PROCEEDINGS OF THE INTERNATIONAL CONFERENCE
ON FIFTH GENERATION COMPUTER SYSTEMS 1988,
edited by ICOT.

UNIFORM ABSTRACTION, ATOMICITY AND CONTRACTIONS IN THE COMPARATIVE SEMANTICS OF CONCURRENT PROLOG

J.W. de Bakker
J.N. Kok

Centre for Mathematics and Computer Science,
P.O. Box 4079, 1009 AB Amsterdam, The Netherlands

ABSTRACT

This paper shows the equivalence of two semantics for a version of Concurrent Prolog with non-flat guards: an operational semantics based on a transition system and a denotational semantics which is a metric semantics (the domains are metric spaces). We do this in the following manner: First a uniform language $\mathcal{L}$ is considered, that is a language where the atomic actions have arbitrary interpretations. For this language we prove that a denotational semantics is correct with respect to the operational semantics. This result relies on Banach's fixed point theorem. Techniques stemming from imperative languages are used. Then we show how to translate a Concurrent Prolog program to a program in $\mathcal{L}$ by selecting certain basic sets for $\mathcal{L}$ and then instantiating the interpretation function for the atomic actions. In this way we induce the two semantics for Concurrent Prolog and the equivalence between the two semantics.

Note: this work was carried out in the context of ESPRIT 415: Parallel Architectures and Languages for Advanced Information Processing – a VLSI-directed approach and in the context of LPC: the dutch National Concurrency Project, supported by the Netherlands Organization for Scientific Research (N.W.O.), grant 125-20-04.

1 Introduction

'Pure' logic programming (LP) has by now a well-established semantic theory, described in, e.g. [Lloyd 1987], [Apt 1987] or [Apt and van Emden 1982]. Traditionally, at least three varieties of semantics are distinguished, viz. the 'declarative', 'procedural' and 'process' interpretations, and, for pure LP, it is a standard result that these semantics all coincide. For logic *programming languages* – with the emphasis now on programming *language* rather that on the underlying mathematical framework of pure LP – the situation is much less clear.

Parallelism in LP languages brings along the well-known (from the field of imperative languages) phenomena such as synchronization, suspension and deadlock, sending and receiving of messages, and process creation. Accordingly, it may be more advantageous to address the semantic issues in parallel LP following the tradition in imperative languages (emphasizing 'control') rather that that of pure LP (emphasizing 'logic').

For operational semantics the method of Structured Operational Semantics ([Plotkin 1981]) has become the standard tool. Systems of (possibly labeled) *transitions* are embedded into syntax directed deductive systems, providing a concise, powerful and flexible tool, as demonstrated by numerous applications (for parallel logic languages we mention [Saraswat 1987]). For denotational semantics, a classification can be based on the underlying mathematical structures, thus leading to (at least) order-theoretic, metric, algebraic and category-theoretic models. We use metric structures as our main tool.

A well-known phenomenon from imperative concurrency is that of *deadlock* (in LP returning as *failure*), inducing the need for a model which embodies more structure than just (sets of) sequences. In case programming notions requiring branching time are combined with state transformations, the need for Plotkin's *resumptions* arises. We have developed our own (metric) way of solving domain equations which are at the bottom of such resumptions (described in [de Bakker and Zucker 1982] or [America and Rutten 1988]). The introduction of committed-choice in parallel logic languages is a cause of deadlock (see for example [Falaschi and Levi 1988] for an analysis of this phenomenon).

In [Kok 1988] we developed a denotational semantics for a version of Concurrent Prolog, employing the metric techniques (domains of processes in the resumptions style, contracting functions etc.) of [de Bakker and Zucker 1982] and successors. The branching structure built up as result of a computation before a *commit* is encountered, is collapsed, at the moment of such an encounter, into a set of streams. The paper [Gerth et al. 1988] develops, for the language TFCP, operational and denotational semantics, the latter based on failure sets. Moreover, a fully abstractness theorem relating the two is presented. The third investigation we mention fol-

lows the approach of declarative semantics. In [Levi and Palamidessi 1985] and [Levi and Palamidessi 1987], a comprehensive analysis is provided of a number of synchronization mechanisms in parallel logic languages.

In this paper we develop an operational and a denotational semantics for a language $\mathcal{L}$. This language is *uniform* in the sense that some basic sets can have arbitrary interpretations. Another feature of $\mathcal{L}$ is that we have an operator that turns its argument (any, possibly complex statement s) into an elementary action or (control) *atom*, denoted by $|s|$. Hence our emphasis on *atomicity* in this investigation. We provide a proof of the correctness of the denotational semantics with respect to the operational semantics (we show that there exists a restriction operator which relates the two). The operational semantics $\mathcal{O}$ is based on a transition system. The denotational semantics $\mathcal{D}$ is a metric semantics: the domains are metric spaces. A key role is played by *contractions*: they are used in almost all definitions. We have used *uniform abstraction*: In order to obtain the two semantics for Concurrent Prolog, we interpret the abstract sets of $\mathcal{L}$. For example, the set of elementary actions B will be the set of pairs (a_1, a_2) of (logical) atoms. The intended meaning of such a pair (a_1, a_2) of atoms is that we have to unify a_1 and a_2. We then show how to translate a Concurrent Prolog program to a program in the uniform language $\mathcal{L}$. The denotational model that is induced in this way (from the denotational model for $\mathcal{L}$) resembles the model given in [Kok 1988]. We also have an induced operational semantics and an induced relation between the two semantics. Figure 1 shows the relations. Note that the heavy lines in this figure refer to induced mappings only.

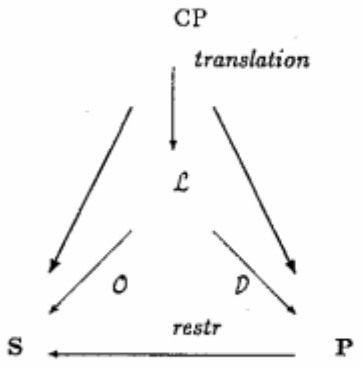

Figure 1: overview of the models

We think that the uniform abstraction procedure of first giving semantics to a uniform language and then the interpretation, gives more insight into the model. Moreover, we have the automatic link with an operational model.

