite stuttering.

1.2 Plan of the chapter

In the next section we start by presenting the paradigm $\mathcal{L}$. The syntax and the computational model (characterizing the observational behaviour) of the language are given. Additionally, we address the problem of a compositional characterization of a notion of observables which involves abstraction from finite stuttering. In Section 3, a detailed description is given of an application of this general theory to the concurrent logic and concurrent constraint languages. Section 4 illustrates how various other formalisms for asynchronous processes, notably IO-automata, can be obtained as instances of $\mathcal{L}$ as well. The last section briefly sketches some future research.

2 The paradigm and its semantics

We shall introduce a simple programming language $\mathcal{L}$, which constitutes the announced paradigm for asynchronous communication. The terminology is motivated by the fact that $\mathcal{L}$ represents an entire family of different languages, parameterized by a set of *atomic actions*. These languages have in common that the basic mechanism for interaction is based on querying and updating a shared data structure (the *state*). Each particular choice for the set of actions yields a specific language, also called an instance of $\mathcal{L}$. Various examples of such instances will be presented in the next two sections.

After having given the syntax of $\mathcal{L}$, an extensive description of two semantic models (operational and compositional (denotational)) is given. Next variations on these models will be discussed dealing with the abstraction from so-called *finite stuttering*.

2.1 The language $\mathcal{L}$

Let $(a \in)A$ be an arbitrary set, the elements of which are called *atomic* (or *basic*) *actions*. (Note that here and in the sequel we use $(x \in)X$ for introducing at the same time a set X and a special element x ranging over X.) We define the set $(s \in)\mathcal{L}$ of statements as follows:

$$s ::= a \mid s_1; s_2 \mid s_1 + s_2 \mid s_1 \parallel s_2$$

Moreover, $\mathcal{L}$ contains a special element E, the terminated statement.

An atomic action a can perform one computation step, in which it may change the state of the system. (The set of states will be introduced shortly.) The *sequential* composition $s_1; s_2$ is executed by first performing s_1 and next s_2. The execution of the *nondeterministic* choice $s_1 + s_2$ between the statements s_1 and s_2 amounts to executing either s_1 or s_2. It will be global in the sense that such a choice can be influenced by the activity of the environment. The *parallel* composition $s_1 \parallel s_2$ of s_1 and s_2 is executed by interleaving computation steps from both components. We do not include any constructs for recursion for the sake of simplicity. The main results that follow can be extended to cover also infinite behavior[1].

2.2 The operational model

The operational model will be defined in three stages. First the meaning of atomic actions is given, next a transition system for $\mathcal{L}$ is defined, and finally a notion of observables is derived from the transition system.

The actions of our language will be interpreted as transformations on a set of abstract *states*.

Definition 2.1 *Let $(\sigma \in)\ \Sigma$ be a set of abstract states. Let $(\psi \in)\ \Sigma_{\delta\Delta}$ be defined by $\Sigma \cup \{\delta, \Delta\}$. An interpretation is a function of type*

$$I:\ A \rightarrow (\Sigma \rightarrow \Sigma_{\delta\Delta})$$

An interpretation maps atomic actions to state transformations. If $I(a)(\sigma) = \delta$, the action cannot proceed in the current state σ; its execution is suspended. This need not necessarily lead to a definitive deadlock of the whole system, because some other component of the program may be enabled to take a next step. (See the transition rules below.) Note that the operator plus models global rather than local nondeterminism, since the choice may depend on the state, which is global.

The interpretation of $I(a)(\sigma) = \Delta$ is that the attempt to execute the action a leads to a failure (system error). Examples are dividing by 0 or, in the context of constraint programming, adding a fact to the store that makes it inconsistent. Failure is regarded as the "most undesirable situation", and we assume that if an action generates a failure in a certain state σ, then the same action will still generate failure in all possible future states (after σ) in which the system can evolve. This implies that failure is definitive, a situation of no recovery.

Given an arbitrary triple (A, Σ, I) (the set of atomic actions, the set of states, and the interpretation function I, respectively), we next describe how the semantics of the language $\mathcal{L}$ can be constructed. It is based on a *labelled transition system* $(\mathcal{L}, \mathit{Label}, \rightarrow)$. The set Label of labels is defined by $\mathit{Label} = \Sigma \times \Sigma_{\delta\Delta}$. A label represents the state transformation caused by the action that is performed during the transition step.

Definition 2.2 *Let I be an interpretation. The transition relation $\rightarrow \subseteq \mathcal{L} \times \mathit{Label} \times \mathcal{L}$ is defined as the smallest relation satisfying the rules of Table 1.*

In Table 1, the following conventions are used: if $s' = E$ then read t for $s'; t$, $s' \parallel t$ and $t \parallel s'$ in the rules R2 and R3. Note that (as always) σ and σ' range over the set of states Σ without δ and Δ.

The interpretation of a transition step $s \xrightarrow{\langle\sigma,\sigma'\rangle} s'$ is as follows. (As usual, we write $s \xrightarrow{\langle\sigma,\sigma'\rangle} s'$ for $(s, \langle\sigma, \sigma'\rangle, s') \in\rightarrow$.) Under the assumption that the current state is σ, the statement s can perform a computation step, changing the state into σ', and resulting in the statement s'.

[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]).

R1 $a \xrightarrow{\langle\sigma,\sigma'\rangle} E$ if $I(a)(\sigma)=\sigma'$	D1 $a \xrightarrow{\langle\sigma,\delta\rangle} E$ if $I(a)(\sigma)=\delta$	F1 $a \xrightarrow{\langle\sigma,\Delta\rangle} E$ if $I(a)(\sigma)=\Delta$
R2 $\dfrac{s \xrightarrow{\langle\sigma,\sigma'\rangle} s'}{s;t \xrightarrow{\langle\sigma,\sigma'\rangle} s';t}$	D2 $\dfrac{s \xrightarrow{\langle\sigma,\delta\rangle} E}{s;t \xrightarrow{\langle\sigma,\delta\rangle} E}$	F2 $\dfrac{s \xrightarrow{\langle\sigma,\Delta\rangle} E}{s;t \xrightarrow{\langle\sigma,\Delta\rangle} E}$
R3 $\dfrac{s \xrightarrow{\langle\sigma,\sigma'\rangle} s'}{\begin{array}{c} s \parallel t \xrightarrow{\langle\sigma,\sigma'\rangle} s' \parallel t \\ t \parallel s \xrightarrow{\langle\sigma,\sigma'\rangle} t \parallel s' \end{array}}$	D3 $\dfrac{s \xrightarrow{\langle\sigma,\delta\rangle} E \quad t \xrightarrow{\langle\sigma,\delta\rangle} E}{s \parallel t \xrightarrow{\langle\sigma,\delta\rangle} E}$	F3 $\dfrac{s \xrightarrow{\langle\sigma,\Delta\rangle} E}{\begin{array}{c} s \parallel t \xrightarrow{\langle\sigma,\Delta\rangle} E \\ t \parallel s \xrightarrow{\langle\sigma,\Delta\rangle} E \end{array}}$
R4 $\dfrac{s \xrightarrow{\langle\sigma,\sigma'\rangle} s'}{\begin{array}{c} s + t \xrightarrow{\langle\sigma,\sigma'\rangle} s' \\ t + s \xrightarrow{\langle\sigma,\sigma'\rangle} s' \end{array}}$	D4 $\dfrac{s \xrightarrow{\langle\sigma,\delta\rangle} E \quad t \xrightarrow{\langle\sigma,\delta\rangle} E}{s + t \xrightarrow{\langle\sigma,\delta\rangle} E}$	F4 $\dfrac{s \xrightarrow{\langle\sigma,\Delta\rangle} E \quad t \xrightarrow{\langle\sigma,\Delta\rangle} E}{s + t \xrightarrow{\langle\sigma,\Delta\rangle} E}$
	D5 $\dfrac{s \xrightarrow{\langle\sigma,\delta\rangle} E \quad t \xrightarrow{\langle\sigma,\Delta\rangle} E}{\begin{array}{c} s + t \xrightarrow{\langle\sigma,\delta\rangle} E \\ t + s \xrightarrow{\langle\sigma,\delta\rangle} E \end{array}}$	

Table 1: The transition system.

The last rule D5 shows that in a choice between suspension and failure a process always chooses to suspends. This is because, as explained above, failure is regarded as the most undesirable situation.

The rules for failure are F1-F4. The rule for parallel composition (F3) is based on the fact that failure is definitive. If one component fails, then the complete system can fail immediately. On the other hand, the system is not obliged to fail as soon as one component can fail: R3 can be applied before F3, and this allows the other components to make some proper steps beforehand. Such a possibility is quite natural since, in a model based on interleaving, it would be very expensive to check at every step that there are no components that fail (it would require a negative premise in F3).

Based on this transition system, an observational semantics is defined next. It gives for every statement the set of sequences of states corresponding to its (completed) transition sequences. Since processes only interact via a shared state, the relevant aspects of the behaviour of a process, from the point of view of the environment, consist of its interactions with that shared state. This marks an essential difference with synchronously communicating processes, where the observable behaviour of a process is determined by the actions themselves.

