From Concurrent Logic Programming to Concurrent Constraint Programming

Frank S. de Boer

Department of Mathematics and Computer Science
Free University
de Boelelaan 1081, 1081 HV Amsterdam, The Netherlands
frb@cwi.nl

Catuscia Palamidessi

Dipartimento di Informatica e Scienze dell'Informazione
(DISI)
Università di Genova
via Benedetto XV 3, 16132, Genova, Italy
catuscia@di.unipi.it

Abstract

The endeavor to extend logic programming to a language suitable for concurrent systems has stimulated in the last decade an intensive research, resulting in a large variety of proposals. A common feature of the various approaches is the attempt to define mechanisms for concurrency within the logical paradigm, the driving ideal being the balance between expressiveness and declarative reading. In this survey we present the motivations, the principal lines along which the field has developed, the various paradigms which have been proposed, and the main approaches to the semantic foundations.

1 Introduction

Among the various reasons which have contributed to the popularity of logic programming, one is the opinion that it is an *inherently parallel* language, therefore suitable for parallel and distributed architectures. The pure language can already be regarded as a model for parallel computation: in the so-called *process interpretation* (van Emden and de Lucena 1982; Shapiro 1983), the goal is seen as a system of parallel processes. The single process evolves via resolution steps; different clausal definitions for the same predicate provide nondeterminism and the shared variables among processes represent communication channels.

However, in a concurrent system processes *interact* with each other[1], hence a concurrent language should embody some synchronization mechanisms and constructs to specify the dependency of choices upon the environment. Furthermore once a certain choice is selected, the process cannot backtrack. This principle is in contrast with the kind of nondeterminism of logic programming, according to which all possible solutions are pursued (*don't know nondeterminism*).

Since more than a decade there has been an active line of research aimed at extending logic programming with mechanisms for concurrency.

1.1 Don't know versus don't care nondeterminism

In concurrent systems a process cannot 'undo' a choice, even if it reveals to be wrong (for instance because the process gets stuck). The reason is that a choice, and the actions done afterwards, have already influenced the environment: to maintain consistency the whole system should backtrack, but this is too inefficient. It is rather preferable to provide the language with mechanisms to control the choices, so to avoid as far as possible that wrong decisions are taken. The most common such mechanism is the *guard*: a condition associated with each alternative which is tested before selecting that alternative.

Almost all the concurrent logic languages and concurrent constraint languages adopt *don't care nondeterminism*: clauses are provided with a guard part and a process chooses (*commits* to) only one clause, provided its *guard part* is satisfied.

The few languages which maintain don't know nondeterminism, thus preserving the characteristics of logic programming to compute all solutions, are Distributed Logic (Monteiro 1981,1982), the language of Generalized Horn Clauses (Falaschi *et al.* 1984), Delta Prolog (Pereira and Nasr 1984) and the language cc($\downarrow, \rightarrow, \Rightarrow$) (Saraswat 1986, 1989). Andorra Prolog (Haridi and Brand 1988; Haridi 1990; Haridi and Janson 1990, 1991; Haridi *et al.* 1992) combines don't care and don't know nondeterminism.

1.2 Synchronization and communication mechanisms

Roughly, a synchronization mechanism specifies when a process can be activated. In many concurrent paradigms synchronization is in symbiosis with communication. This is the case also in most concurrent logic languages. Some of the first concurrent logic languages, the Relational Language (Clark and Gregory 1981), Concurrent Prolog (Shapiro 1983, 1986, 1988), the language of Guarded Horn Clauses (Ueda 1987, 1988) and PARLOG (Clark and Gregory 1986; Gregory 1987), enforced synchroniza-

[1]The dichotomy sequential programming/concurrent programming is closely related to the distinction between transformational systems and interactive (or reactive) systems, see (Harel and Pnueli 1985).

tion by introducing a directionality on the communication channels. In this way, some processes are seen as *producers* (of bindings) on a certain variable, others as *consumers*. The advantage of such a solution is that the resulting language is quite 'faithful' to the (parallel) execution model of logic programming: only the unification mechanism needs to be modified.

The kind of communication mechanism described above is asynchronous[2], in the sense that the process producing a binding and the process consuming (or, more precisely, testing) the binding, perform their actions at different times. As opposite to this principle, synchronous communication requires the partners to exchange the information *simultaneously*. This latter mechanism has been adopted in Distributed Logic, Generalized Horn Clauses, Delta Prolog and the language of Communicating Clauses (Jacquet and Monteiro 1990, 1991, 1992).

A rather different approach to synchronization has been proposed in P-Prolog (Yang 1986; Yang and Aiso 1986). The idea is to suspend a process until enough bindings are produced so that the number of consistent alternatives for the next step reduces to one. This solution is semantically clean, in the sense that it does not modify the declarative reading of the language. Of course, the disadvantage is that only deterministic processes can be specified in this way. Such a mechanism is also embodied in Andorra Prolog and in the ALPS languages (Maher 1987).

1.3 The influence of Constraint Logic Programming

The ALPS paradigm was introduced as the concurrent version of Constraint Logic Programming (Jaffar and Lassez 1987). The main novelty was the *validation* case for the commitment rule. In short, this rule allows a process to make a derivation step using a certain clause only if the constraint established till that moment *entails* the constraint associated with the guard of the clause. Essentially this is the 'natural evolution' of the GHC commitment rule, but the way it is formulated is much more elegant and combines more naturally with the underlying logic paradigm.

This intuition had a strong impact on the field: it was the precursor of the ask-rule, which is one of the central ideas of concurrent constraint programming and which has become the basic synchronization mechanism of concurrent logic/constraint languages.

1.4 Concurrent constraint programming

Concurrent constraint programming (ccp) (Saraswat 1989; Saraswat and Rinard 1990; Saraswat *et al.* 1991) presents a new perspective on the underlying philosophy of logic programming. In fact it is based on extra-

[2]The terminology *synchronous/asynchronous communication* must not be confused with *synchronization* in the sense of suspending/resuming processes. A synchronization mechanism can be based on asynchronous communication.

logical *operators* typical of the concurrent paradigms, like CCS (Milner 1980), TCSP (Brookes *et al.* 1984) and ACP (Bergstra and Klop 1984, 1986); in particular, the *choice* ($+$), the *action prefixing* ($\rightarrow$), and the *hiding* operator ($\exists$). Additionally, concurrent constraint programming embodies explicit mechanisms for communication and synchronization consisting of two kinds of actions, *ask* and *tell*. Also in the concurrent logic languages these control features were present, but they were 'hidden' in various ways, as explained in Section 1.

There are many advantages in an explicit representation of these concurrency control mechanisms by means of operators. First of all, it is the basis for the definition of a calculus and for an algebraic treatment of processes. Furthermore, the explicit representation allows us to separate each mechanism from the others, thus permitting a better understanding of the laws of its behavior. For instance, one of the main problems in studying the semantics of concurrent logic programming is that the choice mechanism is combined with recursion, since a clause is in general a recursive definition. Another advantage is that the standard tools developed in the theory of concurrency can be applied more easily. Finally, a 'reconciliation' with the declarative principles of logic programming is made more feasible. For instance, the renaming mechanism and the behavior of the *ask* and *tell* actions can be described in a logical way.

Concurrent constraint programming has become rapidly very popular and is regarded as the reference formalism in this area.

1.5 Semantics

Compared to the pure logic and constraint languages, the introduction of the various synchronization and control mechanisms induces a more complicated semantics. At the operational level, these constructs can be appropriately described by means of a *transition system* in the style of SOS (Plotkin 1981), see for instance (Saraswat 1987a, 1987b).

However, problems arise when developing semantics which account for the declarative nature. There are various differences with respect to the pure languages, which make problematic a simple extension of the standard model-theoretic approach. First of all, due to the presence of the synchronization mechanisms, not all the computation steps are allowed and therefore the *success set* becomes smaller. Levi and Palamidessi (1985, 1987) developed a model-theoretic semantics for characterizing the success set of concurrent logic languages. The basic idea was to annotate the data structures in order to model the producer/consumer discipline, thus obtaining a sort of input-output semantics. However not all kinds of concurrent computations could be modeled in this way.

Another problem arises from the don't care nondeterminism: the concept of failure changes from *universal* to *existential*. Furthermore, a new kind of termination must be taken into account: *deadlock*, namely the situation in which all processes are *suspended*, waiting for the conditions for synchronization to become validated. Note that don't care nondeterminism means absence of backtracking, hence non-successful computations are particularly relevant in interactive systems: a system which can deadlock/fail cannot be considered equivalent to a system deadlock/failure free, even if the set of successful computations is the same.

The characterization of non-successful computations requires even more complicated semantic structures and operators, which are difficult to bring into accordance with the logical flavor. However, if it is problematic to maintain the logical flavor, yet we can preserve the other important aspect of the model-theoretic semantics: the compositionality. This amounts to shift to a denotational approach.

The first proposals of denotational semantics for concurrent logic languages were inspired by the classical concepts of concurrency theory. In particular, concerning the choice of the semantical domains: synchronization trees and failure sets. For example, De Bakker and Kok (1988,1990) and de Boer *et al.* (1989,1991) used tree-like structures labeled with functions; Gabbrielli and Levi (1990) considered trees labeled with constraints; Saraswat and Rinard (1990) used similar structures modulo equivalence relations based on bisimulation; Gerth *et al.* (1988) and Gaifman *et al.* (1989) used refusal pairs like in the failure semantics (Brookes *et al.* 1984), although in (Gerth *et al.* 1988) the refusal set is reduced to be a singleton.

Tree-like structures, bisimulation and failure semantics encode, in various degrees of abstraction, the branching structure of a process. Such information is needed, in CCS, TCSP, ACP, and in languages with *synchronous* communication in general, to represent the dependency of a choice upon the environment, hence to model compositionally the choice and the parallel operators.

On the other hand, in concurrent logic programming and in concurrent constraint programming communication is *asynchronous*. Therefore those branching structures are actually not needed: which actions are enabled does not depend upon the current state of the environment, but only upon the store. In a transition system this can be made explicit by adding a *passive* rule that does not exist in the classical concurrent paradigms: an *arbitrary* assumption of a step made by the environment. This amounts to considering all the possible interactions between the given process and arbitrary environments, and it leads to a simple compositional semantics, consisting of sequences of constraints labeled by assume/tell modes (de Boer and Palamidessi 1990a, 1991a). In this framework the parallel combination corresponds to 'unzip' sequences, so that the assumptions of a process match with the actions of the other, and vice-versa. These kind of sequences (*reactive sequences*) were already one component of the semantics in (Gerth *et al.* 1988) and in (Gaifman *et al.* 1989), the other component being the refusal information.

Independently, an intriguing approach was developed by Saraswat *et al.* (1991). The basic idea consists of denoting processes by Scott's closure operators, which have the nice property of being representable via the set of their fixpoints. The operators of the language can then be described as operations on those sets. In particular, parallelism can be modeled simply by intersection. This semantics abstracts from deadlock, however it should be possible to extend it so to model deadlock as well.

The approach of Saraswat *et al.* (1991) is suitable for languages without atomic unification, i.e. languages which do not support the consistency check as a guard of a choice. In this context, the approach of de Boer and Palamidessi (1991a) is equivalent. The semantics of Saraswat *et al.* is more elegant as it exploits the monotonic evolution of the computations; on the other hand the approach of de Boer and Palamidessi is more general because it can be applied also to languages with atomic unification. Furthermore it can be easily adapted to any extension of ccp, provided that we remain in the framework of asynchronous communication (see de Boer *et al.* 1993).

1.6 Current trends

The application of category theory has brought new insight in the semantics of ccp. Panangaden *et al.* (1992) used the framework of hyperdoctrines to model the entailment relation. Saraswat (1992) showed that constraint systems form a Cartesian-closed category, thus establishing the basis for the solution of recursive domain equations and higher-order constraint languages.

Saraswat and Lincoln (1992) have developed a variant of ccp based on linear logic. Linear logic is particularly suitable to deal with concrete aspects of the computation, such as the consumption of resources. Furthermore, it seems to be an appropriate basis for the integration of logic programming and object-oriented programming (Andreoli and Pareschi 1991).

De Boer and Palamidessi (1992) have made a first step towards the algebraic axiomatization of ccp. Process algebra is one of the most widespread methods in concurrency theory both for specification and for program verification, and it has been intensively investigated in the synchronous case. In the asynchronous case however a lot of work remains to be done: modeling various notions of observables and capturing by axioms the specificity of the communication mechanisms of the various languages.

Recently there has been a lot of interest concerning expressiveness questions. Shapiro (1991, 1992) and de Boer and Palamidessi (1990b, 1991b, 1994) have done some preliminary work aimed at the development of a general theory to compare the expressive power of concurrent languages in general, and cc languages in particular. Such studies are meant to provide a formal basis of the design principles and implementation techniques.

Another interesting line of research is the application of abstract interpretation techniques to the analysis of ccp. A compositional approach to this problem has been developed by Falaschi *et al.* (1993). The idea is to consider a denotational semantics which correctly approximate the observables, without being correct in the classical sense. This allows us to consider simpler semantic structures which make the analysis more efficient.

An efficient distributed implementation of ccp requires the development of non-interleaving models. Montanari and Rossi (1991) have developed a first proposal for a 'true concurrent' model. Such a model is based on *graph grammars*: the inference rules of the underlying constraint system and the (user specified) agents are expressed as *graph productions*. A computation is a sequence of graph rewriting steps, and it is possible to associate to it an *occurrence net*, a kind of Petri Net which expresses *causality information*.

1.7 Plan of the paper

For various reasons, which include implementability considerations and smooth integration with the logical paradigm, the asynchronous approach to communication, based on shared variables, has become more popular and more widespread in the design of concurrent logic languages. Therefore this paper will focus on the class of asynchronous languages.

The paper is organized as follows. Next section contains some preliminaries about logic programming. Section 3 presents the main lines along which the field of concurrent logic programming has developed; the motivations, the main ideas, the various proposals for a right balance between expressivity and faithfulness to the logical paradigm. In order to point out similarities and differences of the various languages, we try to be as uniform as possible in the formal description of their operational semantics, given via a transition system.

In Chapter 4 we review the innovative ideas introduced by the ALPS paradigm. Finally, in Section 5 the concurrent constraint paradigm is presented. For this class of languages we study in detail the denotational semantics. This will provide also a model for the concurrent logic languages, since concurrent constraint programming can be seen as a generalization of (most of) them. We review and compare the two main denotational approaches presented in literature, one based on reactive sequences, the other one on Scott's closure operators.

2 Preliminaries.

In this section we recall the terminology and the basic definitions of logic programming. For more details we refer to (Lloyd 1987; Apt 1990).

We assume given a finite set of *data constructors* $a, b, c \ldots$ (constant symbols), $f, g, h \ldots$ (function symbols), a finite set of *predicate symbols* $p, q, r \ldots$ and a denumerable set *Var* of *variable symbols* $x, y, z \ldots$. *Terms* $t, u \ldots$ are constructed from data constructors and variables in the usual way. *Atoms* $A, B \ldots$ are objects of the form $p(t_1, \ldots, t_n)$. Terms and atoms are called *ground* if they don't contain occurrences of variables. A *substitution* ϑ is a mapping from variables to terms such that the set $dom(\vartheta) = \{x \mid \vartheta(x) \neq x\}$ (*domain* of ϑ) is finite. The empty substitution (identity) is denoted by ϵ. The term $t\vartheta$ results from simultaneously replacing in t every variable x by $\vartheta(x)$. Thus substitutions can be lifted to functions from terms to terms. The composition $\vartheta\sigma$ of the substitutions ϑ and σ is defined as the functional composition. The pre-order $\leq$ on substitutions is defined by $\vartheta \leq \sigma$ iff there exists ϑ' such that $\vartheta\vartheta' = \sigma$. We denote by $\simeq$ the associated equivalence relation. The application of a substitution to an atom is defined as $p(t_1, \ldots, t_n)\vartheta = p(t_1\vartheta, \ldots, t_n\vartheta)$. Given two atoms A and B, their most general unifier (unique modulo $\simeq$) $mgu(A, B)$ is the minimal substitution ϑ such that $A\vartheta = B\vartheta$.

A *definite clause* C is a formula $H \leftarrow B_1, \ldots, B_n$ $(n \geq 0)$, where H and the B_i's are atoms; H is the *head* of the clause and the sequence $B_1, \ldots, B_n$ is the *body*. If the body is empty then the clause is called a *unit clause*. The symbols '$\leftarrow$' and ',' denote logical implication and logical conjunction respectively. Clauses are assumed to be universally closed. A *program* is a finite set of definite clauses $P = \{C_1, \ldots, C_n\}$. A *goal statement* G is a formula $\leftarrow A_1, \ldots, A_m$, where each A_i is an atom.

The standard operational semantics is defined as follows. A *derivation step* is a goal transformation

$$\leftarrow A_1, \ldots, A_i, \ldots, A_m \stackrel{\vartheta}{\longmapsto} \leftarrow (A_1, \ldots, A_{i_1}, B_1, \ldots B_n, A_{i+1}, \ldots, A_m)\vartheta$$

where $H \leftarrow B_1, \ldots B_n$ is a clause in P, such that $\vartheta = mgu(A_i, H)$. The clause must be renamed in order to avoid variable clashes with the goal (*renaming apart*). A *derivation* of $P \cup \{G\}$ is a possibly infinite sequence of goals $G_1, \ldots, G_n, \ldots$, where $G = G_1$, such that for every subsequent goals G_i, G_{i+1} we have $G_i \stackrel{\vartheta_i}{\longmapsto} G_{i+1}$ for some substitution ϑ_i. A derivation is *successful* if it is finite and it terminates with the empty goal ($\Box$). The composition of all the associated substitutions is called *computed answer substitution*. A derivation *fails* if it is finite, maximal (i.e. it is not the prefix of a longer derivation) and it is not successful. A derivation is *fair* if it is either finite or every atom which occurs in some goal of the derivation is eventually selected for a derivation step.