The idea of a translation of Concurrent Prolog is already present in [Beckman 1986]. In that paper a translation to Milner's CCS (Calculus of Communicating Systems) is provided. The recursion structure that is used in the paper is different: a clause is modeled by an agent which tries continuously to apply itself. In our model the equivalent of clauses is (recursive) procedures. The model in [Beckman 1986] is based on synchronous communication, which is not present in our model.

We treat a larger subset of Concurrent Prolog than [Gerth et al. 1988]. The main difference is that we allow non-flat guards. This leads to more complex semantic domains: we have to introduce the notion of atomicity. One of the nice points of [Gerth et al. 1988] is that it makes clear what can be observed from a Concurrent Prolog program: for example that we can distinguish between failure and deadlock. They prove that their semantics is fully abstract with respect to the operational semantics. A point of further research is whether or not non-flat guards influence these results. Following Apt and Plotkin ([Apt and Plotkin 1986]) we recall that in the case of unbounded nondeterminism (caused by non-flat guards) it might be impossible to assign a fully abstract semantics.

We give an outline of the rest of our paper. Metric topological preliminaries are given in section 2. Section 3 describes the language $\mathcal{L}$ and section 4 its operational semantics $\mathcal{O}$. In section 5 the denotational semantics is defined. Section 6 gives the relationship between $\mathcal{O}$ and $\mathcal{D}$. Finally, section 7 provides the translation from Concurrent Prolog to $\mathcal{L}$.

2 Metric Preliminaries

We give in this section some basic definitions and properties about metric spaces. Let $\mathbf{N}$ be the set of natural numbers. For the notions of (complete) (ultra)metric space, compact and closed sets, Cauchy sequence we refer to [Engelking 1977].

Definition 2.1 *Let (M_1, d_1), (M_2, d_2) be metric spaces. Let $0 \leq c < 1$. The set of functions f from M_1 to M_2 that satisfy*

$$\forall x, y \in M[d_2(f(x), f(y)) \leq c.d_1(x, y)]$$

we call contracting.

Theorem 2.2 (Banach's fixed point theorem) *Let (M, d) be a complete metric space and $f : M \to M$ a contracting function. Then there exists an $x \in M$ such that the following holds:*

1. $f(x) = x$ *(x is a fixed point of f),*
2. $\forall y \in M[f(y) = y \Rightarrow y = x]$ *(x is unique).*

Definition 2.3 *The closure $Cl(X)$ of a subset X of a metric space is the set $\{\lim_{i\to\infty} y_i : \forall i[y_i \in X]\}$.*

Definition 2.4 *Let (M,d) (M_1,d_1) (M_2,d_2) be metric spaces.*

1. *We define a metric d on the functions in $M_1 \to M_2$ as follows. For $f_1, f_2 \in M_1 \to M_2$*

$$d(f_1,f_2) = \sup\{d_2(f_1(x),f_2(x)) : x \in M_1\}.$$

2. *Put $\mathcal{P}_{co}(M)$ ($\mathcal{P}_{cl}(M)$) the set of compact (closed) and non empty subsets of M. We define a metric d_H on both $\mathcal{P}_{co}(M)$ and $\mathcal{P}_{cl}(M)$, called the Hausdorff distance, as follows. For every $X, Y \in \mathcal{P}_{co}(M)$ ($\in \mathcal{P}_{cl}(M)$)*

$$d_H(X,Y) =$$

$$\max\{\sup\{d(x,Y) : x \in X\}, \sup\{d(y,X) : y \in Y\}\}$$

where $d(x,Z) = \inf\{d(x,z) : z \in Z\}$ for every $Z \subset M$, $x \in M$.

Theorem 2.5 *Let (M,d), (M_1,d_1), (M_2,d_2) be complete (ultra)metric spaces. We have that $M_1 \to M_2$, $\mathcal{P}_{co}(M)$ and $\mathcal{P}_{cl}(M)$ (with the metrics defined above) are complete (ultra)metric spaces.*

In the sequel we sometimes suppress the definitions of metrics. We then assume that they are constructed in the standard ways as outlined above.

3 Syntax

Assume given a (possibly infinite) set of atomic actions B, with typical element b. Let *Proc*, with typical element P, be a set of procedure variables. These two basic sets are used in

Definition 3.1 *We define the set of statements $\mathcal{L}$, with typical element s, by the following grammar:*

$$s ::= b \mid P \mid s_1; s_2 \mid s_1 + s_2 \mid s_1 \parallel s_2 \mid [s_1].$$

A statement s is one of the following six forms:

- an elementary action b,
- a procedure variable P,
- the sequential composition $s_1; s_2$ of statements s_1 and s_2,
- the nondeterministic choice $s_1 + s_2$,
- the concurrent execution $s_1 \parallel s_2$, modeled by arbitrary interleaving,
- the atomic version $[s]$ of s, modeled by interpreting s as an elementary action.

Assume given a set of states Σ, with typical element σ. Let $Int = B \to \Sigma \to_{partial} \Sigma$ be the set of interpretations and let f be a typical element of *Int*. Given an elementary action b and an initial state σ, $f(b)(\sigma)$ (if it exists) is the state after the execution of b in state σ. The set of declarations *Decl* (with typical element d) has as elements functions from $Proc \to \mathcal{L}_g$, where $\mathcal{L}_g$ (the set of guarded statements) is defined in

Definition 3.2 *We define the set of guarded statements $\mathcal{L}_g$, with typical element g, by the following syntax:*

$$g ::= b \mid g; s \mid g_1 + g_2 \mid g_1 \parallel g_2 \mid [g_1].$$

Note that $\mathcal{L}_g \subset \mathcal{L}$. Intuitively, a statement s is guarded if all procedure variables are preceded by some statement. A program is a triple (f,d,s), where s is a statement, $d \in Decl$ is a declaration for the procedure variables in s and f is an interpretation of the atomic actions. Let *Prog* be the set of programs. In the sequel we sometimes suppress the declaration and interpretation parts of a program: instead of writing (f,d,s) we write just s.