Definition 2.3 *Define* $\Sigma^+_{\delta\Delta} = \Sigma^+ \cup \Sigma^+ \cdot \{\delta, \Delta\}$. *(Concatenation of words is defined as usual, with the convention that for all* $w \in \Sigma^+_{\delta\Delta}$*:* $\delta \cdot w = \delta$ *and* $\Delta \cdot w = \Delta$*.) Let* $\mathcal{P}(\cdot)$ *be the set of subsets of* $(\cdot)$*. The function*

$$O : \mathcal{L} \to \Sigma \to \mathcal{P}(\Sigma^+_{\delta\Delta})$$

is given by $O[\![E]\!](\sigma) = \{\sigma\}$ *and, for* $s \neq E$,

$$O[\![s]\!](\sigma) = \{\sigma\sigma_1 \cdots \sigma_n\psi \mid s \xrightarrow{\langle\sigma,\sigma_1\rangle} s_1 \xrightarrow{\langle\sigma_1,\sigma_2\rangle} \cdots \xrightarrow{\langle\sigma_n,\psi\rangle} E\}$$

In the definition of O, only *connected* transition sequences are considered: the labels of successive transitions have the property that the last element of the first label equals the first element of the second. Note that the last element ψ of the sequences in $O[\![s]\!](\sigma)$ is either a proper state $\sigma' \in \Sigma$, or the deadlock state δ, or the failure state Δ.

The function O can also be recursively described as follows. First an auxiliary operator is introduced. Let $\cdot: \Sigma \times \mathcal{P}(\Sigma^+_{\delta\Delta}) \to \mathcal{P}(\Sigma^+_{\delta\Delta})$ be given by

$$\sigma \cdot X = \{\langle\sigma\rangle \cdot w \mid w \in X\}$$

Now we have, for $s \neq E$,

$$\begin{array}{rcl} O[\![s]\!](\sigma) & = & \bigcup\{\sigma \cdot O[\![s']\!](\sigma') \mid s \xrightarrow{\langle\sigma,\sigma'\rangle} s'\} \\ & \cup & \{\sigma \cdot \delta \mid s \xrightarrow{\langle\sigma,\delta\rangle} E)\} \\ & \cup & \{\sigma \cdot \Delta \mid s \xrightarrow{\langle\sigma,\Delta\rangle} E)\} \end{array}$$

The model O is in general (i.e., for arbitrary choice of A, I and Σ) not compositional for two different reasons. They are related to the fact that the semantics of our language is *nonuniform* (the meaning of atomic actions depends on the current state), and the fact that deadlock in general is dependent on the choice structure of statements—which is not taken into account by O. This we will illustrate by means of two examples.

Example 2.4 *Consider $\Sigma = \{0, 1, \ldots\}$ and let $s(\sigma)$ denote the successor of σ for $\sigma \in \Sigma$. Further let the set of actions A and the interpretation I be given by*

$$A = \{x := 0,\ x := 1,\ x := s(x)\},$$

$$I(x := 0)(\sigma) = 0,\ \ I(x := 1)(\sigma) = 1,\ \ I(x := s(x))(\sigma) = s(\sigma)$$

Now $O[\![x := 0; x := 1]\!](0) = \{001\} = O[\![x := 0; x := s(x)]\!](0)$, whereas

$$O[\![(x := 0; x := 1) \parallel x := 1]\!](0) = \{0011, 0101\}$$

$$O[\![(x := 0; x := s(x)) \parallel x := 1]\!](0) = \{0011, 0012, 0101\}$$

Example 2.5 *Consider $A = \{0, 1, ?1, \delta\}$, $\Sigma = \{0, 1\}$, and*

$$I(0)(\sigma) = 0,\ \ I(1)(\sigma) = 1,\ \ I(\delta)(\sigma) = \delta,\ \ I(?1)(\sigma) = \begin{cases} 1 & if\ \sigma = 1 \\ \delta & if\ \sigma = 0 \end{cases}$$

Now $O[\![0; (?1 + \delta)]\!](\sigma) = \{\sigma 0 \delta\} = O[\![(0; ?1) + (0; \delta)]\!](\sigma)$ whereas

$$\sigma 01\delta \in O[\![((0; ?1) + (0; \delta)) \parallel 1]\!](\sigma)$$

$$\sigma 01\delta \notin O[\![(0; (?1 + \delta)) \parallel 1]\!](\sigma)$$

The latter set does not contain $\sigma 01\delta$ because in the following transition sequence

$$(0; (?1 + \delta)) \parallel 1 \xrightarrow{\langle\sigma,0\rangle} (?1 + \delta) \parallel 1 \xrightarrow{\langle 0,1\rangle} ?1 + \delta \not\xrightarrow{\langle 1,\delta\rangle} E$$

the last transition is not possible.

2.3 Compositional semantics

Next we introduce a semantics D that describes the behaviour of $\mathcal{L}$ in a compositional manner. It is introduced using the transition system, and it is later shown to be compositional.

Definition 2.6 *Let $(X, Y \in) P$ be defined by*

$$P = \mathcal{P}(Q)$$

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

(The empty sequence is denoted by ϵ.) Again the concatenation $w_1 \cdot w_2$ of two sequences w_1 and w_2 in Q is recursively defined by $\langle \sigma, \delta \rangle \cdot w = \langle \sigma, \delta \rangle$, $\langle \sigma, \Delta \rangle \cdot w = \langle \sigma, \Delta \rangle$, and

$$(\langle \sigma, \sigma' \rangle w_1) \cdot w_2 = \langle \sigma, \sigma' \rangle (w_1 \cdot w_2)$$

Next a compositional model $D : \mathcal{L} \to P$ is defined as follows. We put $D[\![E]\!] = \{\epsilon\}$ and, for $s \neq E$,

$$D[\![s]\!] = \{\langle \sigma_1, \sigma_1' \rangle \cdots \langle \sigma_n, \psi \rangle \mid s \xrightarrow{\langle \sigma_1, \sigma_1' \rangle} s_1 \xrightarrow{\langle \sigma_2, \sigma_2' \rangle} \cdots \xrightarrow{\langle \sigma_n, \psi \rangle} E\}$$

The function D yields sets of sequences of *pairs* of states, rather than just states. The intuition behind such a pair $\langle \sigma, \sigma' \rangle$ is that if the current state is σ, then the computation at hand can transform this state into σ'. An important difference between the functions O and D is that in the definition of the latter, the transition sequences need not be connected: for instance, in the above definition σ_1' may be different from σ_2. The idea behind such a "gap" is that the state transformation from σ_1' to σ_2 may be caused by the environment, e.g., a program running in parallel with the present one.

The main interest of D 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 2.7 *Three operators $;, +, \| : P \times P \to P$ are introduced as follows.*

$$\{\epsilon\}; X = \{\epsilon\} \parallel X = X \parallel \{\epsilon\} = \{\epsilon\} + X = X + \{\epsilon\} = X$$

and, for $X_1 \neq \{\epsilon\} \neq X_2$,

- $$\begin{aligned} X_1; X_2 = \; & \bigcup \{\langle \sigma, \sigma' \rangle \cdot (\{w\}; X_2) \mid \langle \sigma, \sigma' \rangle \cdot w \in X_1\} \\ & \cup \\ & \{\langle \sigma, \delta \rangle \mid \langle \sigma, \delta \rangle \in X_1\} \\ & \cup \\ & \{\langle \sigma, \Delta \rangle \mid \langle \sigma, \Delta \rangle \in X_1\} \end{aligned}$$
- $$\begin{aligned} X_1 + X_2 = \; & \{\langle \sigma, \sigma' \rangle \cdot w \mid \langle \sigma, \sigma' \rangle \cdot w \in X_1\} \\ & \cup \\ & \{\langle \sigma, \sigma' \rangle \cdot w \mid \langle \sigma, \sigma' \rangle \cdot w \in X_2\} \\ & \cup \\ & \{\langle \sigma, \delta \rangle \mid \langle \sigma, \delta \rangle \in X_1 \wedge \langle \sigma, \delta \rangle \in X_2\} \\ & \cup \\ & \{\langle \sigma, \delta \rangle \mid \langle \sigma, \Delta \rangle \in X_1 \wedge \langle \sigma, \delta \rangle \in X_2\} \\ & \cup \\ & \{\langle \sigma, \delta \rangle \mid \langle \sigma, \delta \rangle \in X_1 \wedge \langle \sigma, \Delta \rangle \in X_2\} \\ & \cup \\ & \{\langle \sigma, \Delta \rangle \mid \langle \sigma, \Delta \rangle \in X_1 \wedge \langle \sigma, \Delta \rangle \in X_2\} \end{aligned}$$
- $$\begin{aligned} X_1 \parallel X_2 = \; & \bigcup \{\langle \sigma, \sigma' \rangle \cdot (\{w\} \parallel X_2) \mid \langle \sigma, \sigma' \rangle \cdot w \in X_1\} \\ & \cup \\ & \bigcup \{\langle \sigma, \sigma' \rangle \cdot (X_1 \parallel \{w\}) \mid \langle \sigma, \sigma' \rangle \cdot w \in X_2\} \\ & \cup \\ & \{\langle \sigma, \delta \rangle \mid \langle \sigma, \delta \rangle \in X_1 \wedge \langle \sigma, \delta \rangle \in X_2\} \\ & \cup \\ & \{\langle \sigma, \Delta \rangle \mid \langle \sigma, \Delta \rangle \in X_1\} \\ & \cup \\ & \{\langle \sigma, \Delta \rangle \mid \langle \sigma, \Delta \rangle \in X_2\} \end{aligned}$$

The definitions of ; and $\|$ are recursive. They can be formally justified by induction on the "length" of sets X, to be defined as the maximum of the lengths of their elements. (To be precise, this indicates that our domain P has to be restricted to sets for which such maximum exists. In the present context, this is not a limitation since we are not dealing with infinite behavior.)

The sequential composition of two sets X_1 and X_2 is as usual; it consists of the set of all words obtained by concatenating a word from X_1 and a word from X_2.