The success set and finite failure set of a program P are defined[3] respectively by

$$\begin{aligned} SS &= \{A \mid A \text{ is ground and } P \cup \{\leftarrow A\} \text{ has a refutation}\}, \\ FF &= \{A \mid A \text{ is ground and all fair derivations of } P \cup \{\leftarrow A\} \text{ fail}\}. \end{aligned}$$

[3] The failure set is usually defined in terms of a finitely failing SLD-tree; however, the definition that we consider here is equivalent to the standard one (Lassez and Maher, 1984).

3 Concurrent logic languages

In this section we discuss various approaches to turn logic programming into a language suitable for concurrent systems. First we consider the so-called *process interpretation* of the pure language, and then we show what mechanisms have to be added in order to model the basic concepts of concurrency.

3.1 Process interpretation of logic programming

A derivation in logic programming can be viewed as the evolution of a dynamic network of parallel processes (van Emden and de Lucena 1982; Shapiro 1983). An atom in the goal corresponds to a process and a variable occurring in two or more atoms represents a communication channel between the corresponding processes. Thus a goal represents a network of parallel processes connected by communication channels. Communication is modeled by the instantiation of the variables during unification. The resolution step gives rise to a reconfiguration of the network by expanding a process (the selected atom) into a subnetwork of parallel processes (the atoms of the body of the selected clause), and establishing the links on the communication channels. Note that a variable can be instantiated to a term which contains other variables, hence also the connection structure can be modified.

We formalize the above described process view in terms of a *transition system*. In the following a sequence of objects (atoms, terms, variables etc.) $\chi_1, \ldots, \chi_n$ is denoted by $\bar{\chi}$.

Definition 3.1. (A transition system for logic programming) *The process semantics of logic programming is described by a transition system whose configurations are pairs $\langle G; \vartheta \rangle$, where G is a goal and ϑ is a substitution. The transition relation is defined by the rules described in Table 1.*

In the configurations we use the delimiter ';' instead of ',' to avoid confusion with the ',' separating the atoms in a goal. We will use the notations $\longrightarrow^*$ to denote the transitive reflexive closure of the transition relation $\longrightarrow$, and $\not\longrightarrow$ to indicate the absence of any further transition.

The transition system models the inherent parallelism of logic programs in terms of processes interacting via a common datastructure, i.e. the accumulated set of bindings. A configuration $\langle G; \vartheta \rangle$ specifies the goal (or network) $G\vartheta$, where G represents the *generic structure* of the process part and ϑ the accumulated set of bindings. This distinction in the configuration is introduced in order to model the communication in the parallel rule **C2**. We call the components of $G\vartheta$ *instances* (of the corresponding components of G), or, more simply, *processes*.

Rule **C1** describes the resolution step of a single process; P_{ren} is the closure of the program with respect to variable renaming. Rule **C2** models

C1	$\langle \leftarrow A; \vartheta\rangle \longrightarrow \langle \leftarrow \bar{B}; \vartheta\sigma\rangle$	*there exists* $A' \leftarrow \bar{B} \in P_{ren}$ *s.t.* $\sigma = mgu(A\vartheta, A')$
C2	$\dfrac{\langle \leftarrow \bar{A}; \vartheta\rangle \longrightarrow \langle \leftarrow \bar{A}'; \vartheta'\rangle}{\begin{array}{c}\langle \leftarrow \bar{A}, \bar{B}; \vartheta\rangle \longrightarrow \langle \leftarrow \bar{A}', \bar{B}; \vartheta'\rangle \\ \langle \leftarrow \bar{B}, \bar{A}; \vartheta\rangle \longrightarrow \langle \leftarrow \bar{B}, \bar{A}'; \vartheta'\rangle\end{array}}$	

Table 1 The transition system for logic programming

the parallel execution of the processes in an *interleaved* manner; additionally it allows for the transmission of the bindings produced by a single process to the common datastructure, hence to the parallel processes. In order to cope properly with the renaming mechanism of logic programming in this parallel setting, we assume that the variables of the renamed clause used for the resolution step do not occur either in $\bar{A}$, $\bar{B}$ or ϑ. The interested reader can find in (Saraswat and Rinard 1990) and in (de Boer and Palamidessi 1991a) formal solutions to this problem. Furthermore we assume that only one possible renaming is considered in **C1**; as a consequence we have finite-branching nondeterminism.

From the above described operational model we can abstract various other semantics, depending on what we want to *observe* of a process. At first we consider a very simple notion of observables, corresponding to the standard operational semantics of logic programming. We associate to each goal the set of substitutions resulting from successfully terminating computations, and *ff* if the process fails. A computation is *fair* if it is either finite or each of the atoms in the configurations is eventually involved in the application of Rule **C1**.

Definition 3.2. (Observables) *Given a program P, for every goal G we define*[4]

$$\mathcal{O}_{ss}(G) = \{\vartheta_{\restriction G} \mid \langle G; \epsilon\rangle \longrightarrow^* \langle \Box; \vartheta\rangle\},$$

$$\mathcal{O}_{ff}(G) = \{ff \mid \textit{for all fair computation starting from } \langle G;\epsilon\rangle \textit{ there exists } G' \textit{ s.t. } \langle G;\epsilon\rangle \longrightarrow^* \langle G';\vartheta\rangle \not\longrightarrow, \textit{ and } G' \neq \Box \textit{ holds}\}$$

[4] Equivalently, $\mathcal{O}_{ff}(G)$ could be defined as a boolean. We have chosen the set notation for the sake of uniformity with the success case and with more complicated notions of failure which will be considered later.

where $\vartheta_{|G}$ indicates the restriction of the domain of ϑ to the variables of G.

Note that $\mathcal{O}_{ss}(G) = \emptyset$ does not imply $\mathit{ff} \in \mathcal{O}_{\mathit{ff}}(G)$; this is due to the possible presence of infinite fair computations. The standard notions of success set and failure set can be retrieved from the above defined observables as follows:

$$\begin{aligned} SS = \quad & \{A \mid A \text{ is ground and } \epsilon \in \mathcal{O}_{ss}(\leftarrow A)\}, \\ FF = \quad & \{A \mid A \text{ is ground and } \mathit{ff} \in \mathcal{O}_{\mathit{ff}}(\leftarrow A)\}. \end{aligned}$$

Next we show that the success component of the observables is compositional with respect to the parallel operator (i.e. the logical conjunction). In other words, the substitutions resulting from the execution of a network of processes can be obtained by executing the single processes independently, and then combining their results. This should be contrasted with the case of imperative parallel languages, in which compositionality requires information about the intermediate states of the computations, and can be interpreted as a formal justification of the claim that logic programming is inherently parallel. The main reason of this difference is that in logic programming the state (i.e. the accumulated set of bindings) evolves monotonically. In the presence of non-monotonic predicates, like *var*, this result does not hold for logic programming either, see (Apt *et al.* 1992).

Compositionality is the basis of a denotational approach to the semantics, which will be treated in Section 5.4. In order to show the compositionality of $\mathcal{O}_{ss}$ it is convenient to consider a *parallel composition* operator on substitutions (Palamidessi 1990; Jacquet 1991), which combines the bindings on shared variables produced independently by parallel processes.

A substitution ϑ is naturally associated with a set of equations

$$\mathcal{E}(\vartheta) = \{x = t \mid x \in dom(\vartheta) \text{ and } \vartheta(x) = t\}.$$

Given a set of equations $E = \{t_1 = u_1, \ldots, t_n = u_n\}$, we extend the notion of most general unifier by defining $mgu(E)$ as the least (up to $\simeq$) substitution ϑ such that for each $i \in [1, n]$ we have $t_i\vartheta = u_i\vartheta$. If such a substitution does not exist then we define $mgu(E) = \mathit{false}$.

Definition 3.3. *The parallel composition of two substitutions ϑ and σ is defined as*

$$\vartheta \parallel \sigma = mgu(\mathcal{E}(\vartheta) \cup \mathcal{E}(\sigma)).$$

Proposition 3.4. (Compositionality of $\mathcal{O}_{ss}$, *(de Boer et al. 1992)***)** *For any goal $\leftarrow \bar{A}, \bar{B}$*

$$\mathcal{O}_{ss}(\leftarrow \bar{A}, \bar{B}) = \mathcal{O}_{ss}(\leftarrow \bar{A}) \,\tilde{\parallel}_{ss}\, \mathcal{O}_{ss}(\leftarrow \bar{B})$$

where for sets of substitutions X and Y

$$X \mathbin{\tilde{\|}_{ss}} Y = \{\vartheta \parallel \sigma \mid \vartheta \in X,\ \sigma \in Y \ \textit{and}\ \vartheta \parallel \sigma \neq \textit{false}\}.$$

The simplicity of this result (compositionality of $\mathcal{O}_{ss}$) is due essentially to (a) the already mentioned monotonic evolution of the state, and (b) the absence of any suspension mechanism for process synchronization. In the following sections we will show that such a synchronization mechanism requires more elaborate semantic structures.

Concerning the failure component of the semantics, the following example shows that it is not compositional.

Example 3.5. *Consider the program*

$$p(a) \leftarrow p(a),$$
$$q(b) \leftarrow q(b),$$
$$r(a) \leftarrow .$$

We have that $\leftarrow p(x)$ and $\leftarrow q(x)$ are observationally equivalent, since $\mathcal{O}_{ss}(\leftarrow p(x)) = \mathcal{O}_{ss}(\leftarrow q(x)) = \emptyset$ and $\mathcal{O}_{ff}(\leftarrow p(x)) = \mathcal{O}_{ff}(\leftarrow q(x)) = \emptyset$. On the other hand, when $p(x)$ and $q(x)$ are inserted in a parallel context containing $r(x)$ the equivalence is lost: $\mathcal{O}_{ff}(\leftarrow p(x), r(x)) = \emptyset$ while $\mathcal{O}_{ff}(\leftarrow q(x), r(x)) = \{ff\}$.

Full compositionality can be achieved by observing also the 'limit results' computed by the infinite derivations. In short, the idea is the following: first turn the domain of substitutions into a CPO (by using one of the standard completion techniques), then extend the notions of restriction and *mgu* to the infinite elements (Levi *et al.* 1990), finally define the infinite component of the observables $\mathcal{O}_{ii}$ as follows

$$\begin{aligned}\mathcal{O}_{ii}(G) = \ \{(lub_n \vartheta_n)_{\restriction G} \mid \ & \text{there exists an infinite fair computation} \\ & \langle G; \epsilon\rangle \longrightarrow \langle G_1; \vartheta_1\rangle \longrightarrow \ldots \langle G_n; \vartheta_n\rangle \longrightarrow \ldots\}.\end{aligned}$$

In the above definition the infinite computations are required to be fair essentially because also the failures are defined in terms of fair computations. Arbitrary infinite computations would not provide the necessary information to model failure compositionally. The reader is invited to find an example of two processes having the same results when arbitrary infinite computations are also considered, and which behave differently with respect to failure when put in parallel with another process.

With this new definition, the observables are fully compositional. For the failure component we have:

$$\mathcal{O}_{ff}(\leftarrow \bar{A}, \bar{B}) = \mathcal{O}(\leftarrow \bar{A}) \mathbin{\tilde{\|}_{ff}} \mathcal{O}(\leftarrow \bar{B})$$

where $\mathcal{O}(G) = \mathcal{O}_{ss}(G) \cup \mathcal{O}_{\mathit{ff}}(G) \cup \mathcal{O}_{ii}(G)$ and

$$\begin{array}{rcll} X \,\tilde{\|}_{\mathit{ff}}\, Y & = & \{\mathit{ff} \mid & \mathit{ff} \in X \cup Y \text{ or} \\ & & & \forall\vartheta \in X\ \forall\sigma \in Y\ \vartheta \,\|\, \sigma = \mathit{false}\}. \end{array}$$

For the infinite component:

$$\mathcal{O}_{ii}(\leftarrow \bar{A}, \bar{B}) = \langle \mathcal{O}_{ss}(\leftarrow \bar{A}), \mathcal{O}_{ii}(\leftarrow \bar{A})\rangle \,\tilde{\|}_{ii}\, \langle \mathcal{O}_{ss}(\leftarrow \bar{B}), \mathcal{O}_{ii}(\leftarrow \bar{B})\rangle,$$

where

$$\begin{array}{rcll} \langle X_1, X_2\rangle \,\tilde{\|}_{ii}\, \langle Y_1, Y_2\rangle & = & \{\vartheta \,\|\, \sigma \mid & \vartheta \in X_2, \sigma \in Y_1 \cup Y_2 \\ & & & \text{and } \vartheta \,\|\, \sigma \neq \mathit{false}\ \} \\ & \cup & \{\vartheta \,\|\, \sigma \mid & \vartheta \in X_1 \cup X_2, \sigma \in Y_2 \\ & & & \text{and } \vartheta \,\|\, \sigma \neq \mathit{false}\ \}. \end{array}$$

Example 3.6. *For the program in Example 3.5 we have* $\mathcal{O}_{ii}(\leftarrow p(x)) = \{\{x/a\}\}$ *whereas* $\mathcal{O}_{ii}(\leftarrow q(x)) = \{\{x/b\}\}$.

3.2 Explicit mechanisms for concurrency

We describe now the main characteristics of concurrent logic languages.

3.2.1 *Don't care nondeterminism*

The kind of nondeterminism in logic programming is *don't know*, like in (nondeterministic) Turing machines and finite automata, i.e. it is meant for exhaustive search algorithms. Failure is *universal*, namely failing computations are discarded if there are successful ones. Therefore, the implementation of don't know nondeterminism requires some backtracking mechanism.

Concurrent systems imply a different notion: nondeterminism is needed for coping with unpredictable circumstances, like the relative speed of the various processes, and the interaction with the external world. A concurrent process is required to select one of the alternatives which are enabled at a certain moment, and to *commit* to that choice. Note that this induces an *existential* notion of failure. In any case, it seems not to make much sense for a process to backtrack, because it would involve undoing actions that have already influenced the evolution of the environment. This kind of nondeterminism is called *don't care*.

From a formal point of view, the computational model of logic programming with don't care nondeterminism is still expressed by the transition system described in Table 1. The notion of successful computation remains the same. What changes is the notion of failure, which becomes existential:

$$\begin{array}{rcll} \mathcal{O}_{\mathit{ff}}(G) & = & \{\mathit{ff} \mid & \text{there exist } G', \vartheta \text{ such that} \\ & & & \langle G; \epsilon\rangle \longrightarrow^* \langle G'; \vartheta\rangle \not\longrightarrow \text{ and } G' \neq \Box\}. \end{array}$$

Note that with don't care nondeterminism the presence of *ff* in the semantics of a process does not exclude successful or infinite computations.

This notion of failure is not compositional. The reason is that existential failure depends on the *branching structure* of a process, namely the tree which represents the alternative computations of a process, where a choice point indicates the selection of an atom and of a clause for the resolution step.

Example 3.7. *Consider the program*

$$\begin{aligned}
&p(a) \leftarrow, \\
&p(b) \leftarrow, \\
&q(x) \leftarrow r_1(x), \\
&q(x) \leftarrow r_2(x), \\
&r_1(a) \leftarrow, \\
&r_2(b) \leftarrow .
\end{aligned}$$

Consider the network $\leftarrow p(x), p(x)$. *Whatever binding* ($\{x/a\}$ *or* $\{x/b\}$) *is generated by one of the two p's processes, the other one will select the right clause accordingly. Therefore failure is not possible:* $\mathcal{O}_{ff}(\leftarrow p(x), p(x)) = \emptyset$. *Consider now the network* $\leftarrow p(x), q(x)$. *The choice of the clause for q is blind, hence after p has produced a binding, say,* $\{x/a\}$, *q can still make the wrong choice by selecting the clause* $q(x) \leftarrow r_2(x)$, *which brings to failure. Therefore* $\mathcal{O}_{ff}(\leftarrow p(x), q(x)) = \{ff\}$.

The difference in the choice structure between $\leftarrow p(x)$ *and* $\leftarrow q(x)$ *is not reflected by the observables. In fact* $\leftarrow p(x)$ *and* $\leftarrow q(x)$ *are observationally equivalent, since* $\mathcal{O}_{ss}(\leftarrow p(x)) = \mathcal{O}_{ss}(\leftarrow q(x)) = \{\{x/a\}, \{x/b\}\}$, $\mathcal{O}_{ff}(\leftarrow p(x)) = \mathcal{O}_{ff}(\leftarrow q(x)) = \emptyset$ *and* $\mathcal{O}_{ii}(\leftarrow p(x)) = \mathcal{O}_{ii}(\leftarrow q(x)) = \emptyset$.

In Section 5.4 we will see how to encode the relevant information of the branching structure so to achieve compositionality.

Since the main feature of reactive systems is the possibly ongoing interaction with the environment, the infinite behavior is of interest in its own.

$$\begin{aligned}
\mathcal{O}_{ii}(G) = \ \{(lub_n \vartheta_n)_{|G} \mid \ & \text{there exists an infinite computation} \\
& \langle G; \epsilon\rangle \longrightarrow \langle G_1; \vartheta_1\rangle \longrightarrow \ldots \langle G_n; \vartheta_n\rangle \longrightarrow \ldots\}.
\end{aligned}$$

3.2.2 *Control of nondeterminism*

Don't care nondeterminism is usually combined with the notion of *guard* (Dijkstra 1986), which allows controlling the choices. In general, guards are

C1	$\langle \leftarrow A; \vartheta \rangle \longrightarrow \langle \leftarrow \bar{B}; \vartheta\sigma\eta \rangle$	$A' \leftarrow \bar{g} \mid \bar{B} \in P_{ren}$, $\sigma = mgu(A\vartheta, A')$ and $eval(\bar{g}\sigma) = true$ with substitution η

Table 2 The resolution rule for flat guards

conditions associated with the various alternatives in a choice construct. An alternative can be selected (is enabled) only if the associated guard is satisfied. In other words, it is a sort of conditional statement. In a concurrent setting, the guarded choice is often combined with a synchronization mechanism, i.e. the conditions are influenced by the other processes.