4 Operational Semantics

The operational semantics for $\mathcal{L}$ is based on a transition relation in the style of [Plotkin 1981]. A transition relation describes the steps we can take during a computation. We use a special symbol E, which stands for termination. A step can change the state and the (rest of) the program we have to execute.

Definition 4.1 *Let*

$$\to \subseteq (Prog \times \Sigma \times (Prog \cup \{E\}) \times \Sigma)$$

be the smallest relation satisfying (writing $(s,\sigma) \to (s',\sigma')$ for $(s,\sigma,s',\sigma') \in \to$ and $(s,\sigma) \to (E,\sigma')$ for $(s,\sigma,E,\sigma') \in \to$ and writing $A \to A_1 | \ldots | A_n \Rightarrow B \to B_1 | \ldots | B_n$ for $A \to A_1 \Rightarrow B \to B_1 \wedge \ldots \wedge A \to A_n \Rightarrow B \to B_n$ where A, B are typical elements of $(Prog \cup \{E\}) \times \Sigma$)

- $(b,\sigma) \to (E, f(b)(\sigma))$ *if $f(b)(\sigma)$ exists*
- $(d(P),\sigma) \to (s,\sigma') | (E,\sigma') \Rightarrow (P,\sigma) \to (s,\sigma') | (E,\sigma')$
- $(s_1,\sigma_1) \to (s_2,\sigma_2) | (E,\sigma_2) \Rightarrow$

$$(s_1; s,\sigma_1) \to (s_2; s,\sigma_2) | (s,\sigma_2)$$
$$(s_1 \parallel s,\sigma_1) \to (s_2 \parallel s,\sigma_2) | (s,\sigma_2)$$
$$(s \parallel s_1,\sigma_1) \to (s \parallel s_2,\sigma_2) | (s,\sigma_2)$$
$$(s + s_1,\sigma_1) \to (s_2,\sigma_2) | (E,\sigma_2)$$
$$(s_1 + s,\sigma_1) \to (s_2,\sigma_2) | (E,\sigma_2)$$

- $(s,\sigma) \rightarrow^* (E,\sigma') \Rightarrow ([s],\sigma) \rightarrow (E,\sigma')$ *(writing* $\rightarrow^*$ *for the transitive closure of* $\rightarrow$*).*

The last rule takes several transitions together: in order to get a step from $([s],\sigma)$ we analyse sequences of steps from (s,σ). We have the following lemma:

Lemma 4.2 *For all* $s \in \mathcal{L}$ *and* $\sigma \in \Sigma$*, the set*

$$\{s' \in \mathcal{L} : \exists \sigma' \in \Sigma[(s,\sigma) \rightarrow (s',\sigma')]\}$$

is a finite set.

Please note that (for any $s \in \mathcal{L}$ and $\sigma \in \Sigma$) $\{(s',\sigma') \in \mathcal{L} \times \Sigma : (s,\sigma) \rightarrow (s',\sigma')\}$ is in general an infinite set.

We use the transition relation to give an operational semantics: we collect the sequence of states during a computation. Such a sequence can be finite or infinite. We also signal deadlock by a special symbol δ. Deadlock means that from a configuration (s,σ) no transition is possible. This can happen because *Int* contains partial functions. Let Σ^* (Σ^ω) denote the collection of all finite (infinite) words over Σ. Let x be a typical element of Σ^* and let y be a typical element of $\Sigma^\infty = \Sigma^* \cup \Sigma^\omega$. Let Σ^+ be Σ^* without the empty word and let $\Sigma_\delta^{st} = \Sigma^* \cdot \{\delta\} \cup \Sigma^+ \cup \Sigma^\omega$. Let z be a typical element of Σ_δ^{st}. Put $\mathbf{S} = \Sigma \rightarrow \mathcal{P}_{closed}(\Sigma_\delta^{st})$: the set of functions from Σ to the closed subsets of Σ_δ^{st}.
We turn Σ_δ^{st} into a complete metric space in

Definition 4.3 *We define a metric* d_{st} *on* Σ_δ^{st} *by putting* $d_{st}(z_1,z_2) = 2^{-N}$ *where* $N = \sup\{n : z_1[n] = z_2[n]\}$ *where for each* $z \in \Sigma_\delta^{st}$, $z[n]$ *is the prefix of length* n*, if this exists, and* $z[n] = z$*, otherwise.*

The operational semantics is given in

Definition 4.4 (Operational semantics $\mathcal{O}$) *Let* $\mathcal{O} : Prog \rightarrow \mathbf{S}$ *be the unique fixed point of the contraction* $\Delta : (Prog \rightarrow \mathbf{S}) \rightarrow (Prog \rightarrow \mathbf{S})$ *which is defined as follows:*

$$\begin{aligned}\Delta(F)(s) = \lambda\sigma.\ & \{\delta : (s,\sigma) \not\rightarrow\} \cup \\ & \{\sigma_1 : (s,\sigma) \rightarrow (E,\sigma_1)\} \cup \\ & \cup\{\sigma_1 \cdot F(s_1)(\sigma_1) : (s,\sigma) \rightarrow (s_1,\sigma_1)\}.\end{aligned}$$

5 Denotational semantics

In this section we define a denotational semantics for $\mathcal{L}$. We call a semantics denotational if it is compositionally defined and tackles recursion with the help of fixed points. With each operator in $\mathcal{L}$ we associate a semantic operator. The denotational semantics will be the fixed point of a higher-order operator. The denotational semantics will be based on domains which are metric spaces. These domains are defined with the help of domain equations. The equations can be solved with techniques described in [America and Rutten 1988] or [de Bakker and Zucker 1982].