The nondeterministic composition $X_1 + X_2$ contains all sequences of X_1 and X_2 that start with a real computation step (i.e., a pair $\langle\sigma,\sigma'\rangle$). Such steps can always be taken autonomously. In addition, $X_1 + X_2$ contains a suspension step $\langle\sigma,\delta\rangle$ if it is contained in both components, or if $\langle\sigma,\delta\rangle$ is contained in the one component and $\langle\sigma,\Delta\rangle$ is contained in the other. Finally, $X_1 + X_2$ contains $\langle\sigma,\Delta\rangle$ only if both components do. Summarizing, the nondeterministic composition can be viewed as set union with the removal of certain suspension and failure steps. Apart from the interplay between δ and Δ, one can say that those suspension and failure steps are removed that occur only in one of the two components.

In the synchronous case, the parallel composition of two processes usually consists of three parts: the left merge, which contains those interleavings of computations starting with a step from the left process; the right merge, defined similarly; and finally the synchronization merge, consisting of computations that start with a step which is the result of the synchronization of a step from the left and a step from the right process. In the present asynchronous framework, the main difference is the absence of the synchronization merge. The first two sets in the definition of the parallel composition $X_1 \parallel X_2$ correspond to the left and the right merge. Although there is no synchronization merge, some kind of synchronization does take place: $X_1 \parallel X_2$ contains a pair $\langle\sigma,\delta\rangle$ only if it is contained in both components.

The definition of the above operators is clearly inspired by the transition rules of the transition system corresponding with each operator. This is at the same time the basis for their correctness with respect to the operational model O. In the semantics for concurrency, it is often the case that operational semantics (in Plotkin's SOS style, see [Plo81]) is more basic than denotational semantics in the sense that the latter can be derived from the former. See [Rut90] for a systematic development of this idea in the context of bisimulation equivalence.

Example 2.8 *Consider $A = \{?0, ?1, \delta\}$, $\Sigma = \{0,1\}$. Define $I(\delta)(\sigma) = \delta$ and*

$$I(?0)(\sigma) = \begin{cases} 0 & \text{if } \sigma = 0 \\ \delta & \text{if } \sigma = 1 \end{cases} \qquad I(?1)(\sigma) = \begin{cases} 1 & \text{if } \sigma = 1 \\ \delta & \text{if } \sigma = 0 \end{cases}$$

We have

$$D[\![?0;\delta]\!] \parallel D[\![?1;\delta]\!] =$$

$$\{\langle 0,0\rangle\langle 0,\delta\rangle,\ \langle 0,0\rangle\langle 1,\delta\rangle,\ \langle 1,\delta\rangle\} \parallel \{\langle 1,1\rangle\langle 0,\delta\rangle,\ \langle 1,1\rangle\langle 1,\delta\rangle,\ \langle 0,\delta\rangle\} =$$

$$\{\langle 0,0\rangle\langle 0,\delta\rangle,\ \langle 1,1\rangle\langle 1,\delta\rangle,\ \langle 0,0\rangle\langle 1,1\rangle\langle 1,\delta\rangle,\ \langle 1,1\rangle\langle 0,0\rangle\langle 0,\delta\rangle,$$

$$\langle 0,0\rangle\langle 1,1\rangle\langle 0,\delta\rangle,\ \langle 1,1\rangle\langle 0,0\rangle\langle 1,\delta\rangle\}$$

A subtle point is that in computing this parallel composition, one will also try to combine $\langle 0,0\rangle\langle 0,\delta\rangle$ from the left component and $\langle 1,1\rangle\langle 1,\delta\rangle$ from the right one. However, this will not contribute to the final outcome: starting with the step $\langle 0,0\rangle$ from the left followed by $\langle 1,1\rangle$ from the right, one ends up with

$$\langle 0,0\rangle\langle 1,1\rangle \cdot (\{\langle 0,\delta\rangle\} \parallel \{\langle 1,\delta\rangle\})$$

which equals $\emptyset$, since the latter parallel composition is empty and $w \cdot \emptyset = \emptyset$. The parallel composition $\{\langle 0,\delta\rangle\} \parallel \{\langle 1,\delta\rangle\}$ yields the empty set because a suspension pair $\langle\sigma,\delta\rangle$ is only included if it is contained, with the same state σ, in both components. In other words, these components should synchronize with respect to δ.

Using the above operators, the compositionality of D can be stated.

Theorem 2.9 *For all $s, t \in \mathcal{L}$, $* \in \{;, +, \|\}$,*

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

Finally, it is shown that D is *correct* with respect to the operational semantics O. That is, if two statements are distinguished by O, then D should distinguish them as well. The relation between O and D can be made precise using the following abstraction operator. Let $\alpha : P \to \Sigma \to \mathcal{P}(\Sigma^+_{\delta\Delta})$ be defined by

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

where

$$\alpha(x)(\sigma) = \begin{cases} \{\sigma\sigma_1\sigma_2 \cdots \sigma_n\psi\} & \text{if } x = \langle\sigma, \sigma_1\rangle\langle\sigma_1, \sigma_2\rangle \cdots \langle\sigma_n, \psi\rangle \\ \emptyset & \text{otherwise} \end{cases}$$

The operator α selects from a set X, given an initial state σ, all connected sequences starting with σ.

Theorem 2.10 $\alpha \circ D = O$

2.4 Stuttering

So far, we have been considering only one notion of observables (the semantics O). It gives for a program the set of all its computation sequences, each of which consists of all the states through which the computation leads. This notion of observables implicitly assumes the observability of an internal clock during each cycle of which an instruction is executed. The external observer then observes the effect on the state of the instruction executed during each cycle. However, a more abstract and realistic description is obtained when one assumes that during a computation only the state-changes are displayed. In other words, we want to abstract from repetition of states, which is sometimes called called *stuttering*.

In this subsection, we shall investigate how the above models should be changed if one wants to abstract from finite stuttering. That is, we shall define a second notion of observables (O_{ns}) in which finite repetitions of states are deleted. Next, a compositional characterization for this new model is provided.

Operationally, it is straightforward to abstract from stuttering. For a word $w \in \Sigma^+_{\delta\Delta}$, $del(w)$ is the word obtained from w by deleting all subsequent occurrences of identical states: formally, $del(\sigma \cdot \delta) = \sigma \cdot \delta$, $del(\sigma \cdot \Delta) = \sigma \cdot \Delta$, $del(\sigma) = \sigma$, and

$$del(\sigma \cdot \sigma' \cdot w) = \begin{cases} del(\sigma' \cdot w) & \text{if } \sigma = \sigma' \\ \sigma \cdot del(\sigma' \cdot w) & \text{if } \sigma \neq \sigma' \end{cases}$$

If $\beta : \mathcal{P}(\Sigma^+_{\delta\Delta}) \to \mathcal{P}(\Sigma^+_{\delta\Delta})$ is defined by $\beta(X) = \{del(w) \mid w \in X\}$ we can put

$$O_{ns} : \mathcal{L} \to \Sigma \to \mathcal{P}(\Sigma^+_{\delta\Delta}), \qquad O_{ns} = \beta \circ O$$

The above denotational model D is correct with respect to O and hence also with respect to O_{ns}:

$$O_{ns} = \beta \circ O = \beta \circ \alpha \circ D$$

Denotationally, things become more interesting and considerably more difficult if one is not satisfied with the model D. Although it is compositional and correct with respect to O_{ns}, it is not very abstract, i.e., it makes more distinctions than necessary. For instance, let $\tau \in A$ and let $I(\tau)(\sigma) = \sigma$, for all $\sigma \in \Sigma$. Then $D[\![\tau]\!] \neq D[\![\tau; \tau]\!]$, whereas both statements not only have the same operational meaning,

$$O_{ns}[\![\tau]\!](\sigma) = O_{ns}[\![\tau; \tau]\!](\sigma) = \{\sigma\}$$

but, more importantly, cannot be distinguished in any context. (As usual, a context $C(\cdot)$ is a statement 'with a whole' like, for instance, $s + (\cdot)$. The claim that the two statements above cannot be distinguished in any context can then be formulated as follows: for all contexts $C(\cdot)$,

$$O_{ns}[\![C(\tau)]\!](\sigma) = O_{ns}[\![C(\tau;\tau)]\!](\sigma) = \{\sigma\}.)$$

Let us try to define a model D_{ns} that is more abstract than D in such a way that it still is both compositional and correct with respect to O_{ns}.

A first attempt would be to try a similar approach as in the operational case by applying an abstraction operator to D that removes stuttering steps. Since the denotational model D yields sequences of *pairs* of states, one could try to remove those pairs in which the state is not changed: $\langle\sigma,\sigma\rangle$. This is not correct, though. Consider the action ?0 as in Example 2.8 above. Its denotational semantics is $D[\![?0]\!] = \{\langle 0,0\rangle, \langle 1,\delta\rangle\}$. Removing the stuttering pair would leave $\{\langle 1,\delta\rangle\}$ as a result. This is simply not correct with respect to the operational semantics O and O_{ns}, because all information about what happens in the state 0 has disappeared.

A somewhat more subtle proposal would be to replace two pairs $\langle\sigma,\sigma'\rangle\langle\sigma',\sigma'\rangle$ by $\langle\sigma,\sigma'\rangle$ (note that the information that the computation leads through σ' is not lost here). Again, this does not work in general. Consider $\Sigma = \{1,2\}$, $\tau \in A$ as above, and $a \in A$ with $I(a)(1) = 2$ and $I(a)(2) = 1$. Intuitively, the statements a and $a;\tau$ should get the same meaning. Their meanings under D are

$$D[\![a]\!] = \{\langle 1,2\rangle, \langle 2,1\rangle\}$$

$$D[\![a;\tau]\!] = \{\langle 1,2\rangle\langle 1,1\rangle, \langle 1,2\rangle\langle 2,2\rangle, \langle 2,1\rangle\langle 1,1\rangle, \langle 2,1\rangle\langle 2,2\rangle\}$$

In reducing the latter set to the former, the sequences $\langle 1,2\rangle\langle 2,2\rangle$ and $\langle 2,1\rangle\langle 1,1\rangle$ can be replaced by $\langle 1,2\rangle$ and $\langle 2,1\rangle$, respectively. Now the problem arises that it is in general not clear what to do with disconnected sequences like $\langle 1,2\rangle\langle 1,1\rangle$ and $\langle 2,1\rangle\langle 2,2\rangle$.

