Partial Order and SOS Semantics for Linear Constraint Programs*

Eike Best[1], Frank S. de Boer[2] and Catuscia Palamidessi[3]

[1] Fachbereich Informatik, C.v.O.-Universität Oldenburg, Germany.
e.best@informatik.uni-oldenburg.de
[2] Vakgroep Informatica, Universiteit Utrecht, The Netherlands.
frankb@cs.ruu.nl
[3] DISI, Università di Genova, Italy. catuscia@disi.unige.it

Abstract. In this paper we consider linear constraint programming (lcp), a non-monotonic extension of concurrent constraint programming (ccp) which allows to remove information. The entailment relation of a linear constraint system, in terms of which linear constraint programs are defined, is based on the main underlying idea of linear logic: hypotheses in a logical derivation represent physical resources which are consumed, once used in the entailment relation.
We give a semantical analysis of this extension of ccp in terms of the causal relations among occurrences of basic actions (i.e. events). Using a partial order based history model, we define truly concurrent operational and partial order semantic models of lcp. They allow us to compare – and classify – various sublanguages of the proposed extension of ccp (including ccp itself) from the point of view of the degree of parallelism they generate. The two main results of the paper are consistency and completeness of the partial order model with respect to the operational semantics, and thus – as we will argue – its adequacy.

1 Introduction

In the concurrent constraint paradigm [18, 20, 21] (ccp, for short) parallel processes interact via a common store, represented by a constraint or a set of constraints, which expresses some partial information on the values of the variables involved in the computation. One of the most characteristic features of the ccp paradigm is a formalization of the basic operations which allow to update and to query the common store, in terms of the logical notions of consistency, conjunction and entailment supported by a given underlying constraint system. An update of the common store consists of adding (consistently) a constraint, whereas a query consists of checking whether the current store entails some constraint. Thus the computational model of ccp gives rise to a monotonic evolution of the store: more and more information is added in the course of the computation. Moreover, since the definition of the entailment relation of a constraint

* This work has been supported by the HCM Project EXPRESS and by the Esprit Research Project CALIBAN.

system is based on classical logic, it is difficult to envisage extensions of ccp which allow, for example, to *remove* constraints from the common store. Such extensions would greatly enhance the expressive power of ccp and would allow very compact solutions to programming problems which in a purely monotonic setting require considerably elaborate programming techniques.

One of the main contributions of [4] has been the introduction of an operator in ccp which consists of removing some constraint from the common store, and which thus gives rise to non-monotonic computations; the resulting paradigm is called *linear constraint programming* (lcp). The concept of store becomes thus similar to the notion of *blackboard* used in Linda [9, 5] and in Shared Prolog [2]. In the paper [1], it is shown that the underlying linear constraint system of lcp is closely related to a subclass of high-level Petri nets, and this relation is used to investigate the complexity of the entailment relation. In the present paper, correct and complete operational and partial order semantics of lcp are given. They are based on a specific history model which allows constraints, events, and their interdependencies, to be described formally.

The structure of this paper is as follows. In section 2 we define the notion of a linear constraint system, upon which rests the mechanism of information removal. Then, in section 3, we describe the lcp language as a suitable process algebra defined on top of a given linear constraint system. Parallel agents operate on a common store by adding and removing tokens. Adding tokens is modeled basically as in ccp by *tell* actions. Tokens can be removed by the execution of a *get* action of the form $get(c)$, where c is a token of the underlying linear constraint system. The execution of a $get(c)$ with respect to a given store consists of removing those tokens from which c can be derived. Thus in a sense we identify an occurrence of c with a multiset of tokens from which c can be derived. If the current store does not contain a multiset of tokens which entails c, then the get action suspends waiting for parallel processes to add the necessary information. On the other hand in case the store contains different multisets entailing c one is nondeterministically selected and subsequently removed.

In sections 4 and 5, we give a semantical analysis of the proposed non-monotonic extension of ccp in terms of the *causal* relations among occurrences of basic actions (*events*). Such a semantical description is of interest because it allows to treat parallelism as a primitive concept, whereas most semantical models describe parallelism in terms of interleaving and thus reduce it to a form of non-determinism.

In section 4 we define a history model with some specific features that allow executions of a linear constraint program to be described. In this model, the elements of a store (i.e. constraints) and the events, as well as their read/write interdependences, are represented explicitly. The set of histories associated with a linear constraint program is defined in SOS (structured operational semantics) style. By abstracting, we derive an operational semantics, which allows us to observe temporal order and simultaneity of actions, from the set of histories.

In section 5 we define our partial order model, the construction of which is based on the basic distinction between two kinds of causal relations among actions: dependencies based on the 'flow of control' and dependencies based on the 'flow of data'. The control dependencies can be derived in a straightforward manner from the structure of the program (they derive mainly from the sequencing operator). The data dependencies are obtained from the histories, 'at run-time', and are defined in terms of the entailment relation of the underlying linear constraint system. In order to evaluate the adequacy of the description of the causal dependencies we show correctness and completeness with respect to the observational semantics defined in section 4. This is the main difference with respect to other approaches to the true concurrent semantics of ccp [11, 12, 17]. In those works the adequacy of a partial order model is established with respect to an *interleaving* operational semantics.

Finally, in section 6, we show that our semantical model allows us to compare various sublanguages of the proposed extension of ccp (including ccp itself) from the point of view of their degree of parallelism.

Other approaches to combine ccp with linear logic concepts have been proposed in [8, 19]. In general, these works are directly based on the framework of linear logic, i.e. the operators of the language are interpreted as operators of linear logic, and its operational semantics is expressed in terms of the proof theory of linear logic. The approach of this paper, on the contrary, is based on a separation among *the data* (the constraints), whose theory has some linear logic flavour, and *the programming language*, whose operational semantics is defined in terms of a transition system in the usual SOS style. The advantage of our approach is that it gives more freedom in the definition of the language operators. For instance, it is not clear what would be the correspondent in linear logic of the typical ccp actions *ask* and *atomic tell*.

2 Linear constraint systems

As announced in the introduction, the aim is to model a constraint as a physical resource, which can be produced in several copies and each copy of which disappears once it is consumed, i.e. used as a hypothesis in the entailment relation. In order to represent the idea that the same resource can have multiple copies, it is natural to define the elements of the constraint system to be multisets instead than sets. Furthermore the underlying logic must be modified. The derivation of a (basic) constraint by the entailment relation *consumes the hypothesis*, hence, if the derivation of a constraint uses twice the same information, then this information must be represented twice in the hypothesis. Therefore the entailment relation must be defined from multisets of constraints to constraints.