In the construction of the domains for the denotational semantics we need an operator $\Sigma\Box$ which is defined in

Definition 5.1 *Let* (M,d) *be a metric space. We define a metric* $\tilde{d}$ *on* $\Sigma\Box M =^{def} \Sigma_\delta^{st} \cup \Sigma^+ \times M$ *by putting*

$$\tilde{d}(z_1,z_2) = d_{st}(z_1,z_2) \text{ if } z_1,z_2 \in \Sigma_\delta^{st}$$

$$\tilde{d}((z_1,m_1),(z_2,m_2)) =$$

$$\begin{cases} d_{st}(z_1,z_2) & \text{if } z_1,z_2 \in \Sigma^+, m_1,m_2 \in M, z_1 \neq z_2 \\ 2^{-length(z_1)}.d(m_1,m_2) & \text{if } z_1,z_2 \in \Sigma^+, m_1,m_2 \in M, z_1 = z_2 \end{cases}$$

$$\tilde{d}(z_1,(z_2,m)) =$$

$$\begin{cases} d_{st}(z_1,z_2) & \text{if } z_1 \in \Sigma_\delta^{st}, z_2 \in \Sigma^+, m \in M, z_1 \neq z_2 \\ 2^{-length(z_1)} & \text{if } z_1 \in \Sigma_\delta^{st}, z_2 \in \Sigma^+, m \in M, z_1 = z_2 \end{cases}$$

$$\tilde{d}((z_1,m),z_2) =$$

$$\begin{cases} d_{st}(z_1,z_2) & \text{if } z_2 \in \Sigma_\delta^{st}, z_1 \in \Sigma^+, m \in M, z_1 \neq z_2 \\ 2^{-length(z_1)} & \text{if } z_2 \in \Sigma_\delta^{st}, z_1 \in \Sigma^+, m \in M, z_1 = z_2 \end{cases}$$

We have that $(\Sigma\Box M,\tilde{d})$ is a complete ultrametric space if (M,d) is a complete ultrametric space. We briefly recall the notion of a (metric) domain equation. The general form of such an equation is $\mathbf{P} = F(\mathbf{P})$ or, more precisely, $(\mathbf{P},d) \cong F((\mathbf{P},d))$, where the mapping F maps metric spaces to metric spaces. Under certain conditions, we can find unique solutions (upto isometry) in the category of complete metric spaces. We have no room to discuss details. For our purposes, it is sufficient to know that $\mathbf{P} = \Sigma \rightarrow \mathcal{P}_{co}(\Sigma\Box\mathbf{P})$ has a complete ultrametric space as solution.

Definition 5.2 *Let* $\mathbf{P}$ *be the unique complete ultrametric space that satisfies*

$$\mathbf{P} = \Sigma \rightarrow \mathcal{P}_{co}(\Sigma\Box\mathbf{P})$$

Elements of $\mathbf{P}$ are called processes. Let p be a typical element of $\mathbf{P}$. Given an initial state σ, $p(\sigma)$ is a (compact) set. Elements of this set are either in Σ_δ^{st} or in $\Sigma^+ \times \mathbf{P}$. An element in Σ^+ (Σ^ω) can be seen as a (non) terminating computation, an element in $\Sigma^* \cdot \{\delta\}$ as a computation ending in deadlock. An element in $\Sigma^+ \times \mathbf{P}$ can be viewed as a terminating computation which has a resumption: after the computation (which is finite), it turns itself into another proces. It may be surprising that we use streams of states in our processes. This

is done for technical reasons: we then can use compact sets. If we take paths in our processes (to be made precise below) we have that the resulting sets are also compact. This is used in the equivalence proof of the correctness of the denotational semantics with respect to the operational semantics.

For each syntactic operator in $\mathcal{L}$ we define a semantic operator. The semantic operators corresponding with ;, + and $\|$ will be of type $\mathbf{P} \times \mathbf{P} \to \mathbf{P}$ and the semantic operator corresponding to $[\cdot]$ will be of type $\mathbf{P} \to \mathbf{P}$.

Definition 5.3 *The operators* $\otimes$, $\tilde{+}$, $\tilde{\|}$: $\mathbf{P} \times \mathbf{P} \to \mathbf{P}$ *and stream:* $\mathbf{P} \to \mathbf{P}$ *are defined as follows. Let*

$$p_1 \tilde{+} p_2 = \lambda\sigma.(p_1(\sigma) \cup p_2(\sigma))$$

and let $\otimes, \tilde{\|}$ *be the unique fixed points of the contractions* $\Phi_\otimes, \Phi_{\tilde{\|}} : (\mathbf{P} \times \mathbf{P} \to \mathbf{P}) \to (\mathbf{P} \times \mathbf{P} \to \mathbf{P})$ *that are defined as follows:*

$$\begin{aligned}\Phi_\otimes(F)(p_1,p_2) = \lambda\sigma.\{z : z \in p_1(\sigma) \wedge z \in \Sigma^\omega \cup \Sigma^*.\{\delta\}\} \cup \\ \{(z,p_2) : z \in p_1(\sigma) \wedge z \in \Sigma^+\} \cup \\ \{(z,F(p,p_2)) : (z,p) \in p_1(\sigma)\},\end{aligned}$$

$$\Phi_{\tilde{\|}}(F)(p_1,p_2) = \Phi_\otimes(F)(p_1,p_2) \tilde{+} \Phi_\otimes(F)(p_2,p_1)$$

and let stream: $\mathbf{P} \to \mathbf{P} \cap \mathbf{S}$ *be the unique fixed point of the contraction* $\Phi_{stream} : (\mathbf{P} \to \mathbf{P} \cap \mathbf{S}) \to (\mathbf{P} \to \mathbf{P} \cap \mathbf{S})$ *that is defined by*

$$\begin{aligned}\Phi_{stream}(F)(p) = \lambda\sigma.\{z \in \Sigma^{st}_\delta : z \in p(\sigma)\} \cup \\ \{x\sigma' z : (x\sigma', p') \in p(\sigma) \wedge z \in F(p')(\sigma')\}.\end{aligned}$$

In the sequel we use a left-merge operator:

Definition 5.4 *Define* $\tilde{\lfloor\!\lfloor} : \mathbf{P} \times \mathbf{P} \to \mathbf{P}$ *by* $\tilde{\lfloor\!\lfloor} = \Phi_\otimes(\tilde{\|})$.