The introduction of guards in logic programming leads to the concept of *guarded clause*, which is an expression of the form

$$A \leftarrow \bar{g} \mid \bar{B},$$

where $\bar{g}$ is the guard, consisting of a sequence of atoms; '|' is the *commit* operator, and $\bar{B}$ is the body.

The guard $\bar{g}$ can contain built-in and user-defined predicates. In case there are only built-in's, $\bar{g}$ is *flat*, otherwise it is *deep*.

Built-in predicates are for instance the Prolog built-in's. However, we abstract here from their specific nature, and we only assume the presence of some mechanism *eval* embedded in the language which evaluates each sequence of built-in atoms $\bar{C}$ either to *true* or to *false* and in the first case it delivers a (possibly empty) substitution.

In the flat case, the operational model of guarded clauses can be obtained by a simple modification of the system described in Table 1: we only need to add in Rule **C1** the condition on the guard. The new rule is described in Table 2. The semantics $\mathcal{O}_{ss}$, $\mathcal{O}_{ff}$, and $\mathcal{O}_{ii}$ are defined as before ($\mathcal{O}_{ff}$ being of course the one specified for don't care nondeterminism).

Let's now consider deep guards. A first idea would be to replace the condition $eval(\bar{g}\sigma) = true$ *with substitution* η in **C1** by the premise $\langle \leftarrow \bar{g}\sigma; \epsilon \rangle \longrightarrow^* \langle \Box; \eta \rangle$. This corresponds to the intended meaning of deep guards, and it is sufficient to describe successful computations. On the other hand, when we consider failure, things become more complicated. Concurrent logic languages with deep guards adopt a 'deep' notion of failure, i.e. failure can be caused also by a failure in the guard evaluation.

Example 3.8. *Consider the program*

$$p \leftarrow q \mid,$$
$$q \leftarrow \mid,$$
$$q \leftarrow \mid r.$$

In the intended model $\leftarrow p$ *has the possibility to fail, which happens when the resolution of the guard* $\leftarrow q$ *commits to the third clause. This failure is however not captured by the semantics described above: the configuration* $\langle \leftarrow p; \epsilon \rangle$ *is not stuck since the guard has also the possibility to succeed.*

An analogous problem arises for infinite computations.

Example 3.9. *Consider the programs*

$$\begin{array}{ll} P_1 & p \leftarrow q \mid, \\ & q \leftarrow q \mid, \\ & \\ P_2 & p \leftarrow q \mid, \\ & q \leftarrow \mid, \\ & q \leftarrow q \mid . \end{array}$$

In P_1 *the evaluation of the guard* $\leftarrow q$ *diverges; in the semantics described above* $\langle \leftarrow p; \epsilon \rangle$ *has no outgoing transitions. In* P_2 *the guard* $\leftarrow q$ *can still diverge, but this possibility is not reflected in the above semantics:* $\langle \leftarrow p; \epsilon \rangle$ *has an outgoing transitions corresponding to the successful derivation of* $\langle \leftarrow q; \epsilon \rangle$ *by using the second clause.*

The above examples show that the definition of $\mathcal{O}_{ff}$ and $\mathcal{O}_{ii}$ require a much more elaborate formulation. However for the case of failure we can obtain a concise description by adding some additional transition rules. In Tables 3 and 4 we give the rules for success and failure; for a treatment of divergence we refer to (Knijnenburg and Kok 1989). For further discussion on the operational semantics of deep guards see (Saraswat 1987a) and (Levy 1988).

The observables $\mathcal{O}_{ss}$ are defined as usual; for $\mathcal{O}_{ff}$ we have

$$\mathcal{O}_{ff}(G) \quad = \quad \{ff \quad \mid \text{there exists } \vartheta \text{ such that } \langle G; \epsilon \rangle \longrightarrow^* \langle ff; \vartheta \rangle\}.$$

Most of the deep concurrent logic languages adopt the principle that guards are evaluated in a *fair manner*. Roughly, this means that in the derivation step all guards are tried in parallel; if one guard is successful and the others are not then the former is eventually selected. The following

C1	$\langle \leftarrow A; \vartheta \rangle \longrightarrow \langle \Box; \vartheta\eta \rangle$	A is a built-in and $eval(A\vartheta) = true$ with substitution η
C2	$\dfrac{\langle \leftarrow \bar{g}\sigma; \epsilon \rangle \longrightarrow^* \langle \Box; \eta \rangle}{\langle \leftarrow A; \vartheta \rangle \longrightarrow \langle \leftarrow \bar{B}; \vartheta\sigma\eta \rangle}$	there exists A' s.t. $A' \leftarrow \bar{g} \mid \bar{B} \in P_{ren}$ and $\sigma = mgu(A\vartheta, A')$
C3	$\dfrac{\langle \leftarrow \bar{A}; \vartheta \rangle \longrightarrow \langle \leftarrow \bar{A}'; \vartheta' \rangle}{\begin{array}{c}\langle \leftarrow \bar{A}, \bar{B}; \vartheta \rangle \longrightarrow \langle \leftarrow \bar{A}', \bar{B}; \vartheta' \rangle \\ \langle \leftarrow \bar{B}, \bar{A}; \vartheta \rangle \longrightarrow \langle \leftarrow \bar{B}, \bar{A}'; \vartheta' \rangle\end{array}}$	

Table 3 Deep guards: computation rules

is one of the typical examples used by Shapiro (1983) to illustrate the usefulness of fair evaluation of guards.

Example 3.10. (Interrupt-handler in a Unix-like shell) *Consider an interactive system consisting of a process user and a process shell (see Figure 1). The user sends a stream (i.e. a possibly infinite list) l of commands to the shell, and the shell executes them. If the stream contains an interrupt, namely a command of the form $''\uparrow c''$ (control_c), then the shell must abort the current execution, flush the portion of stream up to the element $''\uparrow c''$, and resume the execution from there.*

Using the notation $[x \mid l]$ to represent the list whose head is x and whose tail is l, the program for shell can be written as follows

$$shell([x \mid l]) \leftarrow command(x) \mid shell(l),$$

$$shell(l) \leftarrow control\_c(l, l') \mid shell(l'),$$

$$control\_c([x \mid l], l') \leftarrow x \neq {''\uparrow c''} \mid control\_c(l, l')\},$$

$$control\_c([{''\uparrow c''} \mid l], l') \leftarrow \mid l = l',$$

$$\ldots$$

where the process $command(x)$ is supposed to execute the command x. The goal $\leftarrow control\_c(l, l')$ searches for an occurrence of $''\uparrow c''$ in l; in case of success, it produces the stream l' of commands following $''\uparrow c''$ in l. A

F1	$\langle \leftarrow A; \vartheta \rangle \longrightarrow \langle ff; \vartheta \rangle$	A is a built-in and $eval(A\vartheta) = false$
F2	$\langle \leftarrow A; \vartheta \rangle \longrightarrow \langle ff; \vartheta \rangle$	for all $A' \leftarrow \bar{g} \mid \bar{B} \in P_{ren}$, $mgu(A\vartheta, A') = false$ or $\langle \leftarrow \bar{g}\ mgu(A\vartheta, A'); \epsilon \rangle \longrightarrow^* \langle ff; \eta \rangle$ for some η
F3	$\dfrac{\langle \leftarrow \bar{A}; \vartheta \rangle \longrightarrow \langle ff; \eta \rangle}{\begin{array}{c}\langle \leftarrow \bar{A}, \bar{B}; \vartheta \rangle \longrightarrow \langle ff; \eta \rangle \\ \langle \leftarrow \bar{B}, \bar{A}; \vartheta \rangle \longrightarrow \langle ff; \eta \rangle\end{array}}$	

Table 4 Deep guards: failure rules

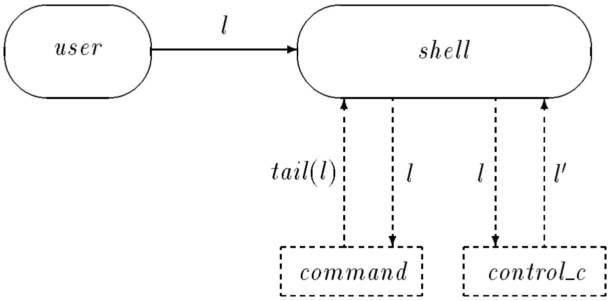

FIG. 1. Interrupt-handler in a Unix-like shell.

call to command(x) may cause an infinite loop; in this case the fairness of the guard evaluation ensures that a possible interrupt sent by the user will eventually be effective.

For further discussion about fairness in concurrent logic programming, and for an interesting idea about the possibility to replace parallel guard evaluation by AND parallelism, see (Saraswat 1986).

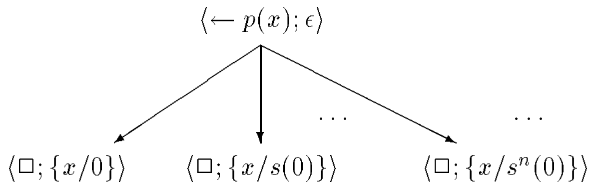

FIG. 2. Transitions in the program with deep guards.

3.2.3 *Deep guards versus flat guards*

Deep guards are clearly more expressive than flat guards. In particular, they permit the implementation of *unbounded nondeterminism*, i.e. choice constructs with infinitely many alternatives. In other words, deep guards allow infinite-branching nondeterminism, in contrast with the finite-branching nondeterminism of flat guards.

Example 3.11. *Consider the program*

$$\begin{aligned} &p(s(x)) \leftarrow p(x) \mid, \\ &p(0) \leftarrow \mid . \end{aligned}$$

From the configuration $\langle \leftarrow p(x); \epsilon \rangle$ infinitely many transitions are possible, see Figure 2.

Note that this process cannot be specified in a language with finite-branching nondeterminism. In fact the transition graph would be a finite-branching tree with infinitely many leaves, hence by König's lemma there would be an infinite derivation. For instance, the following flat version of the program

$$\begin{aligned} &p(s(x)) \leftarrow \mid p(x), \\ &p(0) \leftarrow \mid, \end{aligned}$$

gives rise to the transitions shown in Figure 3.

However, apart from infinite-branching nondeterminism, in most cases deep guards are not necessary. Furthermore, deep guards are quite difficult to implement, especially in the presence of synchronization, and particularly when atomic unification (see Section 3.3.4) is adopted; see (Takeuchi and Furukawa 1986) for a discussion about this topic. For these reasons, nowadays deep guards are not much used anymore. Perhaps the only example of a language in which they are still used is ANDORRA Prolog (Haridi and

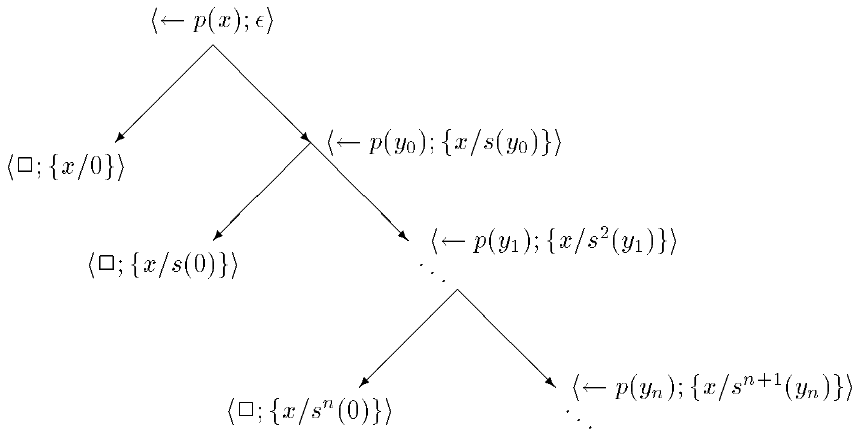

FIG. 3. Transitions in the program with flat guards. For simplicity, only the bindings on x are represented.

Brand 1988; Haridi 1990; Haridi and Janson 1990, 1991; Haridi *et al.* 1992), where they are used in combination with don't know nondeterminism.

3.2.4 *Synchronization*

Most of the concurrent logic languages realize synchronization by means of restrictions on the resolution step (head unification, guard evaluation). Usually these restrictions depend on the state of the variables, which is influenced by the actions done by the other processes in the network.

In the presence of this synchronization mechanism, a process can get stuck not only because of the absence of compatible clauses (failure), but also because the conditions which control its next steps are not satisfied. In this case the process *is suspended.* Later on, the bindings on common variables established by the other processes might cause the conditions to get satisfied, and in this case the process is *resumed.* Thus suspension is not definitive, while failure is. Another difference is that failure is *catastrophic*, namely when a process fails the whole network fails, whereas the network suspends (deadlocks) only if all the processes suspend. For these reasons usually one distinguishes in the observables these two kinds of termination mode. We use an additional symbol, dd, to represent suspension.

In order to describe this synchronization mechanism in a general way we assume two conditions, $cond_1$ and $cond_2$, associated with the head unification and with the guard evaluation respectively, which have to be

C1	$\langle \leftarrow A ; \vartheta \rangle \longrightarrow \langle \Box ; \vartheta\eta \rangle$	A is a built-in and $eval(A\vartheta) = true$ with substitution η and $cond_3(\eta, A\vartheta)$ holds
C2	$\langle \leftarrow A ; \vartheta \rangle \longrightarrow \langle \bar{B} ; \vartheta\sigma\eta \rangle$	there exists A' s.t. $A' \leftarrow \bar{g} \mid \bar{B} \in P_{ren}$, $\sigma = mgu(A\vartheta, A')$, $eval(\bar{g}\sigma) = true$ with substitution η and $cond_1(\sigma, A\vartheta)$, $cond_2(\sigma\eta, A\vartheta)$ hold
C3	$\dfrac{\langle \leftarrow \bar{A} ; \vartheta \rangle \longrightarrow \langle \leftarrow \bar{A}' ; \vartheta' \rangle}{\begin{array}{c}\langle \leftarrow \bar{A}, \bar{B} ; \vartheta \rangle \longrightarrow \langle \leftarrow \bar{A}', \bar{B} ; \vartheta' \rangle \\ \langle \leftarrow \bar{B}, \bar{A} ; \vartheta \rangle \longrightarrow \langle \leftarrow \bar{B}, \bar{A}' ; \vartheta' \rangle\end{array}}$	

Table 5 Synchronization and flat guards: computation rules

tested before committing to a certain clause. Furthermore we assume a third condition, $cond_3$, specific for built-in's. These conditions depend on the binding which would be generated and on the selected atom. Tables 5-7 specifies the transition system for a language with flat guards. Deep guards are more complicated; the reader interested in a formal treatment of them is referred to (Saraswat 1987a) and to (Levy 1988).

The observables are defined as follows:

Definition 3.12. (Observables) *Given a program P, for every goal G we define*

$$\mathcal{O}_{ss}(G) = \{\vartheta_{|G} \mid \langle G ; \epsilon \rangle \longrightarrow^* \langle \Box ; \vartheta \rangle\},$$

$$\mathcal{O}_{ff}(G) = \{ff \mid \textit{there exists } \vartheta \textit{ such that } \langle G ; \epsilon \rangle \longrightarrow^* \langle ff ; \vartheta \rangle\},$$

$$\mathcal{O}_{dd}(G) = \{\vartheta_{|G} \mid \langle G ; \epsilon \rangle \longrightarrow^* \langle dd ; \vartheta \rangle\},$$

$$\mathcal{O}_{ii}(G) = \{(lub_n \vartheta_n)_{|G} \mid \textit{there exists an infinite computation } \langle G ; \epsilon \rangle \longrightarrow \langle G_1 ; \vartheta_1 \rangle \longrightarrow \ldots \langle G_n ; \vartheta_n \rangle \longrightarrow \ldots\}.$$

F1	$\langle \leftarrow A;\vartheta\rangle \longrightarrow \langle f\!f;\vartheta\rangle$	A is a built-in and $eval(A\vartheta\sigma) = false$
F2	$\langle \leftarrow A;\vartheta\rangle \longrightarrow \langle f\!f;\vartheta\rangle$	for all $A' \leftarrow \bar{g} \mid \bar{B} \in P_{ren}$, $mgu(A\vartheta, A') = false$ or $eval(\bar{g}\ mgu(A\vartheta, A')) = false$
F3	$\dfrac{\langle \leftarrow \bar{A};\vartheta\rangle \longrightarrow \langle f\!f;\eta\rangle}{\begin{array}{c}\langle \leftarrow \bar{A},\bar{B};\vartheta\rangle \longrightarrow \langle f\!f;\eta\rangle\\ \langle \leftarrow \bar{B},\bar{A};\vartheta\rangle \longrightarrow \langle f\!f;\eta\rangle\end{array}}$	

Table 6 Synchronization and flat guards: failure rules

S1	$\langle \leftarrow A;\vartheta\rangle \longrightarrow \langle dd;\vartheta\rangle$	A is a built-in and $eval(A\vartheta) = true$ with substitution η and $cond_3(\eta, A\vartheta)$ does not hold
S2	$\langle \leftarrow A;\vartheta\rangle \longrightarrow \langle dd;\vartheta\rangle$	Rules **C2** and **F2** do not apply
S3	$\dfrac{\langle \leftarrow \bar{A};\vartheta\rangle \longrightarrow \langle dd;\vartheta\rangle \quad \langle \leftarrow \bar{B};\vartheta\rangle \longrightarrow \langle dd;\vartheta\rangle}{\langle \leftarrow \bar{A},\bar{B};\vartheta\rangle \longrightarrow \langle dd;\vartheta\rangle}$	

Table 7 Synchronization and flat guards: suspension rules

Note that the failure rules (**F1-F3**) and the suspension rules (**S1-S3**) are complementary and that in order to describe the observables it is not necessary to have both sets of rules. For instance, we could remove Rules **S1-S3** and the configurations of the form $\langle dd, \vartheta\rangle$, and describe $\mathcal{O}_{dd}$ in terms of the absence of outgoing transitions.