Thus, we first recall some basic notions about multisets. Given a set $(c, d, \ldots \in)\, S$, a multiset of elements of S is a mapping $M: S \to \mathbf{N}$ where $\mathbf{N}$ is the set of natural numbers. In the following we will sometimes use the extensional notation: for instance, the multiset M such that $M(c) = 2$, $M(d) = 1$ and $M(e) = 0$ for $e \notin \{c, d\}$, will be represented by $M = \{c, c, d\}$. We denote by $\mathcal{M}(S)$ the set of the multisets on S, and by $\mathcal{M}_{\mathrm{F}}(S)$ the set of the *finite* multisets on S.

A useful notion is *multiset inclusion*, denoted by $\subseteq$, and defined as

$$M \subseteq M' \text{ iff } \forall c \in S.\ M(c) \leq M'(c).$$

The *multiset sum* is a (total) function $\oplus \colon \mathcal{M}(S) \times \mathcal{M}(S) \to \mathcal{M}(S)$ defined as

$$(M \oplus M')(c) = M(c) + M'(c).$$

The *multiset difference* is defined as a function $\ominus : \mathcal{M}(S) \times \mathcal{M}(S) \to \mathcal{M}(S)$ such that for each $c \in S$

$$(M \ominus M')(c) = \begin{cases} M(c) - M'(c) & \text{if } M(c) \geq M'(c) \\ 0 & \text{if } M(c) < M'(c). \end{cases}$$

Definition 1. [1, 4]

- A *basic linear constraint system* is a pair $\langle B, \vdash_B \rangle$ where B is a set of *basic constraints* and the *basic entailment relation* $\vdash_B \subseteq \mathcal{M}_f(B) \times B$ satisfies:
  (i) $\{c\} \vdash_B c$,
  (ii) if $\mu_1 \vdash_B c_1, \ldots, \mu_n \vdash_B c_n$ and $\{c_1, \ldots, c_n\} \vdash_B d$, then $\mu_1 \oplus \ldots \oplus \mu_n \vdash_B d$.
- The *linear constraint system* induced by $\langle B, \vdash_B \rangle$ is the structure $\langle \mathcal{M}(B), \vdash \rangle$, where $\vdash \subseteq \mathcal{M}(B) \times \mathcal{M}(B)$ is the minimal relation such that

$$\text{if} \quad \forall i \in I.\ \mu_i \vdash_B c_i \quad \text{then} \quad \bigoplus_{i \in I} \mu_i \vdash \bigoplus_{i \in I} \{c_i\}.$$

A linear constraint system does not have all the properties of a (usual) constraint system; it is not a complete algebraic lattice and not even a lattice (for instance it does not necessarily have a minimum element). However it is a preorder (see [1] for a proof), and therefore can be transformed into a partial order by factorising it with respect to the associated equivalence relation. The relation $\vdash$ is generally infinite, even when B is finite, and even when restricted to $\mathcal{M}_f(B)$ modulo equivalence.

3 The language

In order to compare our language with previous ccp paradigms, we consider a sort of 'superlanguage' $\mathcal{L}$ embodying the main mechanisms for communication and synchronization which have been proposed: the *eventual tell* (*tell*), the *atomic tell* (*atell*), the *ask*, and also the *removing ask* (*get*) which is our present proposal. A sublanguage $\mathcal{L}_G$ of $\mathcal{L}$ will be characterized by the set G of basic actions which are available in it. For instance, *atomic ccp* [20] corresponds to $\mathcal{L}_{\{atell, ask\}}$, and *eventual ccp* (or *ccp*, [20, 21]) corresponds to $\mathcal{L}_{\{tell, ask\}}$. In the last section we will compare these two languages and the language we propose, $\mathcal{L}_{\{tell, ask, get\}}$, from the point of view of potential parallelism and expressiveness.

We assume given a linear constraint system $\mathbf{L} = \langle \mathcal{M}(S), \vdash \rangle$, with tokens $c, d, \ldots \in S$ and elements $M, M' \ldots \in \mathcal{M}(S)$. The description of the language is parametric with respect to it.

The actions listed above have an argument which is a token. We could be more general and admit a finite multiset as argument, but this would give rise to some technical complications when building the partial order model of the language, possibly deflecting the attention from what we consider to be the main ideas and results of that construction. We will indicate later how this generalisation can be incorporated.

The basic actions operate on a common *store*, which ranges over $\mathcal{M}(S)$. Intuitively, the effect of $tell(c)$ consists of simply adding (in the multiset sense) the token c to the current store. The action $atell(c)$ has the same effect, but it can be executed only if the resulting store is consistent. Otherwise, we say that $atell(c)$ is *not enabled*. The action $ask(c)$ is a test on the current store and its execution does not modify the store: $ask(c)$ is *enabled* in the store M iff $M' \vdash c$, for some $M' \subseteq M$. The action $get(c)$ differs from $ask(c)$ because, when enabled, it removes information from the current store. More precisely, it removes from the current store a multiset of tokens $d_1, \ldots, d_m$ that entail c.

The following grammar defines the syntax of the language $\mathcal{L}$. Every program P consists of a (possibly empty) list of procedure declarations D followed by an agent A. In the syntax, g stands for one of the four elementary actions mentioned above.

$$\begin{array}{lll} \textit{Programs} & P & ::= \; A \mid D\,;P \\ \textit{Declarations} & D & ::= \; p(x) \,\text{:-}\, A \\ \textit{Agents} & A & ::= \; \textit{stop} \mid g \rightarrow A \mid A + A \mid A \,\|\, A \mid p(x). \end{array}$$

The agent *stop* represents successful termination. An agent $g \rightarrow A$ executes g, if g is enabled, and then it behaves like A. If g is not enabled, then execution is *suspended*, waiting for other (parallel) agents to add information to the store. The choice operator in $A + B$ selects nondeterministically one of the agents A and B (if it is enabled, that is, the current store allows it to perform one of its initial actions). If neither the agent A nor B is enabled then the composite agent $A + B$ is suspended. Parallel composition is represented by $\|$. Finally, the agent $p(x)$ is a procedure call, where p is the name of the procedure and x is the actual parameter. The meaning of $p(x)$ is given by a procedure declaration of the form $p(y) \,\text{:-}\, A$, where y is the formal parameter.

The parameter passing is modeled by renaming the formal parameter y in the body A by the actual parameter x. This is formally expressed by substituting every token c occurring in A by $c[x/y]$. The result of this substitution we denote as $A[x/y]$. Concerning the local variables of the body, we assume a mechanism which takes care of renaming them into fresh ones, thus avoiding possible clashes with the global variables of the store.