We often write $+$, $\|$, $\lfloor\!\lfloor$ rather than $\tilde{+}$, $\tilde{\|}$, $\tilde{\lfloor\!\lfloor}$ if no confusion is possible. Now we define a denotational semantics for $\mathcal{L}$ in

Definition 5.5 *Let* $\mathcal{D} : Prog \to \mathbf{P}$ *be the unique fixed point of the contraction* $\Phi : (Prog \to \mathbf{P}) \to (Prog \to \mathbf{P})$ *which is defined inductively as follows:*

- $\Phi(F)(b) = \lambda\sigma.\begin{cases}\{f(b)(\sigma)\} & \text{if } f(b)(\sigma) \text{ exists} \\ \{\delta\} & \text{otherwise,}\end{cases}$
- $\Phi(F)(P) = \Phi(F)(d(P))$,
- $\Phi(F)(s_1;s_2) = \Phi(F)(s_1) \otimes F(s_2)$,
- $\Phi(F)(s_1 + s_2) = \Phi(F)(s_1) + \Phi(F)(s_2)$,
- $\Phi(F)(s_1 \| s_2) = \Phi(F)(s_1) \| \Phi(F)(s_2)$,
- $\Phi(F)(|s|) = stream(\Phi(F)(s))$.

6 Relation between the operational and denotational semantics

The operational semantics $\mathcal{O}$ delivers linear-time objects (for a given state σ, $\mathcal{O}(s)(\sigma) \subseteq \Sigma^{st}_\delta$) whereas the denotational semantics $\mathcal{D}$ delivers branching time objects in $\mathbf{P}$. We define an restriction operator *restr* which will link $\mathcal{O}$ and $\mathcal{D}$: given a process p and an initial state σ, it delivers certain 'paths' in the process p. A path will be an element of Σ^{st}_δ. In next definition we use the operator *last* which takes the last element of a word in Σ^+.

Definition 6.1 *Let* $restr : \mathbf{P} \to \mathbf{S}$ *be the unique fixed point of the contraction* $\Gamma : (\mathbf{P} \to \mathbf{S}) \to (\mathbf{P} \to \mathbf{S})$ *which is given by* $\Gamma(F)(p)(\sigma) = \{\delta\}$ *if* $p(\sigma) \subset \Sigma^\omega \cup \Sigma^* \cdot \{\delta\}$ *and otherwise it equals*

$$\begin{aligned}Cl(\{last(x) : x \in \Sigma^+ \wedge x \in p(\sigma)\} \cup \\ \cup\{last(x) \cdot F(p')(last(x)) : \\ (x,p') \in \Sigma^+ \times \mathbf{P} \wedge (x,p') \in p(\sigma)\}).\end{aligned}$$

We have

Theorem 6.2 $\mathcal{O} = restr \circ \mathcal{D}$

In order to prove this theorem, we will define an intermediate semantics $\mathcal{I}$. It is called intermediate because it serves as an intermediate semantics between $\mathcal{O}$ and $\mathcal{D}$: it is defined with the help of a transition system (like the operational semantics) and it delivers tree-like objects (like the denotational semantics). We will prove the theorem in two steps. First we show that $\mathcal{I} = \mathcal{D}$ and secondly we show that $\mathcal{O} = restr \circ \mathcal{I}$.

We prove $\mathcal{I} = \mathcal{D}$ by showing that $\mathcal{D}$ is a fixed point of the defining contraction of $\mathcal{I}$, and hence, by Banach's theorem, we have that $\mathcal{I} = \mathcal{D}$. The proof is an extension of the ideas present in [Kok and Rutten 1988] and [de Bakker and Meyer 1988].

We give the transition system for the intermediate semantics $\mathcal{I}$ in

Definition 6.3 *Let*

$$\rightarrow \subseteq Prog \times \Sigma \times \Sigma^\infty \times (Prog \cup \{E\}) \times (\Sigma \cup \{\delta\})$$

be the smallest relation satisfying (writing $(s,\sigma) \xrightarrow{y} (s',\sigma')$ *for* $(s,\sigma,y,s',\sigma') \in \rightarrow$, $(s,\sigma) \xrightarrow{y} (E,\sigma')$ *for* $(s,\sigma,y,E,\sigma') \in \rightarrow$, $(s,\sigma) \xrightarrow{y} (s',\delta)$ *for* $(s,\sigma,y,s',\delta) \in \rightarrow$ *and* $(s,\sigma) \xrightarrow{y} (E,\delta)$ *for* $(s,\sigma,y,E,\delta) \in \rightarrow$ *and writing* $A \to A_1|\ldots|A_n \Rightarrow B \to B_1|\ldots|B_n$ *for* $A \to A_1 \Rightarrow B \to B_1 \wedge \ldots \wedge A \to A_n \Rightarrow B \to B_n$ *where* A, B *are typical elements of* $(Prog \cup \{E\}) \times (\Sigma \cup \{\delta\})$*)*

- $(b,\sigma) \xrightarrow{\epsilon} (E, f(b)(\sigma))$ *if* $f(b)(\sigma)$ *exists*
- $(b,\sigma) \xrightarrow{\epsilon} (E,\delta)$ *if* $f(b)(\sigma)$ *does not exist*

- $(d(P),\sigma) \xrightarrow{y} (s,\sigma')|(E,\sigma')|(s,\delta)|(E,\delta) \Rightarrow$

  $(P,\sigma) \xrightarrow{y} (s,\sigma')|(E,\sigma')|(s,\delta)|(E,\delta)$
- $(s_1,\sigma_1) \xrightarrow{y} (s_2,\sigma_2)|(E,\sigma_2)|(s_2,\delta)|(E,\delta) \Rightarrow$

  $(s_1;s,\sigma_1) \xrightarrow{y} (s_2;s,\sigma_2)|(s,\sigma_2)|(s_2;s,\delta)|(s,\delta)$
  $(s_1 \parallel s,\sigma_1) \xrightarrow{y} (s_2 \parallel s,\sigma_2)|(s,\sigma_2)|(s_2 \parallel s,\delta)|(s,\delta)$
  $(s \parallel s_1,\sigma_1) \xrightarrow{y} (s \parallel s_2,\sigma_2)|(s,\sigma_2)(s \parallel s_2,\delta)|(s,\delta)$
  $(s+s_1,\sigma_1) \xrightarrow{y} (s_2,\sigma_2)|(E,\sigma_2)|(s_2,\delta)|(E,\delta)$
  $(s_1+s,\sigma_1) \xrightarrow{y} (s_2,\sigma_2)|(E,\sigma_2)|(s_2,\delta)|(E,\delta)$
- *If* $(s_1,\sigma_1) \xrightarrow{y_1} (s_2,\sigma_2) \xrightarrow{y_2} \cdots (s_n,\sigma_n)$ *then*