3.3 Variables as input-output channels

In most languages the conditions for synchronization specify that certain variables may not get instantiated during the resolution step. These specific conditions give rise to a classification of the variables into *input* and *output* variables, the former being the ones to which the above restriction applies. This leads to the concept of a system of *producers* and *consumers*; the *flow of data* from producers to consumers requires the consumer to wait until some specified data has been produced.

In this section we give an overview of the main languages which are based on this kind of synchronization mechanism.

3.3.1 *The Relational Language and PARLOG*

In PARLOG (Clark and Gregory 1986; Gregory 1987) input and output variables are specified via a *mode declaration*. For every predicate p there exists a declaration of the form

$$p(\ldots, ?, \ldots, \hat{}, \ldots),$$

where '?' and '^' indicate the input and output arguments of p, respectively. Thus the condition $cond_1(\vartheta, p(\bar{t}))$ holds iff ϑ does not bind the input arguments of p. A guard is not allowed to bind the input variables; such a situation generates an error.

The Relational Language (Clark and Gregory 1981), which is the predecessor of PARLOG and corresponds to the subset called Directional PARLOG (Gregory 1987), has more restrictions. In particular, the substitution generated by the head unification cannot bind the variables occurring in an output argument of the head of the clause. This means that the output variables of a process can be bound only by the process itself. Channels are strictly unidirectional; multiproducers are not allowed and unification is in practice replaced by (one-way) pattern matching.

As an example we give the specification of the *angelic merge* process in PARLOG. A merge process takes two input streams l_1, l_2 of data and merges them into an output stream l_3, see Figure 4.

Such a process is called angelic if it is guaranteed that when no data is available on one input then all data from the other input will be processed. In the following, [ ] represents the empty list, and signals the the *end of the stream*. This must not be confused with the *empty stream*, represented by an unbound list variable.

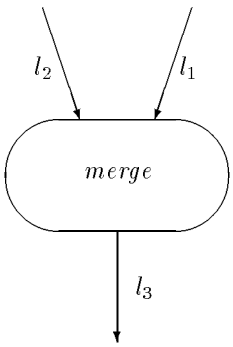

FIG. 4. The merge process.

The angelic merge process in PARLOG can be defined as follows

$$\begin{array}{lcl} merge([x|l_1], l_2, [x|l_3]) & \leftarrow & \mid merge(l_1, l_2, l_3), \\ merge(l_1, [x|l_2], [x|l_3]) & \leftarrow & \mid merge(l_1, l_2, l_3), \\ merge([\,], l, l) & \leftarrow & \mid, \\ merge(l, [\,], l) & \leftarrow & \mid, \end{array}$$

with the mode declaration $merge(?, ?, \hat{})$. The merge specified by the program above is angelic because if one input stream is empty, then the synchronization condition prevents the selection of the clause which would process that input stream. For instance, if the first argument is not bound then only the second or the fourth clause can be selected.

3.3.2 *Concurrent Prolog*

In Concurrent Prolog (Shapiro 1983, 1986, 1988), a variable occurrence in the goal, in the guard or in the body may be decorated with the annotation '?', which marks it as read-only. The conditions $cond_1(\vartheta, A)$ and $cond_2(\vartheta, A)$ are verified iff ϑ does not bind the read-only variables of A to non-variable terms. Variables which are bound to read-only variables inherit the read-only annotation. For example, given a clause $p(x) \leftarrow q(x)$, the goal $\leftarrow p(y?), r(y)$ can evolve in $\leftarrow q(y?), r(y)$.

The *angelic merge* process is specified in Concurrent Prolog as follows.

$$
\begin{array}{lcl}
merge([x|l_1], l_2, [x|l_3]) & \leftarrow & |\; merge(l_1?, l_2, l_3), \\
merge(l_1, [x|l_2], [x|l_3]) & \leftarrow & |\; merge(l_1, l_2?, l_3), \\
merge([\,], l, l) & \leftarrow & |, \\
merge(l, [\,], l) & \leftarrow & |\,.
\end{array}
$$

A call to the merge process should be of the form $\leftarrow merge(l_1?, l_2?, l_3)$.

3.3.3 *Guarded Horn Clauses*

In the Guarded Horn Clauses language (Ueda 1987, 1988), GHC for short, there is a predefined binary predicate $=$ which performs the unification of the two arguments. The atoms of the form $t = u$ are the only ones which can produce bindings; for all the other atoms A the conditions $cond_1(\vartheta, A)$ and $cond_2(\vartheta, A)$ are validated iff ϑ does not bind any variable of A. In other words, all the variables of A are treated as input variables. Note that therefore the predicate $=$ cannot be defined in GHC.

The angelic merge process can be specified in GHC as follows:

$$
\begin{array}{lcl}
merge([x|l_1], l_2, l_3) & \leftarrow & |\; l_3 = [x|l_3'], merge(l_1, l_2, l_3'), \\
merge(l_1, [x|l_2], l_3) & \leftarrow & |\; l_3 = [x|l_3'], merge(l_1, l_2, l_3'), \\
merge([\,], l_2, l_3) & \leftarrow & |\; l_2 = l_3, \\
merge(l_1, [\,], l_3) & \leftarrow & |\; l_1 = l_3.
\end{array}
$$

3.3.4 *Atomic versus eventual unification*

The main distinguishing feature of Concurrent Prolog with respect to PARLOG and GHC is the presence of the so-called *atomic unification*. Roughly, this means that the head unification and the guard evaluation can produce bindings (on the variables which are not read-only) and that this is done atomically. In GHC and PARLOG bindings can be produced only after the commitment to a certain clause (*eventual unification*). In GHC the conditions $cond_1$ and $cond_2$ are such that no bindings are produced during the selection of the clause in Rule **C2** of Table 5. In PARLOG, at least in the latter versions, there are some additional restrictions that prevent from producing bindings during the head unification and the guard evaluation.

Our transition system models a *global memory*, represented by the accumulated bindings in the configuration. Furthermore, the head unification and the guard evaluation are seen as one atomic action, i.e. they are performed in one single step. In a different model, either based on *distributed memory*, or where the head unification and the guard evaluation might require more than one step, additional machinery would be necessary to realize the synchronization mechanism of Concurrent Prolog.

In a system where the head unification and the guard evaluation are not atomic actions, Concurrent Prolog can be modeled by encapsulating them into a critical section (Knijnenburg and Kok 1989).

On the other hand, a distributed model can be obtained by associating to each process its local memory, where the bindings established by a process are stored, instead of being immediately communicated to the other processes. Of course, the 'atomic unification' principle requires a *consistency check* before the commitment, to verify that the bindings produced by the head unification and by the guard evaluation are consistent with the bindings established by the other processes. When a process starts performing the consistency check, the other processes must wait before producing new bindings until it has finished.

Of course, in a distributed model atomic unification is very expensive to implement: the consistency check is a bottle-neck which actually reduces the potential parallelism of the system. However, it is very powerful. In fact it can be argued that languages with atomic unification are strictly more expressive than languages without it (de Boer and Palamidessi 1994).

Example 3.13. *Suppose we want to define a process $p(x)$ which produces either $\{x/a\}$ or $\{x/b\}$, and that maintains these possibilities (without failing) when put in parallel with itself. In Concurrent Prolog such $p(x)$ can be specified by the program:*

$$p(a) \leftarrow |,$$
$$p(b) \leftarrow | \,.$$

On the other hand this process cannot be specified in GHC or in PARLOG. The 'best approximation' in GHC would be the program

$$p(x) \leftarrow | \; x = a,$$
$$p(x) \leftarrow | \; x = b,$$

but such a program does not satisfy all the requirements: the goal $\leftarrow p(x), p(x)$ might fail.

A more interesting example is shown by Shapiro (1989). Such example, taken from a protocol proposed by Safra (1986), can be illustrated as follows. There are two 'boy' processes and two 'girl' processes. Boys and girls are connected by one-to-one channels. Each boy wishes to 'start a relation' with one girl, and he doesn't care who. Each girl wishes to start a relation with one boy, and she doesn't care who. We want to program the nodes so to establish the couples, without risk of deadlock. The solution proposed in

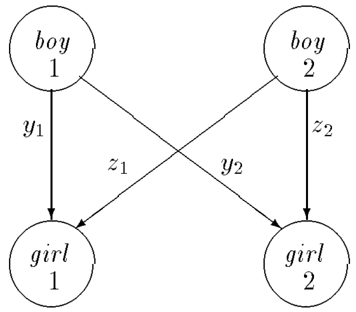

FIG. 5. Partners selection.

(Safra 1986; Shapiro 1989) is relative to the case in which boys take the initiative. Each boy sends to the girls the incomplete message $hello(x)$ (where x is a local variable). Each girl selects nondeterministically one message and instantiates x to her unique identifier. The program is as follows:

$$boy(x, y_1, y_2) \quad \leftarrow \quad \mid y_1 = hello(x), y_2 = hello(x),$$

$$\begin{aligned} girl(Id, hello(x_1), z) \quad &\leftarrow \quad x_1 = Id \mid, \\ girl(Id, y, hello(x_2)) \quad &\leftarrow \quad x_2 = Id \mid . \end{aligned}$$

The initial network, also illustrated in Figure 5, is

$$\leftarrow boy(x_1, y_1, y_2), boy(x_2, z_1, z_2), girl(1, y_1?, z_1?), girl(2, y_2?, z_2?).$$

This example shows that atomic unification allows mimicking the communication protocol of synchronous concurrent languages like CSP, where it is possible to define processes whose choices depend upon the outcomes of simultaneous interactions with the environment. Thus we can say in a sense that atomic unification makes asynchronous communication as powerful as synchronous communication.

The case in which both boys and girls take the initiative is more complicate, but still solvable with atomic unification. Particularly interesting is the case of a symmetric network, in which all nodes are copies of the same process, i.e. they are defined by the same program. The interested reader is invited to write such program in Concurrent Prolog. This case seems very difficult to be described in a language with eventual unification. Our conjecture is that it is not possible at all.

Symmetricity seems to be the key point in which atomic unification

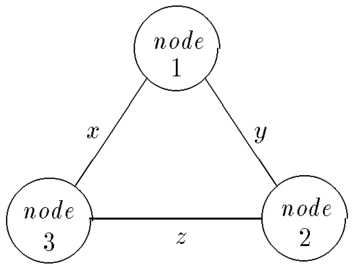

FIG. 6. Election of a leader.

shows its supremacy concerning expressivity. Let's consider the problem of finding *symmetric algorithms for the election of a leader in a symmetric network*. Such problem was analyzed by Bougé (1988) to show that CSP without output guards is strictly less expressive than full CSP (Hoare 1978). In fact, it is possible to write a program deadlock-free in full CSP which guarantees that a leader is eventually elected, whereas this is not possible in the sublanguage without output guards. The proof of Bougé can be adapted to show that an algorithm satisfying these requirements cannot be written in a concurrent logic language with eventual unification. On the other hand, it can be written by using atomic unification:

$$\begin{array}{lcl} node(x,y,Id) & \leftarrow & x = elected(Id), y = elected(Id) \mid, \\ node(x,y,Id) & \leftarrow & x? = elected(w) \mid, \\ node(x,y,Id) & \leftarrow & y? = elected(w) \mid . \end{array}$$

The initial network, which is illustrated in Figure 6, is

$$\leftarrow node(x,y,1), node(y,z,2), node(z,x,3).$$

3.4 An alternative approach to synchronization: P-Prolog

The synchronization mechanism of P-Prolog (Yang 1986; Yang and Aiso 1986) is completely different from those of the languages considered so far. The basic idea is that a process can proceed only when there is only one clause which can be applied for the resolution step. Formally this mechanism is described as follows. Given an atom A and a clause $C =$

C2	$\langle \leftarrow A; \vartheta \rangle \longrightarrow \langle \bar{B}; \vartheta\sigma \rangle$	there exists $C = A' \leftarrow \bar{g} \mid \bar{B} \in P_{ren}$ s.t. $try(A\vartheta, C) = \sigma$ and for all $C' \in P_{ren}$ if $C' \neq C$ then $try(A\vartheta, C') = f\!f$

Table 8 The computation rule of P-Prolog

$A' \leftarrow \bar{g} \mid \bar{B}$, let $try(A, C)$ be defined by

$$try(A,C) = \begin{cases} \sigma\eta & \text{if } \mathrm{mgu}(A,A') = \sigma \text{ and } \langle \leftarrow \bar{g}\sigma; \epsilon \rangle \longrightarrow^* \langle \Box; \eta \rangle \\ f\!f & \text{if } \mathrm{mgu}(A,A') = \mathit{false} \text{ or} \\ & \mathrm{mgu}(A,A') = \sigma \text{ and } \exists\eta.\ \langle \leftarrow \bar{g}\sigma; \epsilon \rangle \longrightarrow^* \langle f\!f; \eta \rangle \\ dd & \text{otherwise} \end{cases}$$

The transition system of P-Prolog is obtained by modifying the Rule **C2** of Table 5 as described in Table 8.

Note that P-Prolog is a deterministic language, in the following sense:

Proposition 3.14. *Let P be a P-Prolog program and $\leftarrow \bar{A}$ be a P-Prolog goal. If $\langle \leftarrow \bar{A}; \epsilon \rangle$ has a successful computation in P with c.a.s. ϑ, then every computation of $\langle \leftarrow \bar{A}; \epsilon \rangle$ in P terminates with success and a c.a.s. equivalent (up to renaming) to ϑ.*

As a consequence, typical non-deterministic processes, such as the merge process, cannot be expressed in P-Prolog.

The main use of this synchronization mechanism is to guarantee the correct 'flow of data' among the parallel components of a network, thus avoiding redundant or wrong computations.

Example 3.15. *Consider the following program, which computes the Fibonacci function*

$$fib(0, s(0)) \leftarrow \mid,$$

$$fib(s(0), s(0)) \leftarrow \mid,$$

$$fib(s(s(x)), y) \leftarrow \mid fib(s(x), z), fib(s(x), w), plus(z, w, y),$$

$$plus(0, y, y) \leftarrow \mid,$$

$$plus(s(x), y, s(z)) \leftarrow \mid plus(x, y, z).$$

Consider the initial goal $\leftarrow \mathit{fib}(s(s(0)), y)$. *After one resolution step, the goal becomes* $\leftarrow \mathit{fib}(s(0), z)$, $\mathit{fib}(0, w)$, $\mathit{plus}(z, w, y)$. *Now, the plus process cannot proceed until the first process fib has computed a binding for* z.

The principle of P-Prolog has been embodied later on in the ALPS languages and in Andorra-Prolog.

4 The ALPS languages

Recently Jaffar and Lassez (1987) proposed the paradigm of Constraint Logic Programming (CLP), an extension to logic programming which gained immediate popularity in the community. The extension consists of considering an arbitrary domain of data instead of the standard Herbrand universe, and by replacing the concept of unification with the concept of constraint over such domain. The unifiability test becomes a test of consistency of the new constraint with respect to the constraints previously established.

More formally, a CLP clause and goal have the form

$$A \leftarrow \bar{c}, \bar{B},$$
$$\leftarrow \bar{c}, \bar{A},$$

where $\bar{c}$ is a sequence of constraints. The computation step is defined as follows: let A_i be an atom in the goal

$$\leftarrow \bar{c}', \bar{A},$$

and let $H \leftarrow \bar{c}, \bar{B}$ be a renamed clause of the program, such that $\bar{c} \wedge A_i = H \wedge \bar{c}'$ is satisfiable (consistent), where '$\wedge$' is the logical conjunction. Then the new goal becomes

$$\leftarrow \bar{c}, \bar{c}', A_i = H, \bar{A} \setminus A_i, \bar{B}.$$

Maher (1987) has developed a concurrent version of CLP, which he called 'The ALPS family' (each member being associated to a constraint system). Among the other features, his proposal containes an innovative principle for synchronization: the *validation rule*. From a technical point of view this rule is the natural extension of the GHC synchronization mechanism to the constraint framework, but the very elegant, logical formulation has induced the logic programming community to regard this idea as the 'right' approach to synchronization. Also the *ask rule* of ccp is essentially based on the same concept.

In order to describe formally the ALPS family we briefly survey the notion of constraint system. We follow here the approach of Saraswat and Rinard (1990), which is based on Scott's theory of *information systems* (Scott 1982).

4.1 Constraint systems

Saraswat and Rinard (1990) define a constraint system starting from a primitive notion of simple constraints (tokens) on which a compact entailment relation $\vdash$ is defined. Then a constraint system is constructed by considering sets of tokens and by extending the entailment relation on it. This construction is made in such a way that the resulting structure is a complete *algebraic* lattice, which ensures the effectiveness of the extended entailment relation. In this paper we abstract from this construction, and we just consider the resulting structure.

Definition 4.1. (Constraint system) *A constraint system is a complete algebraic lattice $\langle \mathcal{C}, \leq, \wedge, true, false \rangle$ where $\wedge$ is the lub operation, and true, false are the least and the greatest elements of $\mathcal{C}$, respectively.*

In this framework, $\leq$ is the reverse of the entailment relation $\vdash$. Formally: $\forall c, d \in C.\ c \vdash d \Leftrightarrow d \leq c$. Inconsistency is modeled by false: two elements c, c' are consistent iff $c \wedge c' \neq false$.

In order to model local variables we need the existential quantifier. An elegant way to formalize the existential quantification in the constraint framework is through the theory of cylindric algebras of Henkin *et al.* (1971). This leads to the concept of *cylindric constraint system*.

In the following, *Var* is a (denumerable) set of variables with typical elements $x, y, z, \ldots$.