Therefore an alternative approach is taken. Rather than trying to make two sets of sequences equal by only *removing* certain stuttering steps, one can instead also *add* such steps, in case they are present in the one set but not in the other.

Definition 2.11 *The operator Close* $: P \to P$ *is defined as follows. For* $X \in P$*, the set Close*(X) *is the smallest set* $Y \in P$ *such that, for all* $\sigma,\sigma' \in \Sigma$ *and* $w, w_1, w_2 \in Q$*,*

1. $X \subseteq Y$
2. *if* $w_1 \cdot w_2 \in Y$ *and either* $w_1 \neq \epsilon$ *or* $\langle\sigma,\Delta\rangle \neq w_2 \neq \langle\sigma,\delta\rangle$ *then* $w_1 \cdot \langle\sigma,\sigma\rangle \cdot w_2 \in Y$
3. *if* $w_1 \cdot \langle\sigma,\sigma'\rangle \cdot \langle\sigma',\sigma'\rangle \cdot w_2 \in Y$ *then* $w_1 \cdot \langle\sigma,\sigma'\rangle \cdot w_2 \in Y$
4. *if* $w_1 \cdot \langle\sigma,\sigma\rangle \cdot \langle\sigma,\sigma'\rangle \cdot w_2 \in Y$ *then* $w_1 \cdot \langle\sigma,\sigma'\rangle \cdot w_2 \in Y$
5. *if* $w \cdot \langle\sigma,\sigma\rangle \cdot \langle\sigma,\delta\rangle \in Y$ *then* $w \cdot \langle\sigma,\delta\rangle \in Y$
6. *if* $w \cdot \langle\sigma,\sigma\rangle \cdot \langle\sigma,\Delta\rangle \in Y$ *then* $w \cdot \langle\sigma,\Delta\rangle \in Y$

Next we put

$$D_{ns} : \mathcal{L} \to P, \qquad D_{ns} = Close \circ D$$

Of the closure conditions above, clause 2 is the most difficult one to understand. It allows the addition of arbitrary stuttering steps $\langle\sigma,\sigma\rangle$ at any place in the sequences of Y, which is in itself quite intuitive. There is however one exception: if $\langle\sigma,\delta\rangle \in Y$ then it is not allowed to add $\langle\sigma,\sigma\rangle\cdot\langle\sigma,\delta\rangle$ (and similarly for Δ). This would violate the compositionality of D_{ns}, as the following example illustrates. Let $\Sigma = \{1\}$ and $\tau \in A$ be as above; let $a \in A$ with $I(a)(1) = \delta$. Then

$$D[\![a]\!] = \{\langle 1,\delta\rangle\}$$

$$D[\![\tau;a]\!] = \{\langle 1,1\rangle\cdot\langle 1,\delta\rangle\}$$

If the restriction in condition 2 above were to be dropped, then $D_{ns}[\![a]\!] = D_{ns}[\![\tau; a]\!]$. This is undesirable, since the two statements can be distinguished by O_{ns} using the context $\cdot + \tau$; that is,

$$1\delta \notin O_{ns}[\![a + \tau]\!](1)$$

$$1\delta \in O_{ns}[\![(\tau; a) + \tau]\!](1)$$

Under the assumption that D_{ns} is correct with respect to O_{ns}, this implies $D_{ns}[\![a+\tau]\!] \neq D_{ns}[\![(\tau; a)+\tau]\!]$. Hence D_{ns} cannot be compositional.

The new semantics D_{ns} satisfies the required properties.

Theorem 2.12 *The model D_{ns} is compositional and it is correct with respect to O_{ns}. That is, $O_{ns} = \beta \circ \alpha \circ D_{ns}$.*

The correctness is a direct consequence of the fact that the closure operator above does not influence the connectedness of sequences (and the fact that β and *Close* commute).

We conclude this section by giving an alternative description of D_{ns} in terms of an extended version of the transition relation.

Definition 2.13 *Let the transition relation*

$$\rightarrow_{ns} \subseteq \mathcal{L} \times Label \times \mathcal{L}$$

be defined as the smallest relation satisfying the rules given in Table 1 and, in addition, the rules in Table 2 ($\tau \in A$ is as above).

$$S1 \quad a \xrightarrow{\langle\sigma,\sigma'\rangle} \tau \quad \text{if } I(a)(\sigma)=\sigma'\neq\delta,\Delta$$

$$S2 \quad \frac{s \xrightarrow{\langle\sigma,\sigma'\rangle} s'}{s \xrightarrow{\langle\sigma,\sigma\rangle} s}$$

$$S3 \quad \frac{s \xrightarrow{\langle\sigma,\sigma'\rangle} s' \quad s' \xrightarrow{\langle\sigma',\sigma'\rangle} s''}{s \xrightarrow{\langle\sigma,\sigma'\rangle} s''}$$

$$S4 \quad \frac{s \xrightarrow{\langle\sigma,\sigma\rangle} s' \quad s' \xrightarrow{\langle\sigma,\sigma'\rangle} s''}{s \xrightarrow{\langle\sigma,\sigma'\rangle} s''}$$

$$S5 \quad \frac{s \xrightarrow{\langle\sigma,\sigma\rangle} s' \quad s' \xrightarrow{\langle\sigma,\delta\rangle} E}{s \xrightarrow{\langle\sigma,\delta\rangle} E}$$

$$S6 \quad \frac{s \xrightarrow{\langle\sigma,\sigma\rangle} s' \quad s' \xrightarrow{\langle\sigma,\Delta\rangle} E}{s \xrightarrow{\langle\sigma,\Delta\rangle} E}$$

Table 2: Additional rules for stuttering.

Now we have the following theorem.

Theorem 2.14 *For all $s \in \mathcal{L}$, $s \neq E$,*

$$D_{ns}[\![s]\!] = \{\langle\sigma_1,\sigma_1'\rangle \cdots \langle\sigma_n,\psi\rangle \mid s \xrightarrow{\langle\sigma_1,\sigma_1'\rangle}_{ns} s_1 \xrightarrow{\langle\sigma_2,\sigma_2'\rangle}_{ns} \cdots \xrightarrow{\langle\sigma_n,\psi\rangle}_{ns} E\}$$

This theorem can be proved using the observation that there is a clear correspondence between clause 2 of Definition 2.11 and rules S1 and S2 of Table 2, and similarly between the clauses 3, 4, 5 and 6 of Definition 2.11 and rules S3, S4, S5 and S6 of Table 2, respectively.

The above characterisation of D_{ns} is reminiscent of the way one can model weak observational congruence ([Mil80]) (or rooted tau-bisimulation ([BK86])), as strong bisimulation by adding rules like, for instance,

$$if \quad s \xrightarrow{\tau} s' \quad and \quad s' \xrightarrow{a} s'' \quad then \quad s \xrightarrow{a} s''$$

(See [BK86, vG87].) The present case is by its non-uniform nature more intricate.

In [dBKPR91], it is shown that D_{ns} is fully abstract with respect to O_{ns} for two classes of interpretations (generalizing earlier results of [HdBR90] and [dBP91]).

3 Concurrent constraint programming

In this section we present 'concurrent constraint programming' (cc programming), and show that it is an instance of the language $\mathcal{L}$ introduced in the previous section. Namely, we show that the computational model of cc and its compositional semantics can be described by instantiating properly the set of actions, the set of states, and the interpretation function which constitute the parameters of $\mathcal{L}$.

Concurrent constraint programming was proposed by Saraswat [Sar89, SR90, SRP90]. We follow here the definition given in [SR90], which is more general than the later formulations, because it deals with a more general notion of guard. In this version, concurrent constraint programming can be regarded as a generalization of most of the concurrent logic languages, including the 'more powerful' ones like Concurrent Prolog. See [Sar89] for a detailed list of the logic languages which are subsumed by the cc language, and the corresponding justification.

Before describing the cc paradigm we have to introduce the notion of *constraint system*. This notion plays a central role in concurrent constraint programming: it is both the data structure on which processes operate, and the communication medium by which processes interact with each other. Actually, the only visible activity of a process consists of manipulating this data structure.

3.1 Constraint systems

The definition of constraint system we will give here is a slightly modified version of the one described in [SR90]. We first need to introduce the notion of *simple constraint system*.

Definition 3.1 *A simple constraint system is a structure $\langle \mathcal{S}, \vdash \rangle$ where $(a, b \in)\mathcal{S}$ is a non-empty (denumerable) set of primitive constraints and $\vdash \subseteq \mathcal{P}(\mathcal{S}) \times \mathcal{S}$ is an entailment relation satisfying the following conditions:*

- *if $a \in \alpha$ then $\alpha \vdash a$, and*
- *if $\alpha \vdash a$ and $\forall b \in \alpha.\, \beta \vdash b$ then $\beta \vdash a$.*

Now extend $\vdash$ to a relation on $\mathcal{P}(\mathcal{S}) \times \mathcal{P}(\mathcal{S})$ as follows:

$$\alpha \vdash \beta \text{ iff } \forall b \in \beta.\, \alpha \vdash b$$

It is easy to see that $\vdash$ is a preorder. Denote by $\sim$ the associated equivalence relation:

$$\alpha \sim \beta \text{ iff } \alpha \vdash \beta \;\wedge\; \beta \vdash \alpha$$

We use the notation $[\alpha]$ to indicate the equivalence class of α, namely

$$[\alpha] = \{\beta |\, \alpha \sim \beta\}$$

The ordering relation induced by $\vdash$ on $\mathcal{P}(\mathcal{S})_{/\sim} \times \mathcal{P}(\mathcal{S})_{/\sim}$ is again denoted by $\vdash$.