  $(s_n,\sigma_n) \xrightarrow{y_n} (E,\sigma)|(E,\delta)|(s,\delta) \Rightarrow$

  $([s_1],\sigma_1) \xrightarrow{y_1\sigma_2\cdots\sigma_n y_n} (E,\sigma)|(E,\delta)|(s,\delta)$
- $(s_1,\sigma_1) \xrightarrow{y_1} (s_2,\sigma_2) \xrightarrow{y_2} \cdots (s_n,\sigma_n) \xrightarrow{y_n} \cdots \Rightarrow$

  $([s_1],\sigma_1) \xrightarrow{y_1\sigma_2\cdots\sigma_n y_n\cdots} (E,\delta)$

Note that we have defined two transition relations: one in definition 4.1 and the other in definition 6.3. The second relation is always written with a superscript. The following lemma holds

Lemma 6.4

$$\exists y[(s,\sigma) \xrightarrow{y} (s',\sigma')] \Leftrightarrow (s,\sigma) \rightarrow (s',\sigma')$$
$$\exists y[(s,\sigma) \xrightarrow{y} (E,\sigma')] \Leftrightarrow (s,\sigma) \rightarrow (E,\sigma')$$

It follows that

$$(s,\sigma) \not\rightarrow \Leftrightarrow \neg\exists y,s',\sigma'[(s,\sigma) \xrightarrow{y} (s',\sigma') \vee (s,\sigma) \xrightarrow{y} (E,\sigma')]$$

Next we give the intermediate semantics:

Definition 6.5 *Let* $I : Prog \rightarrow \mathbf{P}$ *be the unique fixed point of the contraction* $\Psi : (Prog \rightarrow \mathbf{P}) \rightarrow (Prog \rightarrow \mathbf{P})$ *which is defined as follows.*

$$\begin{aligned}\Psi(F)(s) = \lambda\sigma.\ & \{y\sigma' : (s,\sigma) \xrightarrow{y} (E,\sigma')\} \cup \\ & \{y\delta : (s,\sigma) \xrightarrow{y} (E,\delta)\} \cup \\ & \{y\delta : (s,\sigma) \xrightarrow{y} (s',\delta)\} \cup \\ & \{(y\sigma', F(s')) : (s,\sigma) \xrightarrow{y} (s',\sigma')\}\end{aligned}$$

We provide a lemma with properties of the defining contraction Ψ of I:

Lemma 6.6

1. $\Psi(\mathcal{D})(b) = \mathcal{D}(b)$
2. $\Psi(\mathcal{D})(P) = \Psi(\mathcal{D})(d(P))$
3. $\Psi(\mathcal{D})(s_1;s_2) = \Psi(\mathcal{D})(s_1) \otimes \mathcal{D}(s_2)$
4. $\Psi(\mathcal{D})(s_1+s_2) = \Psi(\mathcal{D})(s_1) + \Psi(\mathcal{D})(s_2)$
5. $\Psi(\mathcal{D})(s_1 \parallel s_2) =$
   $\Psi(\mathcal{D})(s_1) \between \mathcal{D}(s_2) + \Psi(\mathcal{D})(s_2) \between \mathcal{D}(s_1)$
6. $\Psi(\mathcal{D})([s]) \in \mathbf{P} \cap \mathbf{S}$
7. $\Psi(\mathcal{D})([s]) =$
   $\lambda\sigma.\ \{y\sigma' : (s,\sigma) \xrightarrow{y} (E,\sigma')\} \cup$
   $\{y\delta : (s,\sigma) \xrightarrow{y} (s',\delta)\} \cup$
   $\{y\delta : (s,\sigma) \xrightarrow{y} (E,\delta)\} \cup$
   $\cup\{y\sigma' \cdot \Psi(\mathcal{D})([s'])(\sigma') : (s,\sigma) \xrightarrow{y} (s',\sigma')\}$

Lemma 6.7 $\Psi(\mathcal{D}) = \mathcal{D}$

Proof We show that for all $s \in \mathcal{L}$ $d(\Psi(\mathcal{D})(s),\mathcal{D}(s)) \leq \frac{1}{2}.d(\Psi(\mathcal{D}),\mathcal{D})$. This implies that $d(\Psi(\mathcal{D}),\mathcal{D}) \leq \frac{1}{2}.d(\Psi(\mathcal{D}),\mathcal{D})$, i.e. $d(\Psi(\mathcal{D}),\mathcal{D}) = 0$, i.e. $\Psi(\mathcal{D}) = \mathcal{D}$.

We first prove it for $g \in \mathcal{L}_g$. We use structural induction on the elements of $\mathcal{L}_g$. We give only the cases $g;s$, $[g]$:

$(g;s)$ $d(\Psi(\mathcal{D})(g;s),\mathcal{D}(g;s)) =$

$d(\Psi(\mathcal{D})(g) \otimes \mathcal{D}(s),\mathcal{D}(g) \otimes \mathcal{D}(s)) \leq$

$\max\{d(\Psi(\mathcal{D})(g),\mathcal{D}(g)),d(\mathcal{D}(s),\mathcal{D}(s))\} =$

$d(\Psi(\mathcal{D})(g),\mathcal{D}(g)) \leq$ (induction)

$\frac{1}{2}.d(\Psi(\mathcal{D}),\mathcal{D})$.