4 Operational semantics and the history model

In this section we introduce a transition system which allows the construction of a partial order model describing the *causal dependencies* and the *weak causal* *dependencies* among (occurrences) of actions. In order to describe the *data dependencies* among actions, we represent a history as a pattern of occurrences of those actions which have been executed, together with the resources (constraints) they have produced or consumed. This history associates with each action the set of its 'causes', namely, those actions in the store that enabled it. As a model of histories, we use a combination of the *causal nets* of net theory [15] and the *composets* of Janicki and Koutny [6]. Our nets are a particular case of *contextual nets*, which have been introduced by Montanari and Rossi [14] and used in [13] for modeling ccp.

Formally, a *history* is an object (R, E, λ, F, G) where R are the resources, E are the events ($R \cap E = \emptyset$), λ is a labelling function from R to S (set of constraints or, as we will say exchangeably, tokens) and from E to the set $\{\mathit{tell}(c), \mathit{atell}(c), \mathit{ask}(c), \mathit{get}(c) \mid c \in S\}$, the relation F is a subset of $(R \times E) \cup (E \times R)$, and the relation G is a subset of $(R \times E)$ such that $(F \cup G)^\star$ is a partial order relation on $(R \cup E)$ and $F \cap (R \times E)$ and $F^{-1} \cap (R \times E)$ are partial functions. Two histories are isomorphic if they differ only in the (names of) elements of R and E. As usual, we consider isomorphic histories as equal.

The basic intuition behind the labelling is as follows. $\lambda(r)$ denotes the constraint associated with the resource r; a constraint can be produced and consumed several times during an execution, and resources are a means of distinguishing these productions and consumptions. Similarly, $\lambda(e)$ denotes the action of which e is an execution. There may be many events which are executions of the same action.

The basic intuition of the relations F and G is as follows. $(e, r) \in F$ could be read as 'event e produces the resource r'; this could be true, for instance, if $\lambda(e)$ is a $\mathit{tell}(c)$ and $\lambda(r)$ is c. $(r, e) \in F$ could be read as 'event e consumes the resource r'; this could be true, for instance, if $\lambda(e)$ is $\mathit{get}(c)$ and $\lambda(r)$ belongs to a multiset M over S such that $M \vdash c$. $(r, e) \in G$ could be read as 'event e reads the resource r'; this could be true, for instance, if $\lambda(e)$ is $\mathit{ask}(c)$ and $\lambda(r)$ belongs to a multiset M such that $M \vdash c$.

We use the letter χ for histories, and we use decorations consistently. Thus, if χ_1 is some history, then R_1, E_1, etc., are the sets belonging to χ_1, and if χ' is a history then R', E', etc. are its sets, and so on. The empty history is defined as $\emptyset = (\emptyset, \emptyset, \emptyset, \emptyset, \emptyset)$. The union of two histories, $\chi_1 \cup \chi_2$, is defined as the componentwise union of the constituent sets. For a history χ, the substructure (R, E, F) (without λ and G) is a causal net in the Petri net sense [15].

Example 1. Let $\chi_1 = (R_1, E_1, \lambda_1, F_1, G_1)$, such that $R_1 = \{r, \bar{r}\}$, $E_1 = \{e_1, e_2, e_3\}$, $\lambda_1 = \{(r, c), (\bar{r}, \bar{c}), (e_1, \mathit{tell}(c)), (e_2, \mathit{ask}(c)), (e_3, \mathit{get}(c))\}$, $F_1 = \{(e_1, r), (r, e_3), (e_3, \bar{r})\}$ and $G_1 = \{(r, e_2)\}$. This history is shown on the left-hand side of Figure 1, with elements of F_1 being drawn as solid arrows and elements of G_1 (actually, only one such element) as broken arrows. Intuitively speaking, it represents a history in which a $\mathit{tell}(c)$ action (actually, more precisely, an event representing an execution of a $\mathit{tell}(c)$ action) first produces constraint c (actually, more precisely, a resource representing a constraint c) which is then both read (by an $\mathit{ask}(c)$ action on account of $\{c\} \vdash c$) and removed (by a $\mathit{get}(c)$ action, also on account

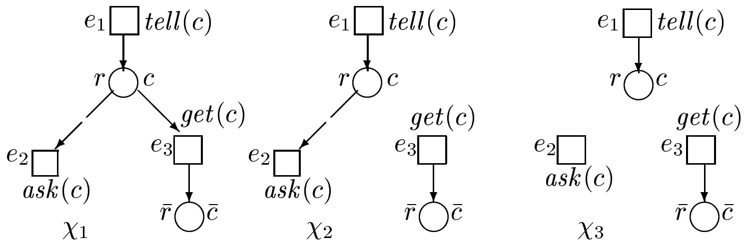

Fig. 1. Three histories χ_1 (left), χ_2 (middle) and χ_3 (right)

of $\{c\} \vdash c$). If the underlying constraint system would also allow the inference $\emptyset \vdash c$ (which might be the case as nothing in the definition of a constraint system prevents it) then a history χ_2 – shown in the middle of Figure 1 – which differs from χ_1 only by deleting the element (r, e_3) from F_1, is also a history consisting of a *tell*(c) action followed by an *ask*(c) action (on account of $\{c\} \vdash c$) and a *get*(c) action (on account of $\emptyset \vdash c$). The history χ_3 arising from χ_2 by deleting (r, e_2) from G_2 – shown on the right-hand side of Figure 1 – is then also a history describing a *tell*(c), an *ask*(c) and a *get*(c), where all three events are concurrent.

We will call a resource r in a history χ *unconsumed* iff $r \notin dom(F)$. We call the set of unconsumed resources $R_u(\chi)$ or just R_u if the particular history χ to which it refers is understood. The *store* associated with χ, called *store*(χ), is the multiset

$$store(\chi) \quad = \quad \bigoplus_{r \in R_u(\chi)} \{\lambda(r)\} \ .$$

Notice that this is indeed an element of $\mathcal{M}(S)$, where the constraints are counted as many times as they occur as labels of unconsumed resources. Later, for reasons of brevity, we abbreviate such a multiset by $\lambda(R_u(\chi))$.

We now give the operational semantics of the language using as basic configurations pairs $\langle A, \chi \rangle$ where A is an agent and χ is a history. Since the intended model is *true concurrent*, we allow also transitions which represent concurrent execution of more than one action, provided they are independent, that is, *steps* denoted by α. Hence in general a transition will be labeled by multisets of actions. Thus a transition has the following format:

$$\langle A, \chi \rangle \xrightarrow{\alpha} \langle A', \chi' \rangle$$

where A and A' are agents, α is a finite multiset of actions *tell*(c), *atell*(c), *ask*(c), and *get*$(c) \mid c \in S\}$, and χ and χ' are histories. The basic intuition underlying this transition can be described as follows:

> A, with a starting history χ, evolves by executing the step α into A', with a resulting history χ' (of which χ will be a prefix).