Definition 4.2. (Cylindric constraint system) *Let $\langle \mathcal{C}, \leq, \wedge, true, false \rangle$ be a constraint system. Assume that for each $x \in Var$ a function $\exists_x : \mathcal{C} \to \mathcal{C}$ is defined such that for any $c, d \in \mathcal{C}$:*

(i) $\exists_x(c) \leq c$,

(ii) *if* $d \leq c$ *then* $\exists_x(d) \leq \exists_x(c)$,

(iii) $\exists_x(c \wedge \exists_x(d)) = \exists_x(c) \wedge \exists_x(d)$,

(iv) $\exists_x(\exists_y(c)) = \exists_y(\exists_x(c))$.

Then $\mathbf{C} = \langle \mathcal{C}, \leq, \wedge, true, false, Var, \exists \rangle$ is a cylindric constraint system.

In the following $\exists_x(c)$ is denoted by $\exists_x c$ with the convention that, in case of ambiguity, the scope of $\exists_x$ is limited to the first constraint subexpression. (So, for instance, $\exists_x c \wedge d$ stands for $\exists_x(c) \wedge d$.) Furthermore, for $\bar{x} = x_1, \ldots, x_n$ the notation $\exists_{\bar{x}}(A)$ stands for $\exists_{x_1}(\ldots \exists_{x_n}(A) \ldots)$.

In order to model parameter passing, it will be useful to consider the so-called *diagonal formulas* (Henkin *et al.* 1971). We assume that, for x, y ranging in *Var*, $\mathcal{A}$ contains the *diagonal elements* d_{xy} which satisfy the following properties.

(i) $d_{xx} = true$,

(ii) if $z \not\equiv x$ and $z \not\equiv y$ then $d_{xy} = \exists_z(d_{xz} \wedge d_{zy})$,

C2′	$\langle \leftarrow A; c\rangle \longrightarrow \langle \bar{B}; c \wedge (A = A') \wedge c'\rangle$	there exists only one clause $A' \leftarrow c' \mid \bar{B} \in P_{ren}$ s.t. $c \wedge (A = A') \wedge c' \neq \mathit{false}$
C2″	$\langle \leftarrow A; c\rangle \longrightarrow \langle \bar{B}; c \wedge (A = A') \wedge c'\rangle$	there exists $A' \leftarrow c' \mid \bar{B} \in P_{ren}$ s.t. $\exists_{\bar{x}_l}(c' \wedge (A = A')) \leq c$

Table 9 The computation rules of ALPS. In **C2″**, $\bar{x}_l$ represent the local variables of the selected clause.

(iii) if $x \not\equiv y$ then $c \leq d_{xy} \wedge \exists_x(c \wedge d_{xy})$.

Note that if **C** models the equality theory, then the elements d_{xy} can be thought of as the formulas $x = y$. In the following, given $\bar{x} = x_1, \ldots, x_n$ and $\bar{y} = y_1, \ldots, y_n$, we use the notation $d_{\bar{x}\bar{y}}$ to represent $d_{x_1y_1} \wedge \ldots \wedge d_{x_ny_n}$.

4.2 The validation rule

In ALPS clauses and goals have the form

$$A \leftarrow \bar{c} \mid, \bar{B},$$
$$\leftarrow \bar{c}, \bar{A},$$

where $\bar{B}$ can contain constraints. Note that the guard part of a clause contains only constraints.

The transition system for ALPS is obtained by replacing Rule **C2** of Table 5 by Rules **C2′** and **C2″** described in Table 9. Of course, the second component of a configurations is now a constraint.

Note that **C2′** corresponds to the P-Prolog rule. **C2″** is called *validation rule*, and it extends the GHC mechanism.

The ALPS paradigm is more flexible than P-Prolog and GHC. Like GHC, it allows the definition of the merge process. Furthermore, thanks to the P-Prolog component, it allows the same program to be used with different functionalities.

Example 4.3. *Consider the following append program in ALPS.*

$$\mathit{append}([\,], l, l) \leftarrow \mid,$$
$$\mathit{append}([x \mid l_1], l_2, [x \mid l_3]) \leftarrow \mid \mathit{append}(l_1, l_2, l_3).$$

This program can be activated by any call of append which has two arguments instantiated, and it will compute the third.

In GHC it is not possible to write an *append* program having the feature above described: the kind of argument (input or output) is fixed by the program.

5 Concurrent constraint programming

Concurrent constraint programming was proposed by Saraswat (1989), Saraswat and Rinard (1990), Saraswat *et al.* (1991). We follow here the definition given in (Saraswat and Rinard 1990), which is more general than the later formulations, because it covers the case of atomic unification.

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.

5.1 The language

Given a set of variables *Var* with typical elements $x, y, \ldots$, we assume fixed a cylindric constraint system $\mathbf{C} = \langle \mathcal{C}, \leq, \wedge, \mathit{true}, \mathit{false}, \mathit{Var}, \exists \rangle$. The description of the language and the semantical definitions are parametric with respect to the constraint system.

Processes are described by the following grammar

$$\begin{array}{rl} \mathit{Processes} & P ::= D.A \\ \mathit{Declarations} & D ::= \epsilon \mid p(\bar{x}) \text{ :- } A \mid D, D \\ \mathit{Guards} & g ::= \mathbf{tell}(c) \mid \mathbf{ask}(c) \\ \mathit{Agents} & A ::= \mathbf{Stop} \mid \sum_{i=1}^{n} g_i \rightarrow A_i \mid A \;\|\; A \mid \exists_x A \mid p(\bar{x}) \end{array}$$

The agent **Stop** represents successful termination. The basic actions are given by the $\mathbf{ask}(c)$ and $\mathbf{tell}(c)$ constructs, where c is a *finite constraint*, i.e. an algebraic element of $\mathcal{C}$. These actions work on a common *store* which ranges over $\mathcal{C}$. The action $\mathbf{ask}(c)$ is a test on the current store and its execution does not modify the store. We say that $\mathbf{ask}(c)$ is *enabled* in d iff $c \leq d$. An action $\mathbf{ask}(c)$ *fails* in d if $c \wedge d = \mathit{false}$. The action $\mathbf{tell}(c)$ is enabled in the current store d if $c \wedge d \neq \mathit{false}$ and its execution sets the store to $c \wedge d$, otherwise it fails. The *guarded choice* agent $\sum_{i=1}^{n} g_i \rightarrow A_i$ selects nondeterministically one g_i which is enabled, and then behaves like A_i. If no guards are enabled, then it *fails* in case all guards fail, otherwise it *suspends*, waiting for other (parallel) agents to add information to the store. Parallel composition is represented by $\|$. The situation in which all components of a system of parallel agents suspend is called *global suspension* or *deadlock*. The agent $\exists_x A$ behaves like A, with x considered *local* to A. Finally, the agent $p(\bar{x})$ is a procedure call, where p is the name of the procedure and

$\bar{x}$ is the list of the actual parameters. The meaning of $p(\bar{x})$ is given by a procedure declaration of the form $p(\bar{y})$:- A, where $\bar{y}$ is the list of the formal parameters. We assume that in a process $D.A$ there is one and only one procedure declaration for every procedure name occurring in A; this is not a restriction with respect to concurrent logic languages, as the presence of alternative clauses can be simulated by the choice construct. In the following, we omit the declaration part when it is empty. Furthermore, we use $\mathbf{tell}(c)$ as syntactic sugar for $\mathbf{tell}(c) \rightarrow \mathbf{Stop}$.

5.2 Operational semantics

The operational model of ccp, informally introduced above, is described by a transition system $T = (\mathit{Conf}, \longrightarrow)$. The configurations (in) *Conf* are pairs consisting of a process or a guard or a *termination mode*, and a constraint representing the store. Tables 10-12 describe the rules of T with respect to a given set of declarations D. Rules **R2**, ..., **R6** describe the execution of the basic actions. The guarded choice operator models global non-determinism (Rules **R7**, ..., **R9**), in the sense that it depends on the current store whether or not a guard is enabled, and the current store is subject to modifications by the external environment as described by Rule **R10**. With respect to Rules **R8** and **R9** it is to be understood that in case $n = 1$ the premise of both rules reduces to $\langle g_1, c\rangle \longrightarrow \langle \tau, c\rangle$, where $\tau = \mathit{ff}, \mathit{dd}$, respectively. Rule **R10** describes parallelism as interleaving.

In order to describe locality (Rules **R13** and **R14**) the syntax has been extended by an agent $\exists_x^d A$ in which x is local to A and d is the store that has been produced locally on x. Initially the local store is empty, i.e. $\exists_x A = \exists_x^{true} A$. Intuitively, a process $\exists_x A$ can make a step in the store c if the process A can make a step in the store $\exists_x c$, since at local level A does not see the global information on x. If the local computation had already produced some constraint d on x, that will be considered as well, i.e. the store seen locally is $d \wedge \exists_x c$. The constraint d' produced by this step is added to the local store, and $\exists_x d'$ (i.e. d' with the information on x removed) is added to the global store.

The execution of a procedure call is modeled by **R15**. $\Delta_{\bar{y}}^{\bar{x}}$ stands for $\exists_{\bar{\alpha}}^{d_{\bar{x}\bar{\alpha}}} \exists_{\bar{y}}^{d_{\bar{\alpha}\bar{y}}}$ and it is used to establish the link between the formal parameters $\bar{y}$ and the actual parameters $\bar{x}$. The variables $\bar{\alpha}$ are introduced in order to avoid problems related to names clash between $\bar{x}$ and $\bar{y}$. They are assumed to be all distinct and to not occur in the program.

In the following we assume the set of declarations to be fixed.

The notion of observables we consider here is the collection of the final results of the computations, together with the termination modes. For the sake of simplicity we only consider finite computations.

R1	$\langle \mathbf{Stop}, c\rangle \longrightarrow \langle ss, c\rangle$		
R2	$\langle \mathbf{tell}(c), d\rangle \longrightarrow \langle ss, c \wedge d\rangle$	$c \wedge d \neq \mathit{false}$	
R3	$\langle \mathbf{tell}(c), d\rangle \longrightarrow \langle \mathit{ff}, c\rangle$	$c \wedge d = \mathit{false}$	
R4	$\langle \mathbf{ask}(c), d\rangle \longrightarrow \langle ss, d\rangle$	$c \leq d$	
R5	$\langle \mathbf{ask}(c), d\rangle \longrightarrow \langle dd, d\rangle$	$c \not\leq d$ *and* $c \wedge d \neq \mathit{false}$	
R6	$\langle \mathbf{ask}(c), d\rangle \longrightarrow \langle \mathit{ff}, d\rangle$	$c \wedge d = \mathit{false}$	

Table 10 The transition system T: basic actions.

Definition 5.1. *The observables are given by the function*

$$\mathcal{O}(A) = \{\langle c, \tau\rangle \mid \langle A, \mathit{true}\rangle \longrightarrow^{+} \langle \tau, c\rangle\}$$

where $\longrightarrow^{+}$ *denotes the transitive closure of* $\longrightarrow$. *We don't need to consider here the reflexive closure because every agent makes at least one step before reaching a final configuration.*

Note that from this definition we can derive the input/output behavior of a process. In fact, the results of A in an initial store $c \neq \mathit{true}, \mathit{false}$ coincide with the results of the process $\mathbf{tell}(c) \parallel A$ in the initial store true.

5.3 Embedding concurrent logic programming

Concurrent constraint programming subsumes the main flat concurrent logic languages. We show here how Flat GHC and FCP (a version of Flat CP, proposed by Kliger *et al.* (1988)) can be translated into ccp.

We will use the following notation: if $\bar{t} = t_1, \ldots, t_n$ and $\bar{u} = u_1, \ldots, u_n$ then $\bar{t} = \bar{u}$ stands for $t_1 = u_1 \wedge \ldots \wedge t_n = u_n$.

5.3.1 *Embedding Flat GHC*

A Flat GHC goal $\leftarrow A_1, \ldots, A_n$ corresponds to the ccp agent $B_1 \parallel \ldots \parallel B_n$; where, if A_i is the atom $p(\bar{t})$, then B_i is the agent $\exists_{\bar{x}}(p(\bar{x}) \parallel \mathbf{tell}(\bar{x} = \bar{t}))$

$$\textbf{R7} \quad \frac{\langle g_i, c\rangle \longrightarrow \langle ss, d\rangle}{\langle \sum_{i=1}^{n} g_i \rightarrow A_i, c\rangle \longrightarrow \langle A_i, d\rangle}$$

$$\textbf{R8} \quad \frac{\langle g_n, c\rangle \longrightarrow \langle \mathit{ff}, c\rangle \quad \langle \sum_{i=1}^{n-1} g_i \rightarrow A_i, c\rangle \longrightarrow \langle \tau, c\rangle}{\langle \sum_{i=1}^{n} g_i \rightarrow A_i, c\rangle \longrightarrow \langle \tau, c\rangle} \quad \tau \in \{\mathit{ff}, dd\}$$

$$\textbf{R9} \quad \frac{\langle g_n, c\rangle \longrightarrow \langle dd, c\rangle \quad \langle \sum_{i=1}^{n-1} g_i \rightarrow A_i, c\rangle \longrightarrow \langle \tau, c\rangle}{\langle \sum_{i=1}^{n} g_i \rightarrow A_i, c\rangle \longrightarrow \langle dd, c\rangle} \quad \tau \in \{\mathit{ff}, dd\}$$

Table 11 The transition system T: guarded choice.

with $\bar{x}$ fresh distinct variables.

Concerning predicate definitions, let p be defined by the following Flat GHC clauses:

$$\begin{array}{lll} p(\bar{t}_1) & \leftarrow & \bar{u}_1 = \bar{u}'_1 \mid \bar{A}_1, (\bar{v}_1 = \bar{v}'_1), \bar{A}'_1, \\ \vdots & & \\ p(\bar{t}_n) & \leftarrow & \bar{u}_n = \bar{u}'_n \mid \bar{A}_n, (\bar{v}_n = \bar{v}'_n), \bar{A}'_n. \end{array}$$

Then p can be equivalently defined by the following ccp program:

$$\begin{array}{lll} p(\bar{x}) & \text{:-} & \mathbf{ask}(\exists_{\bar{y}}(\bar{x} = \bar{t_1} \wedge \bar{u}_1 = \bar{u}'_1)) \rightarrow E_1 \\ & & + \\ & & \vdots \\ & & + \\ & & \mathbf{ask}(\exists_{\bar{y}}(\bar{x} = \bar{t_n} \wedge \bar{u}_n = \bar{u}'_n)) \rightarrow E_n, \end{array}$$

where $\bar{x}$ are fresh distinct variables and $\bar{y}$ are all the other variables occur-

R10 $$\frac{\langle A, c\rangle \longrightarrow \langle A', d\rangle}{\begin{array}{c}\langle A \parallel B, c\rangle \longrightarrow \langle A' \parallel B, d\rangle \\ \langle B \parallel A, c\rangle \longrightarrow \langle B \parallel A', d\rangle\end{array}}$$

R11 $$\frac{\langle A, c\rangle \longrightarrow \langle \mathit{ff}, c\rangle}{\begin{array}{c}\langle A \parallel B, c\rangle \longrightarrow \langle \mathit{ff}, c\rangle \\ \langle B \parallel A, c\rangle \longrightarrow \langle \mathit{ff}, c\rangle\end{array}}$$

R12 $$\frac{\langle A, c\rangle \longrightarrow \langle dd, c\rangle \quad \langle B, c\rangle \longrightarrow \langle dd, c\rangle}{\langle A \parallel B, c\rangle \longrightarrow \langle dd, c\rangle}$$

R13 $$\frac{\langle A, d \wedge \exists_x c\rangle \longrightarrow \langle B, d'\rangle}{\langle \exists_x^d A, c\rangle \longrightarrow \langle \exists_x^{d'} B, c \wedge \exists_x d'\rangle}$$

R14 $$\frac{\langle A, d \wedge \exists_x c\rangle \longrightarrow \langle \tau, d'\rangle}{\langle \exists_x^d A, c\rangle \longrightarrow \langle \tau, c \wedge \exists_x d'\rangle}$$

R15 $\langle p(\bar{x}), c\rangle \longrightarrow \langle \Delta_{\bar{y}}^{\bar{x}} A, c\rangle$ $\qquad p(\bar{y})$:- A is in D

Table 12 The transition system T: parallelism, hiding and procedure call.

ring in the original definition. The E_i's are the agents

$$E_i \quad = \quad \exists_{\bar{y}}(\mathbf{tell}(\bar{x} = \bar{t}_i \wedge \bar{u}_i = \bar{u}_i') \parallel \bar{B}_i \parallel \mathbf{tell}(\bar{v}_i = \bar{v}_i') \parallel \bar{B}_i'\ ),$$

where the $\bar{B}_i$'s and the $\bar{B}_i'$'s are obtained from the goals $\leftarrow \bar{A}_i$ and $\leftarrow \bar{A}_i'$ respectively as specified above.

Note that the ask-rule in ccp is more flexible than in ALPS: the quantification of the local variables is not done automatically by the evaluation of the ask, therefore we must put explicitly the existential quantifier into the guards, and tell the constraints afterwards in order to establish the links with the local variables.