$([g])$ $d(\Psi(\mathcal{D})([g]),\mathcal{D}([g])) \leq$

$\max\{\ d(\Psi(\mathcal{D})([g]),stream(\Psi(\mathcal{D})(g))),$
$d(stream(\Psi(\mathcal{D})(g)),\mathcal{D}([g]))\}.$

We show that

1. $d(\Psi(\mathcal{D})([g]),stream(\Psi(\mathcal{D})(g))) \leq \frac{1}{2}.d(\Psi(\mathcal{D}),\mathcal{D})$
2. $d(stream(\Psi(\mathcal{D})(g)),\mathcal{D}([g])) \leq \frac{1}{2}.d(\Psi(\mathcal{D}),\mathcal{D})$

1. $\Psi(\mathcal{D})([g]) =$

   $\lambda\sigma.\ \{y\sigma' : (g,\sigma) \xrightarrow{y} (E,\sigma')\} \cup$
   $\{y\delta : (g,\sigma) \xrightarrow{y} (s',\delta)\} \cup$
   $\{y\delta : (g,\sigma) \xrightarrow{y} (E,\delta)\} \cup$
   $\cup\{y\sigma' \cdot \Psi(\mathcal{D})([s'])(\sigma') : (g,\sigma) \xrightarrow{y} (s',\sigma')\}.$

   On the other hand, we have

   $stream(\Psi(\mathcal{D})(g)) =$

   $stream(\lambda\sigma.\ \{y\sigma' : (g,\sigma) \xrightarrow{y} (E,\sigma')\} \cup$
   $\{y\delta : (g,\sigma) \xrightarrow{y} (s',\delta)\} \cup$
   $\{y\delta : (g,\sigma) \xrightarrow{y} (E,\delta)\} \cup$
   $\{(y\sigma',\mathcal{D}(s')) : (g,\sigma) \xrightarrow{y} (s',\sigma')\}) =$

   $\lambda\sigma.\ \{y\sigma' : (g,\sigma) \xrightarrow{y} (E,\sigma')\} \cup$
   $\{y\delta : (g,\sigma) \xrightarrow{y} (s',\delta)\} \cup$
   $\{y\delta : (g,\sigma) \xrightarrow{y} (E,\delta)\} \cup$
   $\cup\{y\sigma' \cdot stream(\mathcal{D})(s')(\sigma') :$
   $(g,\sigma) \xrightarrow{y} (s',\sigma')\} =$

$$\lambda\sigma.\ \{y\sigma' : (g,\sigma) \xrightarrow{y} (E,\sigma')\} \cup \{y\delta : (g,\sigma) \xrightarrow{y} (s',\delta)\} \cup \{y\delta : (g,\sigma) \xrightarrow{y} (E,\delta)\} \cup \bigcup\{y\sigma' \cdot \mathcal{D}([s'])) : (g,\sigma) \xrightarrow{y} (s',\sigma')\}$$

so

$$d(\Psi(\mathcal{D})([g]), stream(\Psi(\mathcal{D})(g))) \leq \tfrac{1}{2}.d(\Psi(\mathcal{D}),\mathcal{D})$$

since $y\sigma'$ is not equal to the empty word (hence the factor $\frac{1}{2}$). Note that the last step does not use the induction hypothesis.

2. $d(stream(\Psi(\mathcal{D})(g)), \mathcal{D}([g])) =$

$$d(stream(\Psi(\mathcal{D})(g)), stream(\mathcal{D}(g))) \leq$$

$$d(\Psi(\mathcal{D})(g), \mathcal{D}(g)) \leq$$

$$\tfrac{1}{2}.d(\Psi(\mathcal{D}),\mathcal{D}).$$

Secondly, we extend $\mathcal{L}_g$ to $\mathcal{L}$. We use structural induction on the elements of $\mathcal{L}$. All cases are the same as for $\mathcal{L}_g$, except for P (which is not present in the guarded case).

(P) By lemma 6.6 we have $\Psi(\mathcal{D})(P) = \Psi(\mathcal{D})(d(P))$ and by the definition of $\mathcal{D}$ we have $\mathcal{D}(P) = \mathcal{D}(d(P))$ Hence $d(\Psi(\mathcal{D})(P), \mathcal{D}(P)) = d(\Psi(\mathcal{D})(d(P)), \mathcal{D}(d(P)))$ $\leq \frac{1}{2}.d(\Psi(\mathcal{D}),\mathcal{D})$ because $d(P)$ is a guarded statement.

By Banach's fixed point theorem we have the following

Corollary 6.8 $I = \mathcal{D}$

Lemma 6.9 $\mathcal{O} = restr \circ I$

Proof We show that $\Delta(restr \circ I) = restr \circ I$ where Δ is the defining contraction of $\mathcal{O}$. By the definition of Δ, $\Delta(restr \circ I)(s)(\sigma) = \{\delta\}$ if $(s,\sigma) \not\rightarrow$. Because $(s,\sigma) \not\rightarrow$ implies $\neg\exists y, s', \sigma'[(s,\sigma) \xrightarrow{y} (s',\sigma') \vee (s,\sigma) \xrightarrow{y} (E,\sigma')]$ we have by definition of I that $I(s)(\sigma) \subset \Sigma^\omega \cup \Sigma^* \cdot \{\delta\}$ i.e. $(restr \circ I)(s)(\sigma) = \{\delta\}$. Now assume that there are transitions possible from (s,σ):

$$\Delta(restr \circ I)(s)(\sigma) =$$

$$\{\sigma' : (s,\sigma) \to (E,\sigma')\} \cup \bigcup\{\sigma' \cdot (restr \circ I)(s')(\sigma') : (s,\sigma) \to (s',\sigma')\} =$$

$$Cl(\{\sigma' : (s,\sigma) \to (E,\sigma')\} \cup \bigcup\{\sigma' \cdot (restr \circ I)(s')(\sigma') : (s,\sigma) \to (s',\sigma')\}) =$$

$$Cl(\{last(x \cdot \sigma') : (s,\sigma) \xrightarrow{x} (E,\sigma') \wedge x \in \Sigma^*\} \cup \bigcup\{last(x \cdot \sigma') \cdot (restr \circ I)(s')(last(x \cdot \sigma') : (s,\sigma) \xrightarrow{x} (s',\sigma')\}) =$$

$$(restr \circ I)(s)(\sigma).$$

7 Concurrent Prolog

In this section we apply the framework of the previous sections. We choose a set of elementary actions, a set of procedure variables, a set of states and an interpretation function in such a way that we obtain a denotational and an operational semantics for Concurrent Prolog. A Concurrent Prolog program is 'translated' to an element of *Prog*.