Definition 3.2 *The constraint system generated by* $\langle \mathcal{S}, \vdash \rangle$ *is the structure*

$$\langle \mathcal{P}(\mathcal{S})_{/\sim}, \vdash \rangle$$

The structure $\langle \mathcal{P}(\mathcal{S})_{/\sim}, \vdash \rangle$ is a complete lattice with ordering relation $[\alpha] \leq [\beta]$ iff $[\beta] \vdash [\alpha]$. The least element, denoted by *true*, is given by $[\emptyset] = \{\alpha | \ \emptyset \vdash \alpha\}$. The greatest element, denoted by *false* (or *inconsistency*), is given by $[\mathcal{S}] = \{\alpha | \ \alpha \vdash \mathcal{S}\}$. The *lub* operation, denoted by $\sqcup$, is given by:

$$[\alpha] \sqcup [\beta] \ = \ [\alpha \cup \beta]$$

It is easy to see that this definition is *correct*, i.e., it does not depend upon the choice of the representants of the classes, and that it actually corresponds to the *lub*. Furthermore, it can be naturally extended to arbitrary sets of elements, thus showing that $\langle \mathcal{P}(\mathcal{S})_{/\sim}, \vdash \rangle$ is in fact a complete lattice.

For the sake of simplicity, we will indicate the equivalence class $[\alpha]$ by α.

Note. The standard definition of constraint system includes a restriction (compactness) on the entailment relation which has the effect to turn the structure $\langle \mathcal{P}(\mathcal{S})_{/\sim}, \vdash \rangle$ into a *complete algebraic* lattice. This guarantees that every element of the domain can be generated as the result of a (possibly infinite) computation.

Example 3.3 *Consider an alphabet consisting of*

- *a (denumerable) set of variables* $x, y, z, \ldots$,
- *for every* $n \geq 0$ *a set of n-adic function symbols* $f, g, \ldots$, *(the 0-adic ones will be called constants, and denoted by* $c, d, \ldots$*),*
- *the equality predicate* $=$.

Let $(t, u \in)T$ *be the set of terms over the alphabet. The Herbrand constraint system* $\langle H, \vdash \rangle$ *is the structure generated by the simple constraint system* $\langle Eq, \vdash \rangle$*, where* Eq *is the set of equations on* T*, and* $\vdash$ *is the entailment relation generated by Clark's equality axioms [Cla79], i.e. the minimal relation satisfying the requirements in Definition 3.1 and the following:*

1. $\forall t \in T.\ \emptyset \vdash t = t$
2. $\forall t, u \in T.\ \{t = u\} \vdash u = t$
3. $\forall t, u, v \in T.\ \{t = u, u = v\} \vdash t = v$
4. *If* f *is a n-adic function symbol* $(n \geq 0)$*, then*
$$\forall t_1, \ldots t_n, u_1, \ldots u_n \in T.\ \{t_1 = u_1, \ldots t_n = u_n\} \vdash f(t_1, \ldots t_n) = f(u_1, \ldots, u_n)$$
5. *If* f *is a n-adic function symbol* $(n \geq 0)$*, then*
$$\forall t_1, \ldots t_n, u_1, \ldots u_n \in T. \forall i \in [1, n].\ \{f(t_1, \ldots t_n) = f(u_1, \ldots, u_n)\} \vdash t_i = u_i$$
6. *If* f *and* g *are distinct function symbols of arity* m *and* n *respectively, then,*
$$\forall t_1, \ldots t_m, u_1, \ldots u_n \in T. \forall t, u \in T.\ \{f(t_1, \ldots t_m) = g(u_1, \ldots, u_n)\} \vdash t = u$$
7. *If* x *occurs in* t *and* $x \not\equiv t$ *then* $\forall u, v \in T.\ \{x = t\} \vdash u = v$.

Note. *Conditions (1), (2), (3) and (4) above correspond to the standard equality axioms (reflexivity, symmetry, transitivity and substitutivity). Conditions (5), (6) and (7) correspond to the so-called free-equality axioms, which enforce the interpretation of* $=$ *as syntactical identity, or, in other words, the interpretation of the function symbols as data constructors. The standard formulation of the free equality axioms (in first-order logic) is*

FE1 $f(x_1, \ldots x_n) = f(y_1, \ldots, y_n) \rightarrow x_1 = y_1 \ \wedge \ldots \wedge \ x_n = y_n$

FE2 $f(x_1, \ldots x_m) \neq g(y_1, \ldots, y_n)$

FE3 $x \neq t$ *if* x *occurs in* t *and* $x \not\equiv t$

Observe that the structure $\langle Eq, \vdash\rangle$ *does not contain negated equations. However, the negation of an equation* $t = u$ *can be represented by enforcing the inconsistency of* $t = u$*, i.e. by stating that* $t = u$ *entails the equality of all elements in the domain (of course, this representation makes sense only when* T *contains at least two terms). This is the way in which FE2 and FE3 are expressed (conditions (6) and (7) above).*

Assume, for instance, that the alphabet contains only the variable symbols x, y *and the constant symbols* c, d*. Then, the Herbrand constraint system will be (isomorphic to) the lattice represented in Figure 1.*

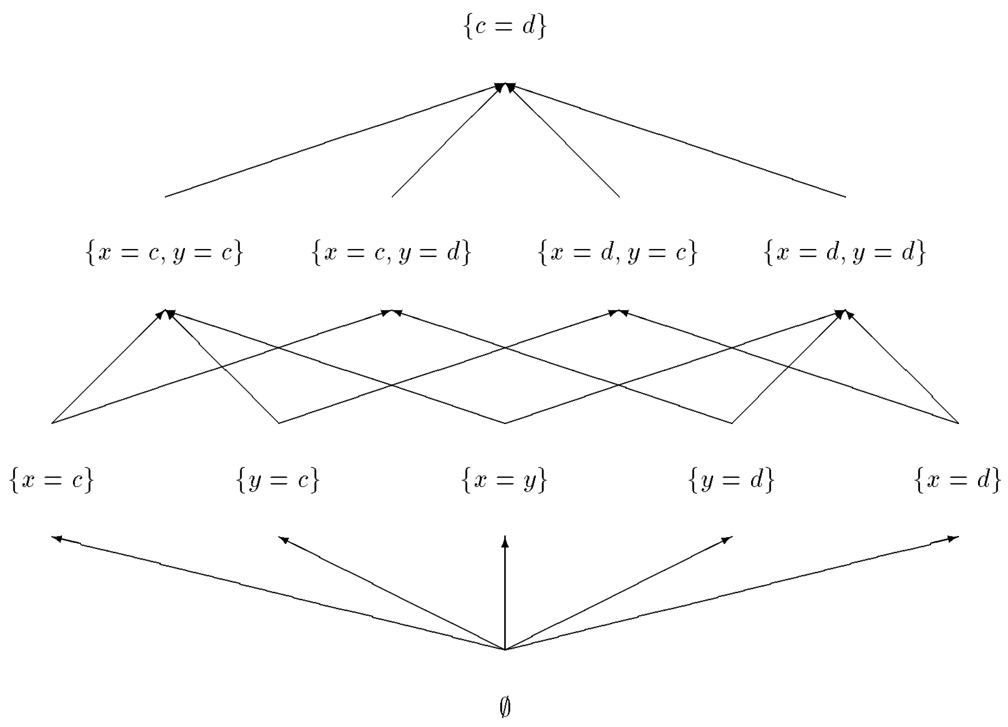

Figure 1: The Herbrand constraint lattice for x, y, c, d

3.2 The concurrent constraint paradigm

The class of cc languages is specified by the following grammar

$$s ::= \mathit{tell}(\alpha) \ | \ \mathit{ask}(\alpha) \ | \ s_1; s_2 \ | \ s_1 + s_2 \ | \ s_1 \parallel s_2$$

where α is a *basic constraint*, i.e., a constraint which is equivalent to a finite set of primitive constraints. The constraint system is a parameter of the language, and we will refer to it as $\langle \mathcal{C}, \vdash \rangle$. The intended computational model can be informally described as follows.

All processes of the system share a common store, which, at any stage of the computation, is given by the constraint established until that moment. (Usually at the beginning of the computation the store is supposed to be $\emptyset$.) The execution of an action of the form $tell(\alpha)$ modifies the current store by adding α to it. More formally, if the store at the moment of the execution is β, it will become $\alpha \sqcup \beta$ afterwards. An action of the form $ask(\alpha)$ is a test on the store: it can be executed only if the current store is *strong enough*, namely if it entails α. If this is not the case, then the process suspends (waiting for the store to get stronger by the contributions of the other processes). In any case, the store remains unchanged.

Note that both the tell and ask actions are monotonic, in the sense that the store after their execution is greater than or equal to the store before. This implies that the store will evolve monotonically during the computation.