A1 $\langle \mathit{tell}(c) \to A, \chi \rangle \xrightarrow{\{\mathit{tell}(c)\}} \langle A, \chi \cup \chi_{new} \rangle$

where $\chi_{new} = (\{r\}, \{e\}, \{(r, c), (e, \mathit{tell}(c))\}, \{(e, r)\}, \emptyset)$,
and r and e are not already in χ.

A2 $\langle \mathit{atell}(c) \to A, \chi \rangle \xrightarrow{\{\mathit{atell}(c)\}} \langle A, \chi \cup \chi_{new} \rangle$

where $\chi_{new} = (\{r\}, \{e\}, \{(r, c), (e, \mathit{atell}(c))\}, \{(e, r)\}, \{(r_1, e), \ldots, (r_n, e)\})$
and r and e are not already in χ,
and $r_1, \ldots, r_n$ are all unconsumed resources of χ,
and $\mathit{store}(\chi \cup \chi_{new})$ is consistent.

A3 $\langle \mathit{ask}(c) \to A, \chi \rangle \xrightarrow{\{\mathit{ask}(c)\}} \langle A, \chi \cup \chi_{new} \rangle$

where $\chi_{new} = (\emptyset, \{e\}, \{(e, \mathit{ask}(c))\}, \emptyset, \{(r_1, e), \ldots, (r_n, e)\})$,
and e is not already in χ,
and $r_1, \ldots, r_n \in R_u(\chi)$ such that $\{\lambda(r_1), \ldots, \lambda(r_n)\} \vdash c$.

A4 $\langle \mathit{get}(c) \to A, \chi \rangle \xrightarrow{\{\mathit{get}(c)\}} \langle A, \chi \cup \chi_{new} \rangle$

where $\chi_{new} = (\{\bar{r}\}, \{e\}, \{(e, \mathit{get}(c)), (\bar{r}, \bar{c})\}, \{(r_1, e), \ldots, (r_n, e), (e, \bar{r})\}, \emptyset)$,
and e and $\bar{r}$ are not already in χ,
and $r_1, \ldots, r_n \in R_u(\chi)$ such that $\{\lambda(r_1), \ldots, \lambda(r_n)\} \vdash c$.

Table 1. Basic Actions.

R1 $\dfrac{\langle A, \chi \rangle \xrightarrow{\alpha} \langle A', \chi' \rangle}{\langle A + B, \chi \rangle \xrightarrow{\alpha} \langle A', \chi' \rangle}$ and $\dfrac{\langle B, \chi \rangle \xrightarrow{\beta} \langle B', \chi' \rangle}{\langle A + B, \chi \rangle \xrightarrow{\beta} \langle B', \chi' \rangle}$

R2 $\dfrac{\langle A, \chi \rangle \xrightarrow{\alpha} \langle A', \chi' \rangle \;,\; \langle B, \chi \rangle \xrightarrow{\beta} \langle B', \chi'' \rangle}{\langle A \| B, \chi \rangle \xrightarrow{\alpha \oplus \beta} \langle A' \| B', \chi' \cup \chi'' \rangle}$ where $\chi' \cup \chi''$ is a history and $\mathit{atell}(\alpha) = \emptyset \vee \mathit{atell}(\beta) = \emptyset$

R3 $\dfrac{\langle A[y/x], \chi \rangle \xrightarrow{\alpha} \langle A', \chi' \rangle}{\langle p(y), \chi \rangle \xrightarrow{\alpha} \langle A', \chi' \rangle}$ where $p(x)$:- A is the declaration for p.

Table 2. The transition system L.

$$\mathbf{I}\ \langle A,\chi\rangle \xrightarrow{\emptyset} \langle A,\chi\rangle$$

Table 3. The idling step.

Table 1 describes the transition rules for the basic actions. The rules for the choice and parallel operator and recursion are given in Table 2. Table 3 shows an auxiliary rule. The rules are explained in turn.

A $\mathit{tell}(c)$ action (Rule **A1**) gives rise to a new event e that neither reads nor consumes resources ($G(e)=\emptyset$ and $F^{-1}(e)=\emptyset$), but produces a new resource r labelled c (by $F(e)=\{r\}$). The $\mathit{atell}(c)$ is like the $\mathit{tell}(c)$ except that it also reads (but does not consume) all unconsumed resources, since the entire store has to be checked for consistency. The $\mathit{ask}(c)$ reads, but does not consume, resources that entail c.

The $\mathit{get}(c)$ operation is more involved. If we would be interested in the subset of the language with no atomic tell, then $\mathit{get}(c)$ could be modeled like $\mathit{ask}(c)$ except that after reading it also consumes the resources. Formally, the distinction would only be in the use of the relation G for the former and F for the latter. However, in the presence of atomic tell, things are more complicated. For instance, consider the agent

$$(\mathit{tell}(c) \rightarrow \mathit{get}(c) \rightarrow \mathit{stop}) \parallel (\mathit{atell}(d) \rightarrow \mathit{stop})$$

where d is inconsistent w.r.t. c. If the first action to be executed is $\mathit{tell}(c)$, then $\mathit{atell}(d)$ can only be executed after $\mathit{get}(c)$. One way to enforce this dependency is to assume that $\mathit{get}(c)$ "produces" a special token, which we call $\bar{c}$, on which $\mathit{atell}(d)$ will depend. This special token is of course neutral w.r.t. the constraint system, in the sense that it neither entails, nor is entailed by, any other contraint.[4]

Note that for *tell* we have essentially a disjoint union of histories while the union is disjoint in the case of *ask* and *get* only if $n=0$ (we do allow this special case), and for *atell* it is disjoint only if its initial store $\mathit{store}(\chi)$ is empty.

The more general case that the actions do not have a single constraint, but a finite multiset of constraints as parameters, can be incorporated in these rules as follows. Let us only consider the action $\mathit{tell}(\{c_1,\ldots,c_n\})$. The rule **A1** needs to be modified such that a single new event e and n new resources $r_1,\ldots,r_n$

[4] Another possibility would be to consider inhibitory arcs ([7]). The idea would be that a resource c inhibits every $\mathit{atell}(d)$ with d inconsistent w.r.t. c. Then the dependency of an action $\mathit{atell}(d)$ upon an action $\mathit{get}(c)$ would be derived by an analysis of the inhibitory relation and the F relation. This solution is perhaps more intuitive and natural, but would complicate the derivation of the dependency relation, hence we do not consider it here.

with labels $c_1, \ldots, c_n$ are added to χ, and the new F relation consists of all pairs (e, r_i) for $1 \leq i \leq n$. The other rules can be extended similarly (where rules **A3** and **A4** need no real extension).