Example 5.2. *Angelic merge can be defined in ccp as follows.*

$$
\begin{array}{lll}
merge(l_1, l_2, l_3) & :- & \mathbf{ask}(\exists_{x,l'_1}\ l_1 = [x|l'_1]) \rightarrow\ \exists_{x,l'_1,l'_3}(\mathbf{tell}(l_1 = [x|l'_1]) \parallel \\
& & \qquad\qquad\qquad\qquad\quad \mathbf{tell}(l_3 = [x|l'_3]) \parallel \\
& & \qquad\qquad\qquad\qquad\quad merge(l'_1, l_2, l'_3)\ ) \\
& & + \\
& & \mathbf{ask}(\exists_{y,l'_2}\ l_2 = [y|l'_2]) \rightarrow\ \exists_{y,l'_2,l'_3}(\mathbf{tell}(l_2 = [y|l'_2]) \parallel \\
& & \qquad\qquad\qquad\qquad\quad \mathbf{tell}(l_3 = [y|l'_3]) \parallel \\
& & \qquad\qquad\qquad\qquad\quad merge(l_1, l'_2, l'_3)\ ) \\
& & + \\
& & \mathbf{ask}(l_1 = [\,]) \rightarrow\ \mathbf{tell}(l_3 = l_2) \\
& & + \\
& & \mathbf{ask}(l_2 = [\,]) \rightarrow\ \mathbf{tell}(l_3 = l_1)
\end{array}
$$

5.3.2 *Embedding FCP*

FCP is a variant of Flat Concurrent Prolog in which input channels are specified statically. A typical FCP clause is of the form

$$p(\bar{x})\ \leftarrow\ \bar{g}_1 : \bar{g}_2 \mid \bar{A},$$

where $\bar{g}_1$ is the input guard, i.e. a sequence of equalities which must be unifiable without introducing new global bindings, and $\bar{g}_2$ is the output guard, namely a sequence of equalities whose unifier must be compatible with the current bindings. The evaluation of $\bar{g}_2$ corresponds to the atomic unification, i.e. it contains the consistency check and introduces new bindings before commitment.

The translation of a FCP goal into ccp is defined as in the Flat GHC case. Concerning the programs, we consider here, for the sake of simplicity, only the case in which each clause contains either $\bar{g}_1$ or $\bar{g}_2$, but not both. The interested reader is invited to extend the translation to the full case.

Let p be defined by the following FCP clauses:

$$
\begin{array}{lll}
p(\bar{x}) & \leftarrow & \bar{u}_1 = \bar{u}'_1 :\mid \bar{A}_1, \bar{v}_1 = \bar{v}'_1, \bar{A}'_1, \\
\vdots & & \\
p(\bar{x}) & \leftarrow & : \bar{u}_n = \bar{u}'_n \mid \bar{A}_n, \bar{v}_n = \bar{v}'_n, \bar{A}'_n.
\end{array}
$$

Then p can be equivalently defined by the following ccp program:

$$\begin{array}{rcl} p(\bar{x}) & \text{:-} & \mathbf{ask}(\exists_{\bar{y}}\,\bar{u}_1 = \bar{u}'_1) \rightarrow E_1 \\ & & + \\ & & \vdots \\ & & + \\ & & \mathbf{tell}(\exists_{\bar{y}}\,\bar{u}_n = \bar{u}'_n) \rightarrow E_n \end{array},$$

where the E_i's are the agents

$$E_i \quad = \quad \exists_{\bar{y}}(\mathbf{tell}(\bar{u}_i = \bar{u}'_i) \parallel \bar{B}_i \parallel \mathbf{tell}(\bar{v}_i = \bar{v}'_i) \parallel \bar{B}'_i\ ),$$

where the y_i's, the $\bar{B}_i$'s and the $\bar{B}'_i$'s are defined as in the Flat GHC case.

5.4 Denotational semantics

In this section we address the problem of a compositional semantics for concurrent constraint programming, correct with respect to the observables $\mathcal{O}$ defined in the previous section. Compositionality is considered one of the most desirable characteristics of a formal semantics, since it provides a foundation of program verification and modular design.

First we observe that $\mathcal{O}$ is not compositional:

Example 5.3. *Let* $A_1 = \mathbf{ask}(c) \rightarrow \mathbf{Stop}$ *and* $A_2 = \mathbf{ask}(d) \rightarrow \mathbf{Stop}$. *We have that* $\mathcal{O}(A_1) = \mathcal{O}(A_2) = \{\langle true, dd\rangle\}$. *However, given the agent* $B = \mathbf{tell}(c) \rightarrow \mathbf{Stop}$, *and assuming* $c \not\leq d$, *we have that* $\mathcal{O}(A_1 \parallel B) \neq \mathcal{O}(A_2 \parallel B)$, *since* $\langle true, c\rangle \in \mathcal{O}(A_1 \parallel B) \setminus \mathcal{O}(A_2 \parallel B)$.

Therefore we have to introduce some additional information which, for example, allows us to distinguish between the above two agents A_1 and A_2. A first natural attempt seems to add information about the *input/output* behavior: define $\mathcal{IO}$ as follows:

$$\mathcal{IO}(A) = \{\langle c, d, \tau\rangle \mid \ \langle A, c\rangle \longrightarrow^{+} \langle \tau, d\rangle\}$$

It is easy to check that indeed the semantics $\mathcal{IO}$ distinguishes between the agents A_1 and A_2. However, still the semantics $\mathcal{IO}$ does not provide the necessary information about the *branching structure* of agents. Consider the following counterexample (by Falaschi *et al.* (1993)) to the compositionality of $\mathcal{IO}$:

Example 5.4. *Let* $a, b, c \in \mathcal{C}$ *be constraints ordered by the relation* $a \leq$

$b \leq c$ and consider the agents A_1 and A_2

$$\begin{array}{rcl} A_1 & = & \mathbf{ask}(true) \rightarrow \mathbf{tell}(a) \\ & & + \\ & & \mathbf{ask}(b) \rightarrow \mathbf{tell}(c) \end{array}$$

$$\begin{array}{rcl} A_2 & = & \mathbf{ask}(true) \rightarrow \mathbf{tell}(a) \\ & & + \\ & & \mathbf{ask}(b) \rightarrow \mathbf{tell}(c) \\ & & + \\ & & \mathbf{ask}(true) \rightarrow \mathbf{tell}(a) \rightarrow \mathbf{ask}(b) \rightarrow \mathbf{tell}(c) \end{array}$$

It is easy to check that $\mathcal{IO}(A_1) = \mathcal{IO}(A_2)$. However, given the agent

$$B = \mathbf{ask}(a) \rightarrow \mathbf{tell}(b)$$

we have $\mathcal{IO}(A_1 \parallel B) \neq \mathcal{IO}(A_2 \parallel B)$ since $\langle true, c\rangle \in \mathcal{IO}(A_2 \parallel B) \setminus \mathcal{IO}(A_1 \parallel B)$.

The problem in the example above is that, when considering the input-output behavior, the (agent corresponding to the) third branch of the agent A_2 is equivalent to the union of the first two branches. Indeed, the control structure disappears and the third branch is represented by the two pairs $\langle true, a\rangle$ and $\langle b, c\rangle$.

5.4.1 *Reactive sequences*

De Boer and Palamidessi (1991a) have shown that the necessary branching information is encoded by a generalization of the input/output behavior in terms of *sequences* of pairs of assume/tell constraints. Such sequences are called reactive sequences and describe the interaction between a process and its environment. Reactive sequences will be represented by finite sequences of labeled constraints $s, s', \ldots$. Updates to the common store by the environment will be indicated by the label a (*assume*), while updates by the process itself will be labeled by t (*tell*). We use ℓ to range over $\{a, t\}$.

The *information content* of a reactive sequence s is the conjunction of all its constraints, and represents the store as determined by the sequence of events encoded by the sequence. The information content of s will be indicated by $con(s)$. Formally:

(i) $con(\lambda) = true$ (λ is the empty sequence), and
(ii) $con(c^{\ell} \cdot s) = c \wedge con(s)$.

R1′	$\langle \mathbf{Stop}, s\rangle \longrightarrow \langle ss, s\rangle$	
R2′	$\langle \mathbf{tell}(c), s\rangle \longrightarrow \langle ss, s\cdot c^t\rangle$	$con(s) \wedge c \neq false$
R3′	$\langle \mathbf{tell}(c), s\rangle \longrightarrow \langle \mathit{ff}, s\rangle$	$con(s) \wedge c = false$
R4′	$\langle \mathbf{ask}(c), s\rangle \longrightarrow \langle ss, s\rangle$	$c \leq con(s)$
R5′	$\langle \mathbf{ask}(c), s\rangle \longrightarrow \langle dd, s\rangle$	$c \not\leq con(s)$ *and* $con(s) \wedge c \neq false$
R6′	$\langle \mathbf{ask}(c), s\rangle \longrightarrow \langle \mathit{ff}, s\rangle$	$con(s) \wedge c = false$

Table 13 The transition system T': basic actions.

To define the compositional semantics based on reactive sequences we use a transition system T' (see Tables 13-16). The configurations are pairs $\langle A, s\rangle$, where s is a reactive sequence such that $con(s)$ is consistent. The difference with the transition system T consists mainly in Rule **R16′**, which models the interaction with the environment. Note that a process A is not immediately affected by actions made by the environment (only its future behavior will depend on them). An arbitrary constraint can be added (by the environment) consistently to the store without changing the state of A. In other words, A can make an *arbitrary assumption* (provided it is consistent) about the store. The other rules correspond to the rules of T. In the modified rules for the guarded choice the operation $\exists_x$ applied to a sequence s denotes the sequence obtained from s by applying $\exists_x$ to the constraints of s. Furthermore, $s_{|\ell}$ denotes the subsequence of s consisting of the constraints labeled by ℓ.

The correspondence between T and T' is expressed by the following lemma.

Lemma 5.5. *(de Boer and Palamidessi 1991a) Rules* **R1′**-**R15′** *of* T' *mimic Rules* **R1**-**R15** *of* T*, in the sense that if*

$$\langle A, s\rangle \longrightarrow \langle A', s'\rangle$$

$$\mathbf{R7'}\quad \frac{\langle g_i, s\rangle \longrightarrow \langle ss, s'\rangle}{\langle \sum_{i=1}^{n} g_i \to A_i, s\rangle \longrightarrow \langle A_i, s'\rangle}$$

$$\mathbf{R8'}\quad \frac{\langle g_n, s\rangle \longrightarrow \langle ff, s\rangle \quad \langle \sum_{i=1}^{n-1} g_i \to A_i, s\rangle \longrightarrow \langle \tau, s\rangle}{\langle \sum_{i=1}^{n} g_i \to A_i, s\rangle \longrightarrow \langle \tau, s\rangle} \qquad \tau \in \{ff, dd\}$$

$$\mathbf{R9'}\quad \frac{\langle g_n, s\rangle \longrightarrow \langle dd, s\rangle \quad \langle \sum_{i=1}^{n-1} g_i \to A_i, s\rangle \longrightarrow \langle \tau, s\rangle}{\langle \sum_{i=1}^{n} g_i \to A_i, s\rangle \longrightarrow \langle dd, s\rangle} \qquad \tau \in \{ff, dd\}$$

Table 14 The transition system T': guarded choice.

is a $\mathbf{Ri}'$ *transition step in* T', *then*

$$\langle A, con(s)\rangle \longrightarrow \langle A', con(s')\rangle$$

is a $\mathbf{Ri}$ *transition step in* T.

We show now that we can obtain a compositional semantics $\mathcal{M}$ by collecting, for every process, the sets of reactive sequences generated by the transition system T'. We need of course to consider all the completed sequences, and we will signal the way the process has terminated by postfixing a termination mode $\tau \in \{ss, ff, dd\}$. However, for a compositional treatment of failure in the presence of infinite processes, we need also to collect the sequences corresponding to unfinished computations. Such sequences will by postfixed by the symbol $\perp$.

Example 5.6. *Consider the process* $p(x)$, *with* $p(x)$:- $\mathbf{tell}(x = f(z)) \to p(z)$. *Since there are no terminating computations of* $p(x)$, *in the absence of a representation of unfinished computations, its semantics will be empty. However, when* $p(x)$ *is put in parallel with* $\mathbf{tell}(false)$, *we get as observable results all the pairs* $\langle \exists_z x = f^n(z), ff\rangle$, *where* $f^0(z) = f(z)$ *and* $f^{n+1}(z) = f(f^n(z))$.

R10′ $$\frac{\langle A, s\rangle \longrightarrow \langle A', s'\rangle}{\begin{array}{c}\langle A \parallel B, s\rangle \longrightarrow \langle A' \parallel B, s'\rangle \\ \langle B \parallel A, s\rangle \longrightarrow \langle B \parallel A', s'\rangle\end{array}}$$

R11′ $$\frac{\langle A, s\rangle \longrightarrow \langle ff, s\rangle}{\begin{array}{c}\langle A \parallel B, s\rangle \longrightarrow \langle ff, s\rangle \\ \langle B \parallel A, s\rangle \longrightarrow \langle ff, s\rangle\end{array}}$$

R12′ $$\frac{\langle A, s\rangle \longrightarrow \langle dd, s\rangle \quad \langle B, s\rangle \longrightarrow \langle dd, s\rangle}{\langle A \parallel B, s\rangle \longrightarrow \langle dd, s\rangle}$$

R13′ $$\frac{\langle A, (\exists_x s)\cdot d^t\rangle \longrightarrow \langle B, s'\rangle}{\langle \exists_x^d A, s\rangle \longrightarrow \langle \exists_x^{con(s')} B, s\cdot(\exists_x\, con(s'))^t\rangle} \qquad where\ s_{\restriction a} = s'_{\restriction a}$$

R14′ $$\frac{\langle A, (\exists_x s)\cdot d^t\rangle \longrightarrow \langle \tau, s'\rangle}{\langle \exists_x^d A, s\rangle \longrightarrow \langle \tau, s\cdot(\exists_x\, con(s'))^t\rangle}$$

R15′ $$\langle p(\bar{x}), s\rangle \longrightarrow \langle \Delta_{\bar{y}}^{\bar{x}} A, s\rangle \qquad p(\bar{y}) \text{ :- } A \text{ is in } D$$

Table 15 The transition system T': parallelism, hiding and procedure call.

R16′ $$\langle A, s\rangle \longrightarrow \langle A, s\cdot c^a\rangle \qquad c \wedge con(s) \neq false$$

Table 16 The transition system T': assumption rule.

So, despite our interest in finite computations only, we need to represent the finite approximations of infinite computations for a proper treatment of failure.

Definition 5.7. *The semantics $\mathcal{M}$ is defined as follows:*

$$\begin{array}{rcll} \mathcal{M}[\![A]\!] & = & \{s\cdot ss & \mid \langle A, true\rangle \longrightarrow^{+} \langle ss, s\rangle\} \\ & \cup & \{s\cdot ff & \mid \langle A, true\rangle \longrightarrow^{+} \langle ff, s\rangle\} \\ & \cup & \{s\cdot dd & \mid \langle A, true\rangle \longrightarrow^{+} \langle dd, s\rangle\} \\ & \cup & \{s\cdot\perp & \mid \langle A, true\rangle \longrightarrow^{+} \langle A', s\rangle\} \end{array}$$

The observables can be retrieved from $\mathcal{M}$ by first selecting those sequences entirely composed of tell constraints and ended by ss, ff, or dd. This amounts to requiring that the process has run in isolation and that the computation has reached a final state. Then we abstract from the particular order in which the constraints have been produced, and close under logical equivalence. This procedure is described by the operator *result*:

$$result(S) = \{\langle con(s), \tau\rangle \mid s\cdot\tau \in S \ \wedge\ s = c_1^t\cdot\ldots\cdot c_n^t \ \wedge\ \tau \in \{ss, ff, dd\}\}.$$

From Lemma 5.5 we obtain the correctness of the model with respect to our notion of observables:

Theorem 5.8. (Correctness of $\mathcal{M}$, *(de Boer and Palamidessi 1991a)***)**

$$\mathcal{O}(A) = result(\mathcal{M}[\![A]\!]).$$

The semantics $\mathcal{M}$ is denotational, i.e. compositional with respect to all the operators of the language. Here we show only the semantic counterparts of the parallel and the hiding operators, the interested reader can find the definition of the other operators in (de Boer and Palamidessi 1991a).

We first consider the parallel operator. A reactive sequence represents a particular sequence of reactions (encoded by the tell constraints) of a process to a particular sequence of actions (encoded by the assume constraints) performed by an hypothetic environment. The model $\mathcal{M}$ gives the set of all these possible sequences. Therefore it encodes already all the outcomes of all possible interactions with a parallel process. To obtain such outcomes the idea is to combine each sequence of the first process with each sequence of the second which is compatible, in the sense that at every point the action of the first corresponds to an assumption by the second and viceversa. Also, we allow the presence of corresponding assumptions on both sides, since they encode possible future interactions with another process. 'To correspond' here means to be relative to the same constraint; in fact our model is based on the assumption of a global store, hence at every step all processes must see the same store. In other words, the parallel operator 'unzips' the sequences, i.e. it verifies that the assumptions made by a process are validated by the other. The compatible sequences are then combined according to the principle that if one of the two constraint is a tell (i.e. it is produced by at least one of the two processes) then the result is a tell, otherwise it is an assume.

Concerning the termination modes, the result is success is both processes successfully terminate. It is deadlock if one deadlocks and the other either deadlocks or terminates, and it is failure if at least one fails. In fact the detection of a failure propagates through the whole system.

The parallel operator described above is similar to an interleaving operator. The difference is that it applies to sequences containing already all the information concerning the way in which processes interleave (the assumptions specify 'where' and 'what'). Such operator was introduced by Gerth *et al.* (1988), following ideas of Ever-Hadani (1987).

Formally we have the following definition.

Definition 5.9. *We define the (partial) parallel operator* $\|$ *on reactive sequences (together with a termination mode) as follows:*

(i) $s_1 \cdot \tau_1 \parallel s_2 \cdot \tau_2 = s_2 \cdot \tau_1 \;\parallel\; s_1 \cdot \tau_2$
(ii) $c^a \cdot s_1 \cdot \tau_1 \parallel c^\ell \cdot s_2 \cdot \tau_2 = c^\ell \cdot (s_1 \cdot \tau_1 \parallel s_2 \cdot \tau_2)$
(iii) $\tau \parallel ss = \tau$
(iv) $\tau \parallel \mathit{ff} = \mathit{ff}$
(v) $dd \parallel dd = dd$
(vi) $\perp \parallel \perp = \perp$

Since arbitrary assumptions can always be made, it is sufficient to consider the cases listed above, and to leave $\|$ *undefined on the sequences of different kind. The extension of the parallel operator to sets is defined in the obvious way:*

$$S_1 \,\tilde{\parallel}\, S_2 = \{s_1 \cdot \tau_1 \,\tilde{\parallel}\, s_2 \cdot \tau_2 \mid s_1 \cdot \tau_1 \in S_1 \;\wedge\; s_2 \cdot \tau_2 \in S_2\}$$

Next we introduce the semantic hiding operator. Intuitively, the hiding operator, applied to a set S of reactive sequences denoting a process, should filter out all those sequences which make assumptions on x, since the global information on x have no influence at the local level. Viceversa, the information on x contained in the tell constraints will be removed, since the local information produced on x will not be visible at the global level. In conclusion, for every sequence $s \in S$ the hiding operator will deliver all those sequences s' which have at every point the the same assume constraints, once the information on x is removed from the constraint in s', and the same tell constraints, after the information on x is removed from the constraint in s.