There is a difference between the various cc languages concerning the tell actions. Some of them check for the consistency of α with respect to the current store, before executing $tell(\alpha)$. Others just add α regardless of whether or not it will lead to an inconsistent situation. The two kinds of tell are called *atomic tell* and *eventual tell* respectively in [SR90]. The different definitions of this operation are already present in the literature concerning concurrent logic programming. There the consistency check is usually called *atomic unification* (this explains the terminology "atomic tell": unification on the Herbrand universe corresponds, roughly, to adding a constraint to the store). An example of a language with atomic unification is Concurrent Prolog ([Sha86]), an example of a language with eventual tell is the language of Guarded Horn Clauses ([Ued87]).

Note that the inconsistent store is a situation of no recovery, since it is the top of the lattice, and the store can only evolve monotonically. Inconsistency is usually regarded as the *most undesirable* situation, the *failure* of the computation.

Obviously, the languages with atomic tell are more powerful, since processes can be specified which will try to avoid inconsistency, if possible. A process which has the alternative of provoking an inconsistency, or suspending, will always choose for the latter. If a process detects an *unavoidable* inconsistency (due to the fact that all the alternatives start with an action of the form $tell(\alpha)$ with α inconsistent with the current store), then it can immediately propagate it to the other processes so that the whole system terminates (with failure). It would not make sense to oblige the system to wait as long as possible (i.e. to execute first all the other processes); in fact the store evolves monotonically, and therefore if α is inconsistent with the current store it will be inconsistent with all the future stores as well.

3.3 Concurrent constraint languages as instances of $\mathcal{L}$

We show now how to derive the formal computational model and, correspondingly, the compositional semantics, for both cc languages with atomic and eventual tell, as an instance of $\mathcal{L}$, i.e. by instantiating properly the parameters of $\mathcal{L}$: the set of actions, the set of states, and the interpretation function.

In both cases of atomic and eventual tell, the set of atomic actions of the language is given by

$$A = \{ask(\alpha) |\ \alpha \text{ is a basic constraint}\} \cup \{tell(\alpha) |\ \alpha \text{ is a basic constraint}\}$$

The state of a computation is given by the store, therefore we can define

$$\Sigma = \mathcal{C}$$

Concerning the interpretation function of the ask actions, remember that $ask\ (\alpha)$ tests the store and it suspends if the store does not entail α. Furthermore, the cc computational model of [SR90] prescribes the following laws for the suspension mechanism:

- a process can suspend only if it has no possibility to proceed with a 'proper transition', and

- the system can suspend only if all processes suspend (as long as some processes can proceed and – hopefully – modify the store, suspension will be delayed).

We see that the suspension mechanism corresponds to a δ transition (cfr the table of the transition system). Therefore we can model the ask actions as follows:

$$I(ask(\sigma))(\sigma') \quad = \quad \begin{cases} \sigma' & \text{if } \sigma' \vdash \sigma \\ \delta & \text{otherwise} \end{cases}$$

For the tell actions, the interpretation function depends upon the kind of tell, atomic or eventual. We start with eventual tell, since it is easier to describe.

3.4 Concurrent constraint languages with eventual tell

In the languages with eventual tell, the action $tell(\alpha)$ always proceeds and it adds α to the store. Therefore we define:

$$I(tell(\sigma))(\sigma') \quad = \quad \sigma \sqcup \sigma'$$

In the following examples, we consider the Herbrand constraint system $\langle H, \vdash \rangle$ described in Example 3.3.

Example 3.4 *Consider the processes*

$$s_1 \quad = \quad tell(x = c) \quad + \quad tell(y = c)$$

$$s_2 \quad = \quad tell(x = d) \quad + \quad tell(y = d)$$

Let s be the system

$$s = s_1 \parallel s_2$$

Bearing in mind that $\emptyset = true$ and $\{c = d\} = false$, the traces generated by s are the following:

$$\begin{aligned} O[\![s]\!](true) \quad = \quad \{ \quad & true \cdot \{x = c\} \cdot false, \\ & true \cdot \{x = c\} \cdot \{x = c, y = d\}, \\ & true \cdot \{y = c\} \cdot \{x = d, y = c\}, \\ & true \cdot \{y = c\} \cdot false, \\ & true \cdot \{x = d\} \cdot false, \\ & true \cdot \{x = d\} \cdot \{x = d, y = c\}, \\ & true \cdot \{y = d\} \cdot \{x = c, y = d\}, \\ & true \cdot \{y = d\} \cdot false \qquad \} \end{aligned}$$

The sequences ending in false derive from the fact that both s_1 and s_2 can perform one of their alternatives regardless of what the other process has done in the past.

Example 3.5 *Consider the processes*

$$s_1 \quad = \quad ask(x = c) \quad ; \quad tell(y = c)$$

$$s_2 \quad = \quad tell(x = c) \quad + \quad tell(x = d)$$

Let s be the system

$$s = s_1 \parallel s_2$$

We have

$$\begin{aligned} O[\![s]\!](true) \quad = \quad \{ \quad & true \cdot \{x = c\} \cdot \{x = c\} \cdot \{x = c, y = c\}, \\ & true \cdot \{x = d\} \cdot \delta \qquad \} \end{aligned}$$

$$\begin{aligned} O_{ns}[\![s]\!](true) \quad = \quad \{ \quad & true \cdot \{x = c\} \cdot \{x = c, y = c\}, \\ & true \cdot \{x = d\} \cdot \delta \qquad \} \end{aligned}$$

Note that the execution of an ask action will always introduce the stuttering phenomenon.

3.5 Concurrent constraint languages with atomic tell

As already explained, the main difference between cc atomic tell and cc eventual tell is that the execution of an atomic tell can take place only if it does not lead to an inconsistent store. The computational model of cc [SR90] prescribes the following laws concerning the detection of an unavoidable inconsistency (failure):

- it can happen only when there are no other alternatives, and
- it can immediately be propagated to the whole system.

We see that the failure corresponds to a Δ transition (cfr the transition system table). Bearing this in mind, we can define the interpretation function of the atomic tell actions as follows:

$$I(tell(\sigma))(\sigma') = \begin{cases} \sigma' \sqcup \sigma & \text{if } \sigma' \sqcup \sigma \neq false \\ \Delta & \text{otherwise} \end{cases}$$

Note that the computations of systems based on atomic tell will never generate an inconsistent store, therefore we could restrict the set of proper states as follows:

$$\Sigma = \mathcal{C} \setminus \{false\}.$$

Example 3.6 *Consider the processes s_1, s_2 and s as described in Example 3.4. With the new interpretation function, which models the consistency check, we obtain*

$$\begin{array}{rcl} O[\![s]\!](true) & = & \{ \quad true \cdot \{x = c\} \cdot \{x = c, y = d\}, \\ & & \quad true \cdot \{y = c\} \cdot \{x = d, y = c\}, \\ & & \quad true \cdot \{x = d\} \cdot \{x = d, y = c\}, \\ & & \quad true \cdot \{y = d\} \cdot \{x = c, y = d\} \quad \} \end{array}$$

Note the difference with Example 3.4: here the processes s_1 and s_2 will always test the compatibility of their choices with the choices already done by the other.

The processes defined in Example 3.5 yield the same traces as before.

Example 3.7 *Consider the processes*

$$\begin{array}{rcl} s_1 & = & ask(x = c) \ + \ (tell(x = c) \ ; \ tell(y = d)) \\ s_2 & = & tell(x = d) \end{array}$$

Let s be the system

$$s = s_1 \parallel s_2$$

We have

$$\begin{array}{rcl} O_{ns}[\![s]\!](true) & = & \{ \quad true \cdot \{x = c\} \cdot \Delta, \\ & & \quad true \cdot \{x = c\} \cdot \{x = c, y = d\} \cdot \Delta, \\ & & \quad true \cdot \{x = d\} \cdot \delta \quad \} \end{array}$$

Note that if s_2 starts first, then s_1 will prefer to suspend (waiting – in this case in vain – for some process to add information to the store) rather than to enter "the state of no return" (Δ).

4 Further Examples of Instances

In this section we give some other examples of instances: a language with assignments and IO-automata. Recall that an instance consists of a specific choice for the set of states Σ, the interpretation I, and the set of atomic actions A. More examples of instances (like asynchronous CSP and asynchronous CCS) can be found in [dBKPR91].

4.1 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 $\mathcal{E}(e)(\sigma)$ 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

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

With this choice for A, Σ and I, the semantic models 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:

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

4.2 Input/Output-automata

The next example shows that Input/Output-automata (IO-automata) are an instance of our paradigm. This may come as a surprise because the communication between IO-automata is defined in a synchronous way. However, as our instance shows, the fact that an IO-automaton is not allowed to refuse inputs makes its communication mechanism asynchronous. We first give a short introduction to IO-automata. More information can be found in [LT87], and in [Jon90] on which this introduction is based.

We assume given a set of events $(e \in)$ $Event$.

An IO-automaton is formally a tuple

$$\langle In, Out, States, InitState, Rel \rangle$$

where $In \subseteq Event$ is a set of events called input events, $Out \subseteq Event$ is a set events called output events, $States$ is an arbitrary set of states, $InitState \in States$ is a special state called the initial state and $Rel \subseteq States \times (In \cup Out) \times States$ is a relation called the transition relation.

A triple (st, e, st') in Rel states that the IO-automaton can make a transition from state st to state st' via event e. Usually we denote an element (st, e, st') in Rel by $st \xmapsto{e} st'$

(This transition relation should not be confused with the transition relation $\to$ which used in the construction of the semantic models for $\mathcal{L}$. The same holds for the set of states.)

There are two restrictions:

- The transition relation Rel should satisfy: for all states $st \in States$ and for all events $e \in In$ there is a state $st' \in States$ such that $st \xmapsto{e} st'$.
- The intersection $In \cap Out$ should be empty.