The most interesting rule of Table 2 is **R2**, for concurrent composition. The other two rules **R1** and **R3** are essentially standard. Note that Rule **R2** has the same initial history, χ, for both A and B in its premise. This models the fact that the store is shared and that two parallel programs A and B pursue parts of a common history. The premise of **R2** allows A to change χ into a new history χ' and B to change χ into a possibly different history χ'', but χ is a common prefix of both, and therefore also of the – elementwise – union $\chi' \cup \chi''$ which is the result of $A \| B$, evolving in parallel from χ. It is assumed that the events and resources newly chosen in the separate evolutions of A and B are different; for instance, if A starts with a $tell(c)$ action and B also starts with a $tell(c)$ action, then the two new events chosen according to the two applications of rule **A1** are different (although their labels are the same), and the same is true for the two new resources (whose labels are, of course, also the same, namely c).

Note that the requirement that $\chi' \cup \chi''$ be a history amounts to require that F and F^{-1} in $\chi' \cup \chi''$ be partial function. In general this would not be guaranteed. For instance, if in the premise of Rule **R2**, A and B both execute a $get(c)$ action, and χ contains only one unconsumed resource r labelled c. Then A would add a new event e' with $(r, e') \in F'$ in χ' and B would add another event e'' with $(r, e'') \in F''$ in χ''. The above condition forces the unconsumed resources R_u of χ to be partitioned in R_A and R_B, so that the resources consumed in the step $\langle A, \chi \rangle \xrightarrow{\alpha} \langle A', \chi' \rangle$ belong to R_A while the resources consumed in the step $\langle B, \chi \rangle \xrightarrow{\beta} \langle B', \chi'' \rangle$ belong to R_B.

The other condition (applied recursively) implies that all *atell* actions in the computation will be sequentialised. This is necessary because by definition an atomic tell should always prevent unconsistency of the store, hence we cannot allow two atomic tell to be executed in parallel since they could add inconsistent constraints to the store.[5]

Note that Rule **R2**, as it stands, forces all parallel agents to make a transition step at the same time (synchronous cooperation). In order to allow each component to proceed 'at its own speed', we allow an agent to perform an *idling step*, which is described by the axiom **I** of Table 3. Intuitively, an idling step represents the fact that the agent 'skips a turn'.

The behaviour of the get operation can be best understood in terms of a 'ring architecture', in which tokens circulate freely among the nodes where the process reside. Note that in this way get actions can be executed in parallel since disjoint access to the common store is automatically guaranteed.

[5] In our language inconsistency might still arise by the execution of an atomic tell in parallel with an eventual tell. This is not very elegant, but it is only for the sake of generality that we are considering a syntax which has both kinds of tell. In general, in a language, it does not make sense to have both atomic and eventual tell.

Example 2. Consider the agent

$$tell(c) \to ((ask(c) \to stop) \| (get(c) \to stop)).$$

If the underlying constraint system does not contain the entailment $\emptyset \vdash c$, then χ_1 (given previously as an example) is a possible history of this agent, starting with the empty history. If, on the other hand, the underlying constraint system does allow $\emptyset \vdash c$, then χ_2 and χ_3, in addition to χ_1 (as defined previously), are also possible histories of that agent, starting with $\emptyset$.

Note that the operational rules only add more elements to a given history and that they do not remove elements. Thus, in an evolution, histories grow monotonically, as it should be.

¿From the transition system L we derive the operational semantics and the notion of observable. In this paper, we restrict ourselves to finite maximal behavious only.

Definition 2. The operational semantics of an agent A, $\mathcal{O}(A)$, is defined as follows:

$$\mathcal{O}(A) \;=\; \{\, \chi \mid \langle A, \emptyset \rangle \xrightarrow{\alpha_1} \ldots \xrightarrow{\alpha_n} \langle B, \chi \rangle \not\longrightarrow \,\}\;,$$

where $\langle B, \chi \rangle \not\longrightarrow$ indicates that the agent B, starting with history χ, can perform only idling steps.

Note that due to the possible existence of deadlocks, a maximal history is not necessarily a complete one (i.e. it might be the case that not all processes have evolved into *stop*).

Definition 3. The *observables* of an agent A, $Obs(A)$, are defined as follows:

$$Obs(A) \;=\; \{\, \alpha_1 \ldots \alpha_n \mid \langle A, \emptyset \rangle \xrightarrow{\alpha_1} \ldots \xrightarrow{\alpha_n} \langle B, \chi \rangle \not\longrightarrow \,\}.$$

Note that we are assuming that the observer is capable to observe actions which are performed at the same time (represented as a multiset). The semantics $Obs(A)$ represents a significant abstraction from $\mathcal{O}(A)$; all resources, intermediate stores, and identities of events are forgotten.

5 Control flow and partial order semantics

Histories χ describe the data dependencies between events, and the operational semantics restricts histories to those that start with the initially empty history and end in a final state. These histories do not describe the control dependencies. For instance, the agent $A = tell(c) \to (tell(c) \to stop)$ generates in $\mathcal{O}(A)$ the unique (up to renaming of resources and events) final history

$$\chi = (\{r, r'\}, \{e, e'\}, \{(r, c), (r', c), (e, tell(c)), (e', tell(c))\}, \{(e, r), (e', r')\}, \emptyset)$$

which contains two concurrent events, while control flow dictates that the events should be in sequence.[6]

We proceed as follows. First we give a definition which allows the control dependencies to be obtained statically from an agent, again in the form of a partial order or a set of partial orders η with a simpler structure than χ. The two partial orders χ and η are not immediately comparable, but we will make them comparable in a second step by extending η and restricting χ suitably. Then, in a third step, the partial order model for agents is constructed by combining control dependency orders η and data dependency orders χ, essentially by forming the union of the two respective relations.

As announced, control dependencies can be derived statically from the structure of the agent. We use *partial words* [23] which can also be viewed as traces [10]. A partial word, for our purposes, is a triple $(E, \lambda, \prec)$ such that E is a (finite) set of events, λ is a function from E to the set $\{tell(c), atell(c), ask(c), get(c)\}$ (both with the same interpretation as before), and $\prec$ is a partial order on E called the *precedence relation.* $\emptyset = (\emptyset, \emptyset, \emptyset)$ is the empty partial word, and union is defined componentwise. Disjoint union between two partial words, $\eta_1 \uplus \eta_2$, is defined by first making the two event sets disjoint (if they are not already) and then forming the union. As with histories, we consider isomorphic partial words as equal.