Definition 5.10. *Given a set of reactive sequences (together with a termination mode) we define:*

$$\exists_x(S) = \{s \mid \textit{there exists } s' \in S \textit{ s.t. } \exists_x(s_{\restriction a}) = s'_{\restriction a} \textit{ and } \exists_x(s'_{\restriction t}) = s_{\restriction t}\}$$

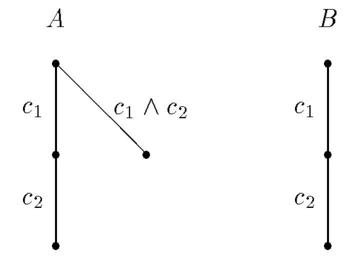

FIG. 7. Two agents observationally equivalent in every context

where $\exists_x s$ denotes the pointwise extension to (labeled) sequences of constraints of $\exists_x$.

By a straightforward case analysis of the transition system T' we have the following theorem:

Theorem 5.11. (Compositionality of $\mathcal{M}$, *(de Boer and Palamidessi 1991a)***)**

(i) $\mathcal{M}[\![A \parallel B]\!] = \mathcal{M}[\![A]\!] \parallel \mathcal{M}[\![B]\!]$,
(ii) $\mathcal{M}[\![\exists_x A]\!] = \exists_x(\mathcal{M}[\![A]\!])$.

5.4.2 *Full abstraction*

The compositional semantics $\mathcal{M}$ is not fully abstract with respect to the observables. Namely, it distinguishes processes which are observationally equivalent in any context. The problem is that sequences contain redundant information about the order and the granularity in which constraints are added to the store.

Example 5.12. *Let A and B be the agents shown in Figure 7 and defined as follows:*

$$\begin{array}{lcl} A & = & \mathbf{tell}(c_1) \rightarrow \mathbf{tell}(c_2) \\ & & + \\ & & \mathbf{tell}(c_1 \wedge c_2) \\ & & \\ B & = & \mathbf{tell}(c_1) \rightarrow \mathbf{tell}(c_2) \rightarrow \mathbf{Stop}. \end{array}$$

We have that $\mathcal{M}[\![A]\!] \neq \mathcal{M}[\![B]\!]$. Nevertheless they will produce the same final results in every context. This is because of the monotonicity of communication: we cannot define an environment that accepts both c_1 and c_2 and not $c_1 \wedge c_2$.

In general, the reaction of a process to a certain sequence of actions will only depend upon the information content of the sequence. Therefore, to obtain full abstraction we must eliminate distinctions between equivalent subsequences (equivalent in the sense of having the same information content). One way to do this is to *saturate* a set of denotations by adding, for any sequence s, all those sequences that differ only for an equivalent subsequence (**C1**). This operation, however, is not enough: since (**C1**) will introduce new possible interleaving points we need a rule (**C2**) which adds assumptions in order to make those interleaving points explicit for the parallel operator. These assumptions of course must not add any new information.

Definition 5.13. *Given a set of reactive sequences S, the saturation of S, $Sat(S)$, is the minimal set of sequences which contain S and satisfies the conditions* **C1** *and* **C2** *of Table 17.*

C1	if $s_1 \cdot c_1^\ell \ldots c_n^\ell \cdot s_2 \cdot \tau \in S$ and $con(s_1 \cdot c_1^\ell \ldots c_n^\ell) = con(s_1 \cdot d_1^\ell \ldots d_m^\ell)$ then $s_1 \cdot d_1^\ell \ldots d_m^\ell \cdot s_2 \cdot \tau \in S$
C2	if $s_1 \cdot s_2 \cdot \tau \in S$ and $con(s_1 \cdot s_2) = con(s_1 \cdot c \cdot s_2)$ then $s_1 \cdot c^a \cdot s_2 \cdot \tau \in S$

Table 17 The saturation conditions

We then define the fully abstract semantics A as follows:

$$\mathcal{A}[\![A]\!] = Sat(\mathcal{M}[\![A]\!]).$$

In (de Boer and Palamidessi 1991a) $\mathcal{A}$ is proved to be correct, compositional and fully abstract.

5.5 Reactive sequences versus closure operators

In this section we show that if we eliminate the consistency check, and we drop the distinction between success and deadlock, then the model $\mathcal{M}$ of the previous sections can be expressed in terms of Scott's *closure operators*. On this basis we prove the isomorphism of our restricted model with the model developed by Saraswat *et al.* (1991).

The consistency check requires to perform, before executing an action of the form **tell**(c), a test on the consistency of c with respect to the current store. If c is inconsistent with the current store then we choose another possibility, and only in case there are no other possibilities then a failure is generated. When the consistency check is not supported then **tell**(c) just adds c to the store regardless of whether or not it will lead to an inconsistent situation (*false*). The two kinds of tell are called *atomic tell* and *eventual tell* respectively in (Saraswat and Rinard 1990), and they clearly reflect the distinction between atomic and eventual unification. 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 regarded as the *most undesirable* situation, the *failure* of the computation.

Concurrent constraint without consistency check can be defined by forbidding the occurrence of a tell-guard in a plus context, thus eliminating escape possibilities when a failure is detected, and by eliminating some rules in the transition system (essentially the ones concerning failure). Note that, since tell is not anymore a guard, it must be introduced as basic action. Namely, **tell**(c) is now an agent.

The identification of deadlock and success in the observables means that we renounce to the possibility of distinguishing processes which have terminated and will not react to any 'stimulus', from processes which are just stuck, and which will still produce information once new input is provided.

Example 5.14. *Consider the two processes*

$$\begin{aligned} A_1 \quad = \quad & \mathbf{ask}(true) \rightarrow \mathbf{tell}(x = a \wedge y = b) \\ & + \\ & \mathbf{ask}(true) \rightarrow \mathbf{ask}(x = a) \rightarrow \mathbf{tell}(x = a \wedge y = b) \\ & + \\ & \mathbf{ask}(true) \rightarrow \mathbf{Stop} \end{aligned}$$

$$\begin{aligned} A_2 \quad = \quad & \mathbf{ask}(true) \rightarrow \mathbf{tell}(x = a \wedge y = b) \\ & + \\ & \mathbf{ask}(true) \rightarrow \mathbf{Stop} \end{aligned}$$

The semantics proposed by Saraswat et al. (1991) does not distinguishes between A_1 and A_2, whereas the semantics presented in previous sections does. In fact, the second branch of A_1 can generate deadlock (depending upon the store) whereas this possibility is excluded in A_2. If we would drop the distinction between success and deadlock then also our model would identify A_1 and A_2.

The identification of deadlock and success is maybe questionable from

rR1	$\langle \mathbf{tell}(c), d\rangle \longrightarrow \langle \mathbf{Stop}, c \wedge d\rangle$	
rR2	$\langle \sum_{i=1}^{n} \mathbf{ask}(c_i) \rightarrow A_i, d\rangle \longrightarrow \langle A_j, d\rangle$	$j \in [1, n]$ *and* $c_j \leq d$
rR3	$\dfrac{\langle A, c\rangle \longrightarrow \langle A', c'\rangle}{\begin{array}{c}\langle A \parallel B, c\rangle \longrightarrow \langle A' \parallel B, c'\rangle \\ \langle B \parallel A, c\rangle \longrightarrow \langle B \parallel A', c'\rangle\end{array}}$	
rR4	$\dfrac{\langle A, d \wedge \exists_x c\rangle \longrightarrow \langle B, d'\rangle}{\langle \exists_x^d A, c\rangle \longrightarrow \langle \exists_x^{d'} B, c \wedge \exists_x d'\rangle}$	
rR5	$\langle p(\bar{x}), c\rangle \longrightarrow \langle \Delta_{\bar{y}}^{\bar{x}} A, c\rangle$	$p(\bar{y})$:- A is in D

Table 18 The restricted Transition System rT

the practical point of view, but it allows to obtain an elegant representation of the reactive sequences in terms of closure operators. The rest of this section is devoted to investigate this characterization.

The transition system for concurrent constraint without consistency check and without deadlock recognition is given in Table 18. From it we can derive the notion of observables (still denoted by $\mathcal{O}$) in the same way as before.

5.5.1 *The restricted denotational model*

To define the compositional semantics we adopt the same 'pruning' (elimination of the rules for failure and deadlock) on the transition system T'. The new transition system, rT', is given in Table 19.

Based on rT' a semantics $\mathcal{M}$ is defined in the same way as before. All the results of previous section hold for $\mathcal{M}$ and $\mathcal{O}$. In particular, the compositionality of $\mathcal{M}$ and the correctness of $\mathcal{M}$ with respect to $\mathcal{O}$. Finally, we obtain the fully abstract model $\mathcal{A}$ for our sublanguage by applying the saturation operator *Sat* to $\mathcal{M}$. Observe that only one termination symbol remains there: *ss*. Thus we can drop it from the reactive sequences of $\mathcal{M}$ and $\mathcal{A}$.

rR1′	$\langle \mathbf{tell}(c), s\rangle \longrightarrow \langle \mathbf{Stop}, s \cdot c^t\rangle$	
rR2′	$\langle \sum_{i=1}^{n} \mathbf{ask}(c_i) \rightarrow A_i, s\rangle \longrightarrow \langle A_j, s\rangle$	$j \in [1, n]$ *and* $c_j \leq con(s)$
rR3′	$\dfrac{\langle A, s\rangle \longrightarrow \langle A', s'\rangle}{\begin{array}{c}\langle A \parallel B, s\rangle \longrightarrow \langle A' \parallel B, s'\rangle \\ \langle B \parallel A, s\rangle \longrightarrow \langle B \parallel A', s'\rangle\end{array}}$	
rR4′	$\dfrac{\langle A, d \wedge \exists_x s \cdot d^t\rangle \longrightarrow \langle B, s'\rangle}{\langle \exists_x^d A, s\rangle \longrightarrow \langle \exists_x^{con(s')} B, s \cdot (\exists_x con(s'))^t\rangle}$	
rR5′	$\langle p(\bar{x}), s\rangle \longrightarrow \langle \Delta_{\bar{y}}^{\bar{x}} A, s\rangle$	$p(\bar{y})$:- A is in D
rR6′	$\langle A, s\rangle \longrightarrow \langle A, s \cdot c^a\rangle$	

Table 19 The Transition System rT'

5.5.2 *Closure operators*

A closure operator on a lattice $\langle \mathcal{C}, \leq\rangle$ is a function $\Psi : \mathcal{C} \rightarrow \mathcal{C}$ which is *idempotent*, *monotonic* and *extensive*. More formally:

(i) $\forall x \in \mathcal{C}.\ \Psi(\Psi(x)) = \Psi(x)$,
(ii) $\forall x, y \in \mathcal{C}.\ x \leq y \Rightarrow \Psi(x) \leq \Psi(y)$,
(iii) $\forall x \in \mathcal{C}.\ x \leq \Psi(x)$.

Closure operators have the following nice property: they are completely determined by their range, which coincides with the set of their fixed points. In fact, if $R \subseteq \mathcal{C}$ is the range of a closure operator Ψ, then we can retrieve Ψ as follows

$$\Psi(x) = min\{y \in R \mid\ x \leq y\}.$$

We can also prove that, given a set R, if every subset of R which is bounded from below admits the least element, then R is the range of a closure operator. In the following, we will identify Ψ and its range.

Closure operators are used by Saraswat *et al.* (1991) to give semantics to determinate cc programming, namely cc programming without nondeterministic choice. A priori the absence of nondeterministic choice does not imply that the language is deterministic, because the parallel operator might re-introduce nondeterminism. In the case in which the consistency check is not supported, however, it is possible to prove a sort of confluence property, which causes a system to reach always the same final store from a given initial store. A determinate cc process can therefore be seen as a function from store to store. Furthermore, the way in which the process transforms the store satisfies the properties mentioned above, hence this function is actually a closure operator. The fact that closure operators can be represented as sets allows the semantic operators to be defined as operators on sets. In the case of the merge operator this is particularly convenient, in fact it just corresponds to set intersection.

Concerning the nondeterminate case, the behavior of a process cannot be described anymore as a *single* function from the initial store to the final store: we need at least a set of functions. This turns out to be sufficient to obtain compositionality (Saraswat *et al.* 1991). Actually it is sufficient to consider a special class of functions: the so-called bounded trace operators. A bounded trace operator (bto) is a closure operator Ψ which is defined on a sublattice of $\mathcal{C}$ of the form

$$\downarrow c = \{x \in \mathcal{C} \mid \ x \leq c\}$$

where c is an element of $\mathcal{C}$. Moreover, the inverse of Ψ, defined as

$$\Psi^{-1} = (\downarrow c \setminus \Psi) \cup \{c\}$$

is a closure operator on $\downarrow c$ as well. For the definition of the semantic operators on sets of bto's consult (Saraswat *et al.* 1991).

Saraswat *et al.* (1991) also obtain full abstraction by applying a sort of saturation condition: if the denotation of a process contains a bto Ψ, then it contains also all the bto's on the same domain which are superset of Ψ. In the next section we will show the intuition behind this.

5.5.3 *Correspondence between reactive sequences and bounded traces operators*

The following definition builds a mapping $\mathcal{F}$ from reactive sequences (without termination modes) to bto's.

Definition 5.15. *Let s be a modular sequence. We define the bto $\mathcal{F}(s)$ by induction on the length of s.*

(i) $\mathcal{F}(\lambda)(c) = c$,

(ii) $\mathcal{F}(c^t \cdot s)(c') = c' \wedge \mathcal{F}(s)(c)$,

(iii) $\mathcal{F}(c^a \cdot s)(c') =$ *if* $c \leq c'$ *then* $\mathcal{F}(s)(c)$ *else* c'.

It is easy to prove that this definition gives indeed a bto, with domain $\downarrow con(s)$.

We consider now the opposite function $\mathcal{G}$ which associates a reactive sequence to a bto.

Definition 5.16. *Let Ψ be a bto. We define the reactive sequence $\mathcal{G}(\Psi)$ as follows*

$$\Psi(true)^t \cdot \Psi^{-1}(\Psi(true))^a \cdot \Psi(\Psi^{-1}(\Psi(true)))^t \ldots$$

The sequence ends when the application of Ψ or Ψ^{-1} doesn't give any new contribution to the store, i.e. when the sequence reaches a fixpoint of Ψ and of Ψ^{-1}.

It is possible to prove that $\mathcal{F}$ is surjective and $\mathcal{G}$ is injective. Furthermore $\mathcal{F}(\mathcal{G}(\Psi)) = \Psi$, whereas $\mathcal{G}(\mathcal{F}(s))$ is in general different from s. We call $\mathcal{G}(\mathcal{F}(s))$ the *canonical representant* of s. The sequences s and $\mathcal{G}(\mathcal{F}(s))$ are equivalent with respect to the saturation conditions: each of them can generate the other by saturation.

Proposition 5.17. *Let s be a reactive sequence. Then*

(i) $s \in Sat(\mathcal{G}(\mathcal{F}(s)))$ *and*

(ii) $\mathcal{G}(\mathcal{F}(s)) \in Sat(s)$.

Another important property is the following: Consider two bto's Ψ and Φ such that $\Psi \subseteq \Phi$. Then $\mathcal{G}(\Phi)$ is a sequence which *asks more* than $\mathcal{G}(\Psi)$ in some points. Note that this corresponds to obtaining $\mathcal{G}(\Phi)$ from $\mathcal{G}(\Psi)$ by application of the saturation condition **C2**. Therefore we have that the application of *Sat* corresponds to the saturation by supersets (upward closure).

Proposition 5.18. *Let Ψ and Φ be two bto's. Then $\Psi \subseteq \Phi$ iff $\mathcal{G}(\Phi) \in Sat(\mathcal{G}(\Psi))$.*

From Propositions 5.17 and 5.18 we derive that the set of saturated sets of reactive sequences and the set of upward closed sets of bto's are isomorphic. This isomorphism is given by the extension of $\mathcal{F}$on sets. Of course, its inverse is given by the extension of $\mathcal{G}$ on sets. Furthermore, it is possible to show that this isomorphism is preserved by the semantics operators of the language, in the following sense:

Proposition 5.19. *For every operator op of the language, let $\widetilde{op}$ be the corresponding semantics operator of our approach, and $\widehat{op}$ the corresponding semantic operator as defined by Saraswat et al. (1991). Then, for $S_1, \ldots, S_n$ saturated sets of sequences we have*

$$\mathcal{F}(\widetilde{op}(S_1, \ldots, S_n)) = \widehat{op}(\mathcal{F}(S_1), \ldots, \mathcal{F}(S_n))),$$

and for $T_1, \ldots, T_n$ upward closed sets of bto's we have

$$\mathcal{G}(\widehat{op}(T_1, \ldots, T_n)) = \widetilde{op}(\mathcal{G}(T_1), \ldots, \mathcal{G}(T_n))).$$

From Proposition 5.19 we finally derive:

Theorem 5.20. *The fully abstract semantics $\mathcal{A}$ presented in previous sections and the semantics defined by Saraswat et al. (1991) are isomorphic.*

The representation of the semantics in terms of upward closed sets of bto's has the advantage that full abstraction is very easy to prove. Given a bto Ψ belonging to (the denotation of) a process A and not to B, consider a context $C[\,]$ consisting of the process $\{\Psi^{-1}\}$ in parallel, i.e. $C[\,] = \{\Psi^{-1}\} \parallel [\,]$. Then $\Psi \mathbin{\widehat{\parallel}} \Psi^{-1} = \Psi \cap \Psi^{-1}$ is a singletone c (the lub of their domain) and it constitutes an observable distinction, in fact $c \in C[A]$ and $c \notin C[B]$. To prove the last fact we use the upward closedness: if by contradiction $c \in C[B]$ then B would contain a bto $\Phi \subseteq \Psi$. But then, since B is upward closed, it would contain Ψ as well.