The behaviour of an IO-automaton $\alpha = \langle In, Out, States, InitState, Rel \rangle$ is defined with the help of transition sequences generate events only from Out. It is given by the set $Steps(\alpha)$, defined as follows:

$$\begin{aligned} Steps(\alpha) = \{ e_1 e_2 \ldots e_n \ &: \ \forall i : 1 \leq i \leq n.\ e_i \in Out \ \wedge \\ &\exists st_1, \ldots, st_n \in States.\ InitState \xmapsto{e_1} st_1 \xmapsto{e_2} \ldots \xmapsto{e_n} st_n \ \wedge \\ &\forall st \in States,\ e \in Out.\ \neg(st_n \xmapsto{e} st) \}. \end{aligned}$$

Transitions with events in In do not play a role in $Steps(\alpha)$, but they play an important role in the parallel composition of IO-automata, which we define next.

Definition 4.1 *Assume given two automata $\alpha_i = \langle In_i, Out_i, States_i, InitState_i, Rel_i\rangle$ $(i = 1, 2)$. The parallel composition $\alpha_1 \parallel \alpha_2$ is a partial operator that is only defined if α_1 and α_2 have disjoint sets of output events*

$$Out_1 \cap Out_2 = \emptyset$$

and if the unions of input and output events are equal

$$In_1 \cup Out_1 = In_2 \cup Out_2.$$

If these conditions are satisfied then the parallel composition $\alpha_1 \parallel \alpha_2$ is the IO-automaton

$$\langle In, Out, States, InitState, Rel\rangle$$

with

$$In = In_1 \cap In_2,$$

$$Out = Out_1 \cup Out_2,$$

$$States = States_1 \times States_2,$$

$$InitState = (InitState_1, InitState_2),$$

and for all states $st_1, st_1' \in States_1$, $st_2, st_2' \in States_2$, for all events $e \in Event$,

$$Rel((st_1, st_2), e, (st_1', st_2')) \text{ if and only if } Rel_1(st_1, e, st_1') \text{ and } Rel_2(st_2, e, st_2').$$

Next we give an example. The example consists of two IO-automata, the first one modelling a light switch, the second an operator of the switch, and their parallel composition.

Example 4.2 1. *The IO-automaton L for the light switch is given by $In_L = \{on, off\}$, $Out_L = \{light\}$, $States_L = \{l, r\}$, $InitState = r$, and the relation Rel_L is given in the following table:*

$$\begin{array}{lcl} l & \xmapsto{light} & r, \\ l & \xmapsto{on} & l, \\ l & \xmapsto{off} & r, \\ r & \xmapsto{on} & l, \\ r & \xmapsto{off} & r. \end{array}$$

Note that $Steps(L) = \emptyset$. The transition graph is given in Figure 2 (the initial state is indicated by the arrow head).

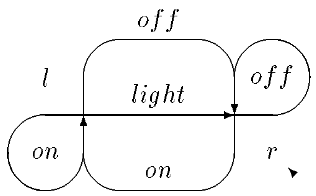

Figure 2: Transition Graph of the IO-automaton L

2. *The second automaton Y describes an operator of the light switch (switching it on, off and on again). It is given by $In_Y = \{light\}$, $Out_Y = \{on, off\}$, $States_Y = \{1,2,3,4\}$, $InitState = 1$, and the relation Rel_Y is given by:*

$$\begin{array}{ccc} 1 & \stackrel{light}{\longmapsto} & 1, \\ 2 & \stackrel{light}{\longmapsto} & 2, \\ 3 & \stackrel{light}{\longmapsto} & 3, \\ 4 & \stackrel{light}{\longmapsto} & 4, \end{array} \qquad \begin{array}{ccc} 1 & \stackrel{on}{\longmapsto} & 2, \\ 2 & \stackrel{off}{\longmapsto} & 3, \\ 3 & \stackrel{on}{\longmapsto} & 4. \end{array}$$

We have that

$$Steps(Y) = \{on\ off\ on\}$$

(a set with one element). This automaton is shown in Figure 3.

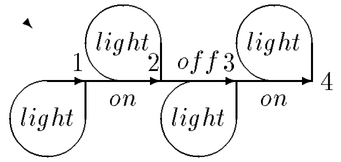

Figure 3: Transition graph of the IO-automaton Y

3. *Next we take the parallel composition $L \parallel Y$ of these two automata and get $In = \emptyset$, $Out = \{on, off, light\}$, $States = \{(l,1),(l,2),(l,3),(l,4),(r,1),(r,2),(r,3),(r,4)\}$, $InitState = (r,1)$, and the relation Rel:*

$$\begin{array}{ccc} (l,1) & \stackrel{light}{\longmapsto} & (l,1), \\ (l,2) & \stackrel{light}{\longmapsto} & (l,2), \\ (l,3) & \stackrel{light}{\longmapsto} & (l,3), \\ (l,4) & \stackrel{light}{\longmapsto} & (l,4), \end{array} \qquad \begin{array}{ccc} (l,1) & \stackrel{on}{\longmapsto} & (l,2), \\ (l,3) & \stackrel{on}{\longmapsto} & (l,4), \\ (r,1) & \stackrel{on}{\longmapsto} & (l,2), \\ (r,3) & \stackrel{on}{\longmapsto} & (l,4), \end{array} \qquad \begin{array}{ccc} (l,2) & \stackrel{off}{\longmapsto} & (r,3), \\ (r,2) & \stackrel{off}{\longmapsto} & (r,3). \end{array}$$

See Figure 4. The set $Steps(L \parallel Y)$ is

$$\{on\ light\ off\ on\ light, on\ off\ on\ light\}.$$

There is a hiding operator on IO-automata (see [LT87]), but we cannot model this operator in the present framework: it requires a block construct in the language $\mathcal{L}$. (This extension will be discussed in a forthcoming paper.)

Next we describe how IO-automata can be embedded into an instantiation of the language $\mathcal{L}$. First we observe that there is no language for IO-automata: they are given as tuples. Therefore we shall encode the transitions of the IO-automaton as a statement (procedure call) and then we use the parallel composition operator of $\mathcal{L}$ as the semantic counterpart of the parallel composition operator on IO-automata. We make the following steps:

1. we introduce a set of identifiers ($\alpha \in$) $Ident$ for IO-automata;

2. let the set of atomic actions (using a CSP-like notation) be given by

$$A = \{\alpha?e\ :\ \alpha \in Ident,\ e \in Event\ \} \cup \{\alpha!e\ :\ \alpha \in Ident,\ e \in Event\ \};$$

   Actions $\alpha?e$ will be associated to input events of α, actions $\alpha!e$ to output events.

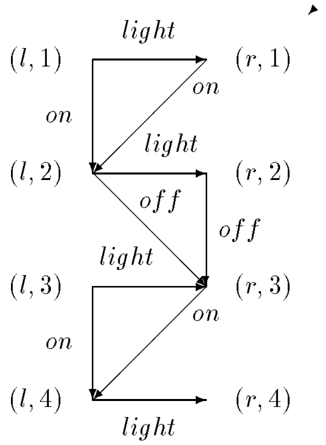

Figure 4: Transition Graph of the IO-automaton $(L \parallel Y)$

3. The set of states is defined by $\Sigma = Ident \rightarrow Event^*$. A state yields for each identifier a sequence of events which models the contents of a buffer, the last event of the word being the last event added to the buffer. Assume that ϵ denotes the empty word, and that operations $first$ and $tail$ are defined in the usual way on $Event^*$.

4. The interpretation function $I : A \rightarrow (\Sigma \rightarrow \Sigma_\delta)$ is defined as follows:

$$I(\alpha?e)(\sigma) = \begin{cases} \delta & if\ first(\sigma(\alpha)) \neq e\ or\ \sigma(\alpha) = \epsilon \\ \lambda\alpha'.\begin{cases} \sigma(\alpha') & if\ \alpha \neq \alpha' \\ tail(\alpha') & if\ \alpha = \alpha' \end{cases} & if\ first(\sigma(\alpha)) = e \end{cases}$$

$$I(\alpha!e)(\sigma) = \begin{cases} \delta & if\ \sigma(\alpha) \neq \epsilon \\ \lambda\alpha'.\begin{cases} \sigma(\alpha')\cdot e & if\ \alpha \neq \alpha' \\ \sigma(\alpha') & if\ \alpha = \alpha' \end{cases} & if\ \sigma(\alpha) = \epsilon \end{cases}$$

   Only when its buffer is empty the automaton can do output events. This is how we simulate the synchronization of the events: if an automaton performs an output event then it is stored in the buffers of all other automata. If an automaton has an non-empty buffer then it has to "follow" its buffer.

5. We code the transitions of the IO-automata in a set of recursive procedures. Here we extend the language $\mathcal{L}$ with a simple mechanism for recursion as described below.

The next step is to construct a statement that "drives" the IO-automaton. The problem is that the transition relation of an IO-automaton can contain cycles. A natural way to model this is to use recursion, but this is not yet included in $\mathcal{L}$. For this example we extend the transition system with rules for recursion. Full treatment of recursion in $\mathcal{L}$ (including divergence) is a topic of future research. Here we extend the language $\mathcal{L}$ as follows: Let $(x \in)\ Pvar$ be a set of procedure variables (for each instantiation we choose a specific set of procedure variables). Then the (extended) language $\mathcal{L}$ contains statements $s \in Stat$ of the form

$$s ::= a \mid s_1 ; s_2 \mid s_1 + s_2 \mid s_1 \parallel s_2 \mid x$$

We assume that every procedure variable x has an associated statement s_x (called the body of x: notation $x \Leftarrow s_x$). Next we extend the transition system of table 1 with four extra rules that can be found in table 3.