Definition 4. Let A be a agent. Define $\mathcal{C}(A)$ as the smallest collection of partial words satisfying the following five properties:

(1) If $A = stop$ then $\mathcal{C}(A) = \{(\emptyset, \emptyset, \emptyset)\}$.
(2) If $A = g \to A'$ then $\mathcal{C}(A) \;=\; \{(\emptyset, \emptyset, \emptyset)\} \cup g.\mathcal{C}(A')$,
where for some set $\mathcal{E}$ of partial words, we define
$g.\mathcal{E} \;=\; \{(\{e_g\} \uplus E, \{(e_g, g)\} \cup \lambda, (\{(e_g, e) \mid e \in E\} \cup \prec)) \mid (E, \lambda, \prec) \in \mathcal{E}\}$,
that is, we prefix every partial word in $\mathcal{E}$ by a new event labelled g.
Notice that $g.\{(\emptyset, \emptyset, \emptyset)\} = \{(\{e_g\}, \{(e_g, g)\}, \emptyset)\}$.
(3) If $A = A_1 + A_2$ then $\mathcal{C}(A) = \mathcal{C}(A_1) \cup \mathcal{C}(A_2)$.
(4) If $A = A_1 \| A_2$ then $\mathcal{C}(A) = \{\eta_1 \uplus \eta_2 \mid \eta_1 \in \mathcal{C}(A_1) \wedge \eta_2 \in \mathcal{C}(A_2)\}$.
(5) If $A = p(y)$ and $p(x)$:- A' is the declaration of p then $\mathcal{C}(A) = \mathcal{C}(A'[y/x])$.

Lemma 5. (Well-definedness) *Set set $\mathcal{C}(A)$ is well defined.*

Proof. (Sketch.) The only interesting case is (5). We may limit ourselves to immediate recursion, i.e. recursion at most in a single process. Other kinds of recursion (two or more processes calling themselves recursively) can be treated by standard methods. Let $A = p(y)$ with $p(x) : -A'$. Let $A'' = A'[y/x]$. Define a function f, depending on A'', from sets-of-partial-words to sets-of-partial-words, by

$$f(\mathcal{E}) = \mathcal{C}(A'')[\mathcal{E}[z/y]/p(z)],$$

[6] Note, however, that $Obs(A_0)$ contains a two-element sequence in addition to a one-element sequence.

where $[\mathcal{E}[z/y]/p(z)]$ means that the occurrences of $p(z)$ in A'' are substituted, in the computation of $\mathcal{C}(A'')$, by $\mathcal{E}$. Then f is monotonic since $\mathcal{E}_1 \subseteq \mathcal{E}_2$ clearly implies $\mathcal{C}(A'')[\mathcal{E}_1[z/y]/p(z)] \subseteq \mathcal{C}(A'')[\mathcal{E}_2[z/y]/p(z)]$. Hence f has a unique minimal fixpoint, which in case recursion is guarded is the limit of the sequence $\emptyset$, $f(\emptyset)$, $f(f(\emptyset))$, ... Define $\mathcal{C}(A)$ as this least fixpoint of f and observe that this satisfies the properties in the previous definition. □

Now, we turn to combining data flow objects in $\mathcal{O}(A)$ (which are histories χ) with control flow objects in $\mathcal{C}(A)$ (which are partial words η) to yield the partial order semantics of A. First, consider a given history χ. Our intention is to abstract from the resources R and to lift F and G to a precedence between the events of χ. However it is not reasonable to abstract χ directly to a partial word, for the reason illustrated by the following example.

Example 3. Consider again the agent

$$\mathit{tell}(c) \to ((\mathit{ask}(c) \to \mathit{stop}) \| (\mathit{get}(c) \to \mathit{stop})).$$

Assume $\emptyset \not\vdash c$. Then the observational semantics of this agent contains the sequences $\{\mathit{tell}(c)\}\{\mathit{ask}(c), \mathit{get}(c)\}$ and $\{\mathit{tell}(c)\}\{\mathit{ask}(c)\}\{\mathit{get}(c)\}$, but not the sequence $\{\mathit{tell}(c)\}\{\mathit{get}(c)\}\{\mathit{ask}(c)\}$.

As is well known [6, 7], partial words are not rich enough to discriminate such behaviour. Following [7] (and similarly to [17], but with differences as discussed there), we introduce *weak precedence* as an additional form of precedence. Thus, we associate with $\chi = (R, E, \lambda, F, G)$ an object $\xi = \xi(\chi) = (E, \lambda, \prec, \gamma)$, which we call an *event history*, defined by:

$$\begin{aligned} e \prec_{aux} e' &\iff \exists r \in R\colon (e, r) \in F \wedge (r, e') \in F \cup G \\ e\, \gamma_{aux}\, e' &\iff \exists r \in R\colon (r, e) \in G \wedge (r, e') \in F \end{aligned}$$

as well as $\prec = \rho^* \circ \prec_{aux} \circ \rho^*$ and $\gamma = \rho^+$ with $\rho = (\prec_{aux} \cup \gamma_{aux})$. The relations $\prec$ and γ are called *precedence* and *weak precedence*, respectively. Note that by this definition, weak precedence is induced when some resource is read by some event and consumed by another event. The intuition is that if $e \prec e'$ then e occurs before e' and if $(e, e') \in \gamma$ then e' does not occur before e. (We will make this intuition precise later, in the definition of the set *Lin*.)

For an arbitrary event history $\xi = (E, \lambda, \prec, \gamma)$, the substructure $(E, \lambda, \prec)$ (without γ) is a partial word and the substructure $(E, \prec, \gamma)$ (without λ) is a relational structure in the sense of [7]. It turns out that for agents A, all associated structures are always stratified in the sense of [7], i.e. they are composets in the sense of [6].

By these definitions, data dependency partial orders and control dependency partial orders are made comparable and can now be combined. We first need to introduce the following definition: Given an arbitrary event history $\xi = (E_d, \lambda_d, \prec_d, \gamma_d)$ and an arbitrary partial word $\eta = (E_c, \lambda_c, \prec_c)$, let $f\colon E_d \to E_c$ be a bijection. Then ξ and η are compatible with respect to f, *compatible*(ξ, f, η), iff f is label-preserving, that is, $\forall e \in E_d \colon \lambda_d(e) = \lambda_c(f(e))$ and, moreover, f is weakly order preserving, i.e. the transitive closure of the relation

$$\prec_d \cup \gamma_d \cup (f \circ \prec_c \circ f^{-1})$$

is irreflexive. Intuitively, this means that the function f identifies the events of ξ and η in a consistent way, i.e. the union of the dependency relations does not contain cycles.