Acknowledgements

We are grateful to Jan Willem Klop for his helpful comments on previous versions of this paper.

Bibliography

1. Andreoli, J.-M. and Pareschi, R. (1991). Linear Objects: Logical processes with built-in inheritance. *New Generation Computing*, 9:445–473. Also in the Proc. of the Seventh International Conference on Logic programming, 1990, Jerusalem. The MIT Press.
2. Apt, K. (1990). Introduction to logic programming. In van Leeuwen, J., editor, *Handbook of Theoretical Computer Science*, volume B. Elsevier Science Publishers. Also available as Technical Report CS-R8826, Centre for Mathematics and Computer Science (CWI), Amsterdam, 1988.
3. Apt, K., Marchiori, E., and Palamidessi, C. (1992). A Theory of First-Order Built-in's of Prolog. In Kirchner, H. and Levi, G., editors, *Proc. of the Third Int. Conference on Algebraic and Logic Programming (ALP)*, volume 632 of *Lecture Notes in Computer Science*, pages 69–83. Springer-Verlag.
4. Bergstra, J. and Klop, J. (1984). Process algebra for synchronous communication. *Information and Control*, 60(1,3):109–137.
5. Bergstra, J. and Klop, J. (1986). Process algebra: specification and verification in bisimulation semantics. In *Mathematics and Computer Science II*, CWI Monographs, pages 61 – 94. North-Holland.

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

7. Brookes, S., Hoare, C., and Roscoe, W. (1984). A theory of communicating sequential processes. *Journal of ACM*, 31:499–560.

8. Clark, K. and Gregory, S. (1981). A relational language for parallel programming. Res. Report DOC 81/16, Imperial College, Dept. of Computing, London.

9. Clark, K. and Gregory, S. (1986). PARLOG: parallel programming in logic. *ACM Trans. on Programming Languages and Systems*, 8(1):1–49.

10. de Bakker, J. and Kok, J. (1988). Uniform abstraction, atomicity and contractions in the comparative semantics of Concurrent Prolog. In *Proc. of the International Conference on Fifth Generation Computer Systems*, pages 347–355, Tokyo, Japan. Institute for New Generation Computer Technology (ICOT). Extended Abstract, full version available as CWI report CS-8834.

11. de Bakker, J. and Kok, J. (1990). Comparative metric semantics for Concurrent Prolog. *Theoretical Computer Science*, 75(1/2):15–44.

12. de Boer, F. and Palamidessi, C. (1990a). Concurrent logic languages: Asynchronism and language comparison. In Debray, S. and Hermenegildo, M., editors, *Proc. of the North American Conference on Logic Programming*, Series in Logic Programming, pages 175–194. The MIT Press. Full version available as technical report TR 6/90, Dipartimento di Informatica, Università di Pisa.

13. de Boer, F. and Palamidessi, C. (1990b). On the asynchronous nature of communication in concurrent logic languages: A fully abstract model based on sequences. In Baeten, J. and Klop, J., editors, *Proc. of Concur 90*, volume 458 of *Lecture Notes in Computer Science*, pages 99–114, Amsterdam. Springer-Verlag. Full version available as Technical Report CSN 90/17, Technische Universiteit Eindhoven.

14. de Boer, F. and Palamidessi, C. (1991a). A Fully Abstract Model for Concurrent Constraint Programming. In Abramsky, S. and Maibaum, T., editors, *Proc. of TAPSOFT/CAAP*, volume 493 of *Lecture Notes in Computer Science*, pages 296–319. Springer-Verlag.

15. de Boer, F. and Palamidessi, C. (1991b). Embedding as a tool for language comparison: On the CSP hierarchy. In Baeten, J. and Groote, J., editors, *Proc. of CONCUR 91*, volume 527 of *Lecture Notes in Computer Science*, pages 127 – 141. Springer-Verlag.

16. de Boer, F. and Palamidessi, C. (1992). A process algebra for concurrent constraint programming. In Apt, K., editor, *Proc. of the Joint International Conference and Symposium on Logic Programming*, Series in Logic Programming, pages 463–477, Washington, USA. The MIT Press.

17. de Boer, F. and Palamidessi, C. (1994). Embedding as a Tool for Language Comparison. *Information and Computation* 108(1):128–157.

18. de Boer, F., Kok, J., Palamidessi, C., and Rutten, J. (1989). Control flow versus logic: a denotational and a declarative model for Guarded Horn Clauses. In Kreczmar, A. and Mirkowska, G., editors, *Proc. of the Symposium on Mathematical Foundations of Computer Science*, volume 379 of *Lecture Notes in Computer Science*, pages 165–176. Springer-Verlag.

19. de Boer, F., Kok, J., Palamidessi, C., and Rutten, J. (1991). Semantic models for Concurrent Logic Languages. *Theoretical Computer Science*, 86(1):3 – 33. A short version appeared on Proceedings of the Seventh International Conference on Logic Programming, Lisboa, 1989.

20. de Boer, F., Kok, J., Palamidessi, C., and Rutten, J. (1992). From failure to success: Comparing a denotational and a declarative semantics for Horn Clause Logic. *Theoretical Computer Science*, 101(2):239–263.

21. de Boer, F., Kok, J., Palamidessi, C., and Rutten, J. (1993). Non-monotonic concurrent constraint programming. *Proc. of the International Symposium on Logic Programming*, Vancouver, 1993.

22. Dijkstra, E. (1986). *A Discipline of Programming.* Prentice Hall, Englewood Cliffs, N.J.

23. Ever-Hadani, R. (1987). Operational and denotational linear sequence semantics for full Concurrent Prolog. Technical report, Department of Computer Science, The Weizmann Institute of Science, Rehovot, Israel. M.Sc. Thesis, 1992.

24. Falaschi, M., Levi, G., and Palamidessi, C. (1984). A synchronization logic: Axiomatics and formal semantics of generalized Horn clauses. *Information and Control*, 60(6):36–69.

25. Falaschi, M., Gabbrielli, M., Marriott, K., and Palamidessi, C. (1993). Compositional analysis for concurrent constraint programming. In *Proc. of the Eight Annual IEEE Symposium on Logic in Computer Science*, pages 210–221, Los Alamitos, California. IEEE Computer Society Press.

26. Gabbrielli, M. and Levi, G. (1990). Unfolding and fixpoint semantics for concurrent constraint logic programs. In Kirchner, H. and Wechler, W., editors, *Proc. of the Second Int. Conf. on Algebraic and Logic Programming*, Lecture Notes in Computer Science, pages 204–216, Nancy, France. Springer-Verlag.

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

28. Gerth, R., Codish, M., Lichtenstein, Y., and Shapiro, E. (1988). Fully abstract denotational semantics for Concurrent Prolog. In *Proc. of the*

Third Annual IEEE Symposium on Logic In Computer Science, pages 320–335. IEEE Computer Society Press, New York.

29. Gregory, S. (1987). *Parallel Logic Programming in PARLOG: the Language and its Implementation.* Addison-Wesley.

30. Harel, D. and Pnueli, A. (1985). On the development of reactive systems. In Apt, K. R., editor, *Logics and Models of Concurrent Systems.* Springer-Verlag, Berlin.

31. Haridi, S. (1990). A logic programming language based on the Andorra model. *New Generation Computing*, 7(2/3):109–125.

32. Haridi, S. and Brand, P. (1988). Andorra Prolog: an integration of Prolog and committed choice languages. In *Proc. of the International Conference on Fifth Generation Computer Systems*, pages 745–754, Tokyo. Institute for New Generation Computer Technology (ICOT).

33. Haridi, S. and Janson, S. (1990). Kernel Andorra Prolog and its computation model. In Warren, D. and Szeredi, P., editors, *Proc. of the Seventh International Conference on Logic Programming*, pages 31–48.

34. Haridi, S. and Janson, S. (1991). Programming paradigms of the Andorra Kernel Language. In Saraswat V. and Ueda K., editors, *Proc. of the International Logic Programming Symposium*, pages 167–186.

35. Haridi, S., Janson, S., and Palamidessi, C. (1992). Structural tranformational semantics for Andorra Kernel Prolog. *Future Generation Computer Systems*, 8(4):409–421.

36. Henkin, L., Monk, J., and Tarski, A. (1971). *Cylindric Algebras (Part I).* North-Holland.

37. Hoare, C. (1978). Communicating sequential processes. *Communications of the ACM*, 21(8):666–677.

38. Jacquet, J.-M. (1991). *Conclog: a Methodological Approach to Concurrent Logic Programming*, volume 556 of *Lecture Notes in Computer Science.* Springer-Verlag.

39. Jacquet, J.-M. and Monteiro, L. (1990). Comparative semantics for a Parallel Contextual Logic Programming Language. In Debray, S. and Hermenegildo, M., editors, *Proc. of the North American Conference on Logic Programming*, pages 195–214. The MIT Press.

40. Jacquet, J.-M. and Monteiro, L. (1991). Extended Horn Clauses: the framework and its semantics. In Baeten, J. and Groote, J., editors, *Proc. of CONCUR 91*, volume 527 of *Lecture Notes in Computer Science*, pages 281–297. Springer-Verlag.

41. Jacquet, J.-M. and Monteiro, L. (1992). Communication Clauses: towards synchronous communication in Contextual Logic Programming. In Apt, K., editor, *Proc. of the Joint International Conference and Symposium on Logic Programming*, Series in Logic Programming, pages 98–112.

The MIT Press.

42. Jaffar, J. and Lassez, J.-L. (1987). Constraint logic programming. In *Proc,. of ACM Symposium on Principles of Programming Languages*, pages 111–119. ACM, New York.
43. Kliger, S., Yardeni, E., Kahn, K., and Shapiro, E. (1988). The language FCP(:,?). In *Proc. of the International Conference on Fifth Generation Computer Systems*, pages 763–773, Tokyo. Institute for New Generation Computer Technology (ICOT).
44. Knijnenburg, P. and Kok, J. (1989). A compositional semantics for the finite failures of a language with atomized statements. In *Proc. Computer Science in the Netherlands (CSN 89)*.
45. Lassez, J. L. and Maher, M. J. (1984). Closures and fairness in the semantics of programming logic. *Theoretical Computer Science*, 29:167–184.
46. Levi, G. and Palamidessi, C. (1985). The declarative semantics of logical read-only variables. In *Proc. of the Third IEEE Symposium on Logic Programming*, pages 128–137, Boston. IEEE Computer Society Press.
47. Levi, G. and Palamidessi, C. (1987). An approach to the declarative semantics of synchronization in logic languages. In Lassez, J.-L., editor, *Proc. of the Fourth International Conference on Logic Programming*, pages 877–893, Melbourne. MTI Press.
48. Levi, G., Martelli, M., and Palamidessi, C. (1990). Failure and success made symmetric. In Debray, S. and Hermenegildo, M., editors, *Proc. of the North American Conference on Logic Programming*, Series in Logic Programming, pages 3–22. The MIT Press.
49. Levy, J. (1988). *Concurrent Prolog and Related Languages.* PhD thesis, Department of Computer Science, The Weizmann Institute of Science, Rehovot, Israel.
50. Lloyd, J. (1987). *Foundations of Logic Programming.* Springer-Verlag, second edition.
51. Maher, M. J. (1987). Logic semantics for a class of committed-choice programs. In Lassez, J.-L., editor, *Proc. of the Fourth International Conference on Logic Programming*, Series in Logic Programming, pages 858–876, Melbourne. The MIT Press.
52. Milner, R. (1980). *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science.* Springer-Verlag, New York.
53. Montanari, U. and Rossi, F. (1991). True concurrency semantics for concurrent constraint programming. In Saraswat, V. and Ueda, K., editors, *Proc. of the International Logic Programming Symposium*, Series in Logic Programming, pages 694–716, San Diego, CA. The MIT Press.

54. Monteiro, L. (1981). An extension to Horn Clause Logic allowing the definition of concurrent processes. In *Proc. of Formalization of Programming Concepts*, volume 107 of *Lecture Notes in Computer Science*, pages 401–407. Springer-Verlag.

55. Monteiro, L. (1982). A Horn Clause-like Logic for specifying concurrency. In *Proc. of the First Int. Conf. on Logic Programming*, pages 1–8.

56. Palamidessi, C. (1990). Algebraic properties of idempotent substitutions. In Paterson, M., editor, *Proc. of the 17th International Colloquium on Automata, Languages and Programming (ICALP)*, volume 443 of *Lecture Notes in Computer Science*, pages 386–399, Warwick, England. Springer-Verlag. Full version available as Technical Report TR-33/89, Dipartimento di Informatica, Università di Pisa.

57. Panangaden, P., Saraswat, V., Scott, P., and Seely, R. (1992). A Hiperdoctrinal view of Constraint Systems. In *Proceedings of the REX Workshop "Semantics: Foundations and Applications"*, Lecture Notes in Computer Science. Springer-Verlag.

58. Pereira, L. and Nasr, R. (1984). Delta-prolog: A distributed logic programming language. In *Proc. of the International Conference on Fifth Generation Computer Systems*, pages 283–291, Tokyo. Institute for New Generation Computer Technology (ICOT).

59. Plotkin, G. (1981). A structured approach to operational semantics. Technical Report DAIMI FN-19, Computer Science Department, Aarhus University.

60. Safra, S. (1986). Partial evaluation of concurrent prolog and its implications. Technical Report CS86-24, Dept. of Computer Science, The Weizmann Institute of Science, Rehovot, Israel. M.Sc. Thesis.

61. Saraswat, V. (1986). Problems with concurrent prolog. Technical Report CMU-CS-86-100, Computer Science Department, Carnegie-Mellon University.

62. Saraswat, V. (1987a). The concurrent logic programming language CP: definition and operational semantics. In *Conference Record of the Fourteenth Annual ACM Symposium on Principles of Programming Languages*, pages 49–63. ACM, New York.

63. Saraswat, V. (1987b). GHC: operational semantics, problems and relationship with CP($\downarrow, |$). In *Proc. of the Fourth IEEE Symposium on Logic Programming*, pages 347–358, San Francisco. IEEE, New York.

64. Saraswat, V. (1989). *Concurrent Constraint Programming.* PhD thesis, Carnegie-Mellon University. In ACM distinguished dissertation series. The MIT Press, 1993.

65. Saraswat, V. (1992). The Category of Constraint Systems is Cartesian-closed. In *Proc. of the Seventh Annual IEEE Symposium on Logic in*

Computer Science, Los Alamitos, California. IEEE Computer Society Press.

66. Saraswat, V. and Lincoln, P. (1992). Higher-order, linear concurrent constraint programming. Technical report, Xerox PARC.
67. Saraswat, V. and Rinard, M. (1990). Concurrent constraint programming. In *Proc. of the seventeenth ACM Symposium on Principles of Programming Languages*, pages 232–245. ACM, New York.
68. Saraswat, V., Rinard, M., and Panangaden, P. (1991). Semantics foundations of Concurrent Constraint Programming. In *Proc. of the eighteenth ACM Symposium on Principles of Programming Languages.* ACM, New York.
69. Scott, D. (1982). Domains for denotational semantics. In *Proc. of ICALP.*
70. Shapiro, E. (1983). A subset of Concurrent Prolog and its interpreter. Technical Report TR-003, Institute for New Generation Computer Technology (ICOT), Tokyo.
71. Shapiro, E. (1988). *Concurrent Prolog: Collected Papers.* Number Vol. 1-2 in Series in Logic Programming. The MIT Press, Cambridge, Mass.
72. Shapiro, E. (1989). The family of concurrent logic programming languages. *ACM Computing Surveys*, 21(3):412–510.
73. Shapiro, E. (1991). Separating concurrent languages with categories of language embeddings. In *Proceedings of the 23^{rd} Annual ACM Symposium on Theory of Computing*, pages 198–208.
74. Shapiro, E. (1992). Embeddings among concurrent programming languages. In Cleaveland, W., editor, *Proc. of CONCUR 92*, volume 630 of *Lecture Notes in Computer Science*, pages 486–503. Springer-Verlag.
75. Shapiro, E. (1986). Concurrent Prolog: A progress report. *Computer*, 19(8):44–58.
76. Takeuchi, A. and Furukawa, K. (1986). Parallel logic programming languages. In Shapiro, E., editor, *Proc. of the Third International Conference on Logic Programming*, volume 225 of *Lecture Notes in Computer Science*, pages 335–349, London. Springer-Verlag.
77. Ueda, K. (1987). Guarded Horn Clauses. In Shapiro, E. Y., editor, *Concurrent Prolog: Collected Papers*, Series in Logic Programming. The MIT Press.
78. Ueda, K. (1988). Guarded Horn Clauses, a parallel logic programming language with the concept of a guard. In Nivat, M. and Fuchi, K., editors, *Programming of Future Generation Computers*, pages 441–456. North Holland, Amsterdam.
79. van Emden, M. and de Lucena, G. (1982). Predicate logic as language for parallel programming. In Clark, K. and Tarnlund, S., editors, *Logic*

Programming, pages 189–198. Academic Press, London.

80. Yang, R. (1986). *A Parallel Logic Programming Language and its implementation.* PhD thesis, Keio University.
81. Yang, R. and Aiso, H. (1986). P-Prolog: a parallel language based on exclusive relation. In Shapiro, E., editor, *Proc. of the Third International Conference on Logic Programming*, volume 225 of *Lecture Notes in Computer Science*, pages 255–269, London. Springer-Verlag.