$\dfrac{s_x \xrightarrow{\langle\sigma,\sigma'\rangle} s}{x \xrightarrow{\langle\sigma,\sigma'\rangle} s}$	$\dfrac{s_x \xrightarrow{\langle\sigma,\sigma'\rangle} E}{x \xrightarrow{\langle\sigma,\sigma'\rangle} E}$
$\dfrac{s_x \xrightarrow{\langle\sigma,\delta\rangle} E}{x \xrightarrow{\langle\sigma,\delta\rangle} E}$	$\dfrac{s_x \xrightarrow{\langle\sigma,\Delta\rangle} E}{x \xrightarrow{\langle\sigma,\Delta\rangle} E}$

Table 3: The four extra rules for the transition system

For the IO-automata instantiation we take $PVar = States \times Ident$.

Assume given an IO-automaton $\alpha = \langle In, Out, States, InitState, Rel\rangle$. With each procedure variable (st, α) (α is the identifier of the automaton, and st is ranging over its states) we associate the following body. It is the sum (the generalization of the choice-operator $+$ to more than two arguments) over the statements

- for every $st' \in States, e \in In$ with $Rel(st, e, st')$ the statement $\alpha?e; (st', \alpha)$,
- for every $st' \in States, e \in Out$ with $Rel(st, e, st')$ the statement $\alpha!e; (st', \alpha)$.

The statement associated to α is $(InitState, \alpha)$ (i.e., one procedure call).

Example 4.3 *As an example consider the automaton* $\alpha = \langle In,\ Out,\ States,\ InitState,\ Rel\rangle$ *of Figure 5, with* $In = \{i_1, i_2\}$, $Out = \{o_1, o_2, o_3\}$, $States = \{0, 1, 2, 3\}$, $InitState = 0$. *We have that* $PVar = \{(0, \alpha), (1, \alpha), (2, \alpha), (3, \alpha)\}$, *elements of which have the following associated statements:*

$$(0, \alpha) \Leftarrow \alpha?i_1; (2, \alpha) + \alpha?i_2; (2, \alpha) + \alpha!o_1; (1, \alpha)$$

$$(1, \alpha) \Leftarrow \alpha?i_1; (3, \alpha) + \alpha?i_2; (3, \alpha)$$

$$(2, \alpha) \Leftarrow \alpha?i_1; (3, \alpha) + \alpha?i_2; (3, \alpha) + \alpha!o_2; (3, \alpha)$$

$$(3, \alpha) \Leftarrow \alpha?i_1; (3, \alpha) + \alpha?i_2; (3, \alpha)$$

The statement associated to α *is the procedure call* $(0, \alpha)$.

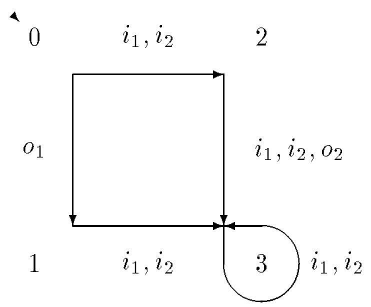

Figure 5: The relation *Rel* of the IO-automaton α

Example 4.4 *We return to Example 4.2. The associated bodies for procedure variables* (r, L) *and* (l, L) *are*

$$(r, L) \Leftarrow L?off; (r, L) + L?on; (l, L)$$

$$(l, L) \Leftarrow L!light; (r, L) + L?off; (r, L) + L?on; (l, L)$$

and for procedure variables $(1, Y), (2, Y), (3, Y)$ *and* $(4, Y)$ *the bodies are*

$$(1, Y) \Leftarrow Y?light; (1, Y) + Y!on; (2, Y)$$

$$(2, Y) \Leftarrow Y?light; (2, Y) + Y!off; (3, Y)$$

$$(3, Y) \Leftarrow Y?light; (3, Y) + Y!on; (4, Y)$$

$$(4, Y) \Leftarrow Y?light; (4, Y)$$

The statement is

$$(r, L) \parallel (1, Y)$$

consisting of two procedure calls in parallel.

As a last step we relate the *Steps* function and the operational model O showing the correctness of the above construction. First we define a function

$$abstr : \Sigma \times (\Sigma_{\delta\Delta}) \rightarrow Event^*$$

by

$$abstr(\sigma, \delta) = \epsilon$$

$$abstr(\sigma, \Delta) = \epsilon$$

$$abstr(\sigma_1, \sigma_2) = \begin{cases} e & if\ \exists\alpha \in Ident.\ \sigma_1(\alpha) = \sigma_2(\alpha) \wedge \forall\alpha' : \alpha' \neq \alpha.\ \sigma_1(\alpha') \cdot e = \sigma_2(\alpha') \\ \epsilon & otherwise \end{cases}$$

We extend the function *abstr* to $\Sigma^+_{\delta\Delta}$ by

$$abstr(\sigma_1\sigma_2\sigma_3 \cdots \sigma_{n-1}\psi) = abstr(\sigma_1, \sigma_2) \cdot abstr(\sigma_2, \sigma_3) \cdots abstr(\sigma_{n-1}, \psi).$$

Then we have

Lemma 4.5 *Let* $\alpha = \langle In, Out, States, InitState, Rel\rangle$ *be an IO automaton. When we take*

$$PVar = \{(st, \alpha)\ :\ st \in States\}$$

and construct procedure bodies as outlined above, then we have

$$abstr(O[\![(InitState, \alpha)]\!](\lambda\alpha'.\epsilon)) = Steps(\alpha)$$

We have a similar lemma for the parallel composition:

Lemma 4.6 *Let* $\alpha_i = \langle In_i, Out_i, States_i, InitState_i, Rel_i\rangle$ *be two IO-automata* $(i = 1, 2)$*. If we take*

$$PVar = \{(st, \alpha_1)\ :\ st \in States_1\} \cup \{(st, \alpha_2) : st \in States_2\}$$

with associated bodies, then we have

$$abstr(O[\![(InitState_1, \alpha_1) \parallel (InitState_2, \alpha_2)]\!](\lambda\alpha'.\epsilon)) = Steps(\alpha_1 \parallel \alpha_2).$$

(In this lemma the first $\parallel$ denotes the operator of the language $\mathcal{L}$ and the second $\parallel$ the parallel composition on IO-automata.)

5 Future Work

Various extensions of the general compositional model for asynchronous communication are being investigated in ongoing research. Important additional features to be incorporated are recursion and infinite behaviour. Moreover, a general definition of a "hiding" operator is being investigated, which covers diverse notions such as the hiding of logical variables in concurrent logic languages and of local states in shared-variable concurrency. Also of interest is the development of process algebras for the various asynchronous communication mechanisms.

Acknowledgements

We would like to thank the members of the Amsterdam Concurrency Group headed by Jaco de Bakker for the discussions and constructive comments on this chapter. Also we like to thank Jan-Willem Klop who suggested the picture of the lattice and for his comments on the contents.

References

[BHR84] S.D. Brookes, C.A.R. Hoare, and W. Roscoe. A theory of communicating sequential processes. *Journal of ACM*, 31:499–560, 1984.

[BK86] 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, 1986. North-Holland.

[Cla79] K.L. Clark. Predicate logic as a computational formalism. Technical Report DOC 79/59, Imperial College, Dept. of Computing, London, 1979.

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

[dBKPR91] F.S. de Boer, J.N. Kok, C. Palamidessi, and J.J.M.M. Rutten. The failure of failures: Towards a paradigm for asynchronous communication. In J.C.M. Baeten and J.F. Groote, editors, *Proc. of CONCUR 91*, volume 527 of *Lecture Notes in Computer Science*, pages 111 – 126. Springer-Verlag, 1991.

[dBP90] F.S. de Boer and C. Palamidessi. Concurrent logic languages: Asynchronism and language comparison. In *Proc. of the North American Conference on Logic Programming*, pages 175–194. The MIT Press, 1990.

[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.

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

[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.

[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.

[Jon90] B. Jonsson. A hiearchy of compositional models of I/O-automata. In *Proc. MFCS*, volume 452 of *Lecture Notes in Computer Science*, pages 347–354, 1990.

[Jos90] M.B. Josephs. Receptive process theory. Technical report, Eindhoven University, 1990.

[Kok89] J.N. Kok. Traces, histories and streams in the semantics of nondeterministic dataflow. Technical Report 91, Abo Akademi, Finland, 1989.

[LT87] N. Lynch and M. Tuttle. Hierarchical correctness proofs for distributed algorithms. In *Proc.* 6^{th} *ACM PoDC*, pages 137–151, 1987.

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

[Plo81] G.D. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, Aarhus University, Computer Science Department, 1981.

[Rut90] J.J.M.M. Rutten. Deriving denotational models for bisimulation from Structured Operational Semantics. In M. Broy and C.B. Jones, editors, *Programming concepts and methods, proceedings of the IFIP Working Group 2.2/2.3 Working Conference*, pages 155–177. North-Holland, 1990.

[Sar89] V.A. Saraswat. *Concurrent Constraint Programming Languages.* PhD thesis, Carnegie-Mellon University, January 1989. Published by The MIT Press, U.S.A., 1990.

[Sha86] E. Y. Shapiro. Concurrent Prolog: A progress report. *Computer*, 19(8):44–58, 1986.

[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, New York, 1990. ACM.

[SRP90] V.A. Saraswat, M. Rinard, and P. Panangaden. A fully abstract semantics for concurrent constraint programming. In *Proc. of the eighteenth ACM Symposium on Principles of Programming Languages*, New York, 1990. ACM.

[Ued87] K. Ueda. Guarded Horn Clauses. In E. Y. Shapiro, editor, *Concurrent Prolog: Collected Papers.* The MIT Press, 1987.

[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 336–447. Springer-Verlag, 1987.