If $compatible(\xi, f, \eta)$ holds, then the combination $combine(\xi, f, \eta)$ is well-defined and it is the following event history:

$$combine(\xi, f, \eta) \quad = \quad (\, E_d \,,\, \lambda_d \,,\, (\prec_d \cup (f \circ \prec_c \circ f^{-1}))^+ \,,\, \gamma_d \,).$$

Definition 6. The partial order semantics of an agent A, $\mathcal{P}(A)$, is defined as follows:

$$\begin{aligned} \mathcal{P}(A) \quad = \quad \{\, \xi \mid\ & \exists \chi \in \mathcal{O}(A), f, \eta \in \mathcal{C}(A): \\ & compatible(\xi(\chi), f, \eta) \wedge \xi = combine(\xi(\chi), f, \eta) \,\}. \end{aligned}$$

Notice that all elements of $\mathcal{P}(A)$ are finite.

The question now arises whether the above definition of the set of partial orders associated with an agent adequately captures the degree of parallelism. More precisely, the question is whether the degree of parallelism given by the partial order model is neither too restrictive nor too liberal. As we have mentioned, we will answer this question by comparing the semantics $\mathcal{P}(A)$ with the observational semantics $Obs(A)$.

Formally, we define the set of all linearisations of a (finite) event history $\xi = (E, \lambda, \prec, \gamma)$ as follows:

$$\begin{aligned} Lin(\xi) \quad = \quad \{\, & \lambda(X_1) \dots \lambda(X_m) \mid \\ & \textstyle\bigcup_{i=1}^{m} X_i \ = E \ \text{ and } \ X_1, \dots, X_m \text{ are mutually disjoint} \\ & \text{and } \ e \prec e' \text{ implies } e \in X_i, e' \in X_j \text{ for some } i < j \\ & \text{and } \ (e, e') \in \gamma \text{ implies } e \in X_i, e' \in X_j \text{ for some } i \leq j \,\} \,. \end{aligned}$$

If ξ is empty then $Lin(\xi) = \{\varepsilon\}$. Note that the definition of $Lin(\xi)$ treats the two precedence relations $\prec$ and γ of ξ slightly differently according to their interpretation given above. Notice also that every element of the set $Lin(\xi)$ is a finite sequence of multisets of actions (since this set is the codomain of λ), and hence the two sets of sequences $Lins(A)$ and $Obs(A)$ are comparable, where

$$Lins(A) = \bigcup_{\xi \in \mathcal{P}(A)} Lin(\xi).$$

Correctness of the partial order model, i.e., the guarantee that it is not too liberal, is ensured by the fact that all linearisations of a partial order associated to an agent represent valid observations.

Theorem 7. (Correctness) *For any agent A we have $Lins(A) \ \subseteq \ Obs(A)$.*

Proof. (Sketch.) Let $\alpha_1 \dots \alpha_n \in \mathit{Lins}(A)$. Then there exists an event history $\xi \in \mathcal{P}(A)$ with $\alpha_1 \dots \alpha_n \in \mathit{Lin}(\xi)$. $\xi \in \mathcal{P}(A)$ means that there is a history $\chi \in \mathcal{O}(A)$, a function f and a partial word $\eta \in \mathcal{C}(A)$ such that they are compatible and ξ is their combination. Such η represents a possible abstract unfolding of the agent, abstract in the sense that the guards are not tested. The fact that there is an event history compatible with η implies that there is a way, compatible with such unfolding, of executing the actions of η so that they will produce and consume the resources in the way needed to activate the guards. Thus we derive from η one of the possible (maximally parallel) computations of the agent, from which we derive $\alpha_1 \dots \alpha_n$ as an observable. $\square$

The reverse inclusion which we call *completeness*, guarantees that our partial order model is not too restrictive.

Theorem 8. (Completeness) *For any agent A we have* $\mathit{Obs}(A) \subseteq \mathit{Lins}(A)$.

Proof. (Sketch.) Let $\alpha_1 \dots \alpha_n \in \mathit{Obs}(A)$. Then there exists some χ such that $\langle A, \emptyset \rangle$ goes by $\alpha_1 \dots \alpha_n$ to $\langle A, \chi \rangle$. We need to combine χ with some good (i.e. compatible) $\eta = (E, \lambda, \prec)$, to yield a good ξ. Such an η can be constructed by enriching the operational rules with a way of construction (inductively) also this η, and then applying this enriched set of rules to the above sequence $\alpha_1 \dots \alpha_n$ which yields χ out of the empty history. For instance, we may define rules such as

$$\langle \mathit{tell}(c) \to A, \eta \rangle \xrightarrow{\{\mathit{tell}(c)\}} \langle A, \eta_{new} \rangle,$$

where η_{new} is defined in an analogous way as the corresponding history. We may then define η by $\langle A, \emptyset \rangle \xrightarrow{\alpha_1} \dots \xrightarrow{\alpha_n} \langle A, \eta \rangle$; note that given $\alpha_1 \dots \alpha_n$, this is a good definition, though in order to be able to combine the η so obtained with χ, $\alpha_1 \dots \alpha_n$ may need to be re-linearised using the silent step axiom (I). $\square$

A similar result is stated in [11], but with an essential difference: in [11] the observations of an agent consist of sequences of basic actions generated by an *interleaving* operational semantics (and, correspondingly, linearisations are sequences of actions instead of sequences of sets of actions).

We argue that, having an operational semantics which allows us to observe simultaneity of actions, our result is stronger. In fact, in an analogous process algebraic setting, consider the agent $A = a \parallel b$, where a and b are some basic actions. If we would consider an interleaving operational semantics, then it would follow that a partial order semantics $\mathcal{P}'(A)$ containing only the two posets with $a \prec b$, and $b \prec a$, is correct and complete (according to the formulation of completeness in [11]). On the other hand, having the possibility to observe the simultaneity of actions, we have a way to rule out that semantics as *incomplete.*

However, also our formulation of completeness does not identify an *optimal* semantics. More precisely, consider the following definition:

Definition 9. Let $\mathcal{P}'$ be a function which maps agents into sets of partial orders. We say that $\mathcal{P}'$ is a *correct interpretation* iff for any agent A, $\mathit{Lins}(A) \subseteq \mathit{Obs}(A)$ holds.

Now, if we take the interpretation

$$\bar{\mathcal{P}}(A) = \bigcup\{\mathcal{P}'(A) \mid \mathcal{P}' \text{ is a correct interpretation}\}$$

we have that $\bar{\mathcal{P}}$ is correct and complete in the above sense, but it does not coincide (in general) with the model $\mathcal{P}$. For instance, in the case of $A = a \parallel b$, the model $\mathcal{P}$ would contain only the poset where a and b are not related, whereas $\bar{\mathcal{P}}$ contains also the other two posets with with $a \prec b$ and $b \prec a$.

However the model $\mathcal{P}$ should be 'more general'. In what sense precisely is a topic of future research.

6 Language comparison

The model we have presented in the previous sections allows us to compare the various sublanguages of $\mathcal{L}$ from the point of view of potential parallelism. In particular we are interested in comparing ccp and atomic ccp with $\mathcal{L}_{\{tell,ask,get\}}$, which we call ccp$_{get}$.

We can see that ccp$_{get}$ presents an intermediate degree of parallelism: less than ccp, but more than atomic ccp. This is formally justified by the following proposition:

Proposition 10. *Let A be an agent, let $\chi \in \mathcal{O}(A)$, let $\xi(\chi) = (E, \lambda, \prec, \gamma)$, let $e, e' {\in} E$ and $g{=}\lambda(e), g'{=}\lambda(e')$. Then*

- *If g, g' are atomic tell actions, then either $e \prec e'$ or $e' \prec e$.*
- *If g, g' are tell actions, then neither $e \prec e'$ nor $e' \prec e$.*
- *If g, g' are ask actions, then neither $e \prec e'$ nor $e' \prec e$.*

This proposition states that two atomic tell actions are always executed in interleaving, and two tell or two ask actions are always independent, hence they can be performed simultaneously. Concerning two get actions, they can be always independent (which happens when the resources they need are disjoint), or not. In the second case, they can still be performed in parallel if the store is 'rich enough' to entail both of them.

Concerning other dependencies, ask and get actions depend in general on some tell or atomic tell actions. Atomic tell actions can depend on some get actions. In fact the removal of information might eliminate some item which is inconsistent with the constraint to be added. Note that a get action is also capable to restore consistency.

The presence of the ask actions in ccp$_{get}$ might seem redundant, since $ask(c)$ would be equivalent, concerning the results in all possible contexts, to two consecutive actions $get(c)$ and $tell(c)$. However, on the basis of the above discussion, we see that from the point of view of the distributed efficiency it is more convenient to use $ask(c)$.

Let us now consider also one of the main important features of a programming language: its expressiveness. With no doubts, atomic ccp is more expressive than ccp (for a formal proof see [3]), but the necessary sequentialisation of atomic tell actions is a serious drawback for distributed implementation. On the other hand, ccp is very suitable for distributed implementation, but many distributed algorithms, which can be easily expressed in atomic ccp, require complicated adaptations in ccp, and it is often necessary to use an arbiter or some kind of centralized control. We argue that ccp_{get} could be a reasonable compromise also in terms of expressiveness.

Let us consider the problem of the dining philosophers, which, according to Dijkstra, can be regarded as a benchmark of expressiveness. This problem has a very simple distributed solution in atomic ccp, which can be obtained by translating the Concurrent Prolog program proposed by Shapiro [22]. In ccp, a solution can be obtained by translating the PARLOG program proposed by Ringwood [16]. This solution, which is rather complicated (70 lines of code), uses a series of agents to implement each state a philosopher can occupy–thinking, preparing to eat, and eating. Philosophers circulate along data streams among these process-states. The agents representing the states can be regarded as a sort of central entity which control the overall situation.

In ccp_{get} a simple, natural and distributed solution can be obtained as follows. Assuming that there are n philosophers, we initialize the store with $n-1$ tokens, representing available tickets, and n tokens representing the forks. When a philosopher wants to eat, he must first get a ticket. If he succeeds (i.e. if there are tickets still available), then he can get the forks, when available. When he has eaten, he releases (by tell actions) forks and ticket. This algorithm, which is inspired by [5], is proved to be deadlock-free.

Acknowledgments

We wish to acknowledge helpful comments by Walter Vogler and by the anonymous reviewers.

References

1. E. Best and C. Palamidessi: Linear constraint systems as high-level nets. Proc. CONCUR'96. Springer LNCS **1119** (1996) 498–513
2. A. Brogi and P. Ciancarini: The concurrent language Shared Prolog. ACM TOPLAS **13** (1991) 99–123
3. F.S. de Boer and C. Palamidessi. Embedding as a Tool for Language Comparison. Information and Computation **108** (1994) 128–157
4. F.S. de Boer, C. Palamidessi and E. Best: Concurrent Constraint Programming with Information Removal. Proc. First Int. Workshop on Concurrent Constraint Programming, Venice (1995) 1–13
5. N. Carriero and D. Gelernter: Linda in context. Comm. ACM **32** (1989) 445–458
6. R. Janicki and M. Koutny: Structure of Concurrency. Theoretical Computer Science **112** (1993) 5–52
7. R. Janicki and M. Koutny: Semantics of Inhibitor Nets. Information and Computation **123** (1995) 1–16

8. N. Kobayashi and A. Yonezawa: ACL–A Concurrent Linear Logic Programming Paradigm. Proc. of the International Logic Programming Symposium (1993) 279–294
9. D. Gelernter: Generative Communication in Linda. ACM TOPLAS **7** (1985) 80–112
10. A. Mazurkiewicz: Trace Theory. Petri Nets: Applications and Relationships to Other Models of Concurrency, Advances in Petri Nets 1986, Part II. Springer-Verlag, LNCS **255** (1987) 279-324
11. U. Montanari and F. Rossi: True concurrency in concurrent constraint programming. Proc. of the International Logic Programming Symposium (1991) 694–716
12. U. Montanari and F. Rossi: Graph rewriting for a partial order semantics of concurrent constraint programming. Theoretical Computer Science **109** (1993)
13. U. Montanari and F. Rossi: Concurrent Semantics for Concurrent Constraint Programming via Contextual Nets. V.J. Saraswat and P. Van Hentenryck (eds.) Constraint Programming (1995)
14. U. Montanari and F. Rossi: Contextual nets. Acta Informatica **32** (1995)
15. W. Reisig: Petri Nets. An Introduction. EATCS Monographs on Theoretical Computer Science **3** (1985)
16. G.A. Ringwood: Parlog86 and the dining logicians. Comm. ACM **31** (1988) 10–25
17. F. Rossi: *Constraints and Concurrency.* PhD thesis, University of Pisa (1993)
18. V.A. Saraswat: *Concurrent Constraint Programming.* PhD thesis, Carnegie-Mellon University (1989) Published by The MIT Press (1993)
19. V.A. Saraswat and P. Lincoln: Higher-order, linear concurrent constraint programming. Technical report, Xerox PARC (1992)
20. V.A. Saraswat and M. Rinard: Concurrent constraint programming. Proc. of the seventeenth ACM Symposium on Principles of Programming Languages (1990) 232–245
21. V.A. Saraswat, M. Rinard, and P. Panangaden: Semantics foundations of concurrent constraint programming. In *Proc. of the eighteenth ACM Symposium on Principles of Programming Languages* (1991) 333–353
22. E. Shapiro: Embedding Linda and other joys of concurrent logic programming. Technical Report, The Weizmann Institute of Science (1989)
23. P.H. Starke: Processes in Petri Nets. Elektronische Informationsverarbeitung und Kybernetik **17** (1981) 389–416