We first introduce the language Concurrent Prolog (CP for short) in an informal way. The reader not familiair with CP should consult [Shapiro 1983], the paper which introduces the concepts of CP. The paper [Saraswat 1987] signals some problems and proposes a family of languages called CP. We can incorparate the features described in that paper in our model. In the present paper we do not give a formal definition of the extended unification. We can take over the formalization proposed in [Saraswat 1987] for the input-only functor. In order to do this properly we should place a restriction on the places where read-only (input-only in terms of [Saraswat 1987]) annotations are allowed. Also our interpretation of the commit operator comes from that paper. For the details we refer to [Saraswat 1987]. The rest of our paper can be read without knowledge of the exact details.

Let a be a typical element of the set of atoms *Atom*. Atoms are built up in the usual way from constants, variables, functors and predicate symbols. In CP there is a special functor ? of arity one which is called the read-only functor. An extension of normal unification is defined in order to cope with this read-only functor. In this paper we call this extension $mgu_?$. It is a partial function on $Atom \times Atom$. If it is defined it delivers a substitution. A CP program is a finite set of elements (called clauses) of the following form:

$$a \leftarrow a_1 \wedge \ldots \wedge a_n | a_{n+1} \wedge \ldots \wedge a_m.$$

Both n, m can be 0. The bar | is called the commit operator, $\bar{a}$ the head, $a_1 \wedge \ldots \wedge a_n$ the guard and $a_{n+1} \wedge \ldots \wedge a_m$ the body of the clause. If $n = 0$ we have an empty guard and if $n = m$ we have an empty body. Let *Clause* be the set of clauses and let c be a typical element of *Clause*. Besides a finite set of clauses, we also have a goal which is of the form $\bar{a}_1 \wedge \ldots \wedge \bar{a}_k$. If $k = 0$ we say that the goal is empty. The (interleaved) execution of a CP program goes as follows. We execute the goal given the identity substitution. The execution of a goal given a (current) substitution means that we try to resolve all the atoms in the goal until the goal is empty. If the goal is empty we return a substitution. In order to resolve an atom we unify it with the head of a clause (taking the current substitution into account) and we try to execute the guard (given the 'new' substitution resulting

from the unification). The unification and the execution of the guard do not yet influence the current substitution. Only after the guard becomes empty we commit: we do not consider alternatives for this clause anymore and we replace the atom by the body of the clause and update the current substitution. The execution model described here is an interleaving model. We do not consider here a parallel model where we have truly parallel processes. In such a model we also would have to check if a substitution delivered by the execution of a guard matches with the current substitution.

We introduce disjoint sets of variables. This is done for technical reasons. During the process described above we replace atoms by bodies of clauses. In order to avoid clashes of variables, every time we rewrite an atom we 'replace' the variables in the clause by new ones. This is intuitively correct because clauses are assumed to be universally quantified. Therefore we partition the set of variables Var into infinite disjunct subsets Var_α, where α ranges over $\mathbf{N}^*$, the set of finite words of integers. Assume injections $\alpha : Var_\epsilon \to Var_\alpha$ (and their natural extensions to elements of $Atom$). Now we choose our basic sets: take Σ the set of substitutions, $B = Atom \times Atom$ and $Proc = Atom \times \mathbf{N}^*$ (where $\mathbf{N}^*$ is the set of finite words of integers). A pair (a, α) in $Proc$ specifies that we have to rewrite the atom a with a clause of the program in which the variables are taken from Var_α. Take $f(a_1, a_2)(\sigma) = mgu_?(a_1, \sigma(a_2)) \circ \sigma$ if $mgu_?(a_1, \sigma(a_2))$ is defined and is undefined otherwise. Fix a CP program and a goal. We assume that all variables in the program and in the goal are taken from Var_ϵ. We define a function $stm : Clause \times Proc \to \mathcal{L}_g$ by

$$stm(\bar{a} \leftarrow a_1 \wedge \ldots \wedge a_n | a_{n+1} \wedge \ldots \wedge a_m, (a, \alpha)) =$$

$$[(\alpha(\bar{a}), a); (\alpha(a_1), \alpha \cdot 1) \parallel \cdots \parallel (\alpha(a_n), \alpha \cdot n)];$$

$$(\alpha(a_{n+1}), \alpha \cdot n + 1) \parallel \cdots \parallel (\alpha(a_m), \alpha \cdot m).$$

Suppose the set of clauses is $\{c_1, \ldots, c_k\}$. Define

$$d(P) = stm(c_1, P) + \cdots + stm(c_k, P).$$

Assume the goal is $\bar{a}_1 \wedge \ldots \wedge \bar{a}_k$. Take

$$s = (\bar{a}_1, 1) \parallel \cdots \parallel (\bar{a}_k, k).$$

Some explanation is at its place. Execution of the goal consists in parallel execution of the k procedure variables $(\bar{a}_1, 1), \ldots, (\bar{a}_k, k)$. When we call stm on a clause c and a pair (a, α) it considers what has to be done in order to rewrite atom a with clause c in which we have to take the variables from Var_α. Suppose $c = \bar{a} \leftarrow a_1 \wedge \ldots \wedge a_n | a_{n+1} \wedge \ldots \wedge a_m$. First we unify a with $\bar{a}$ (the head of clause c). Because we have to take variables from Var_α we rename the variables in $\bar{a}$ with the operator α: this results in the pair $(\alpha(\bar{a}), a)$. After this unification, we have to execute the guard of the clause c, i.e $a_1 \wedge \ldots \wedge a_n$. We can execute all the atoms (in which the variables are renamed by α) in parallel. In order to avoid clashes of variables, we specify that if $\alpha(a_i)$ is rewritten by a clause, variables in that clause are to be taken from $Var_{\alpha \cdot i}$. The resolving of the guard and the unification is not (yet) allowed to influence other computations. This is modeled by considering them to be an elementary action by placing $[\cdot]$ around the unification and the guard. After the execution of the guard, we continue with the execution of the body: again with the renaming and the specification of sets of variables.

This translation induces an operational- and denotational semantics for Concurrent Prolog: We combine the translation to $\mathcal{L}$ with the operational- and denotational semantics for $\mathcal{L}$. Also the equivalence (an operator linking the two semantics for Concurrent Prolog) is induced by the translation: we already have the restriction operator *restr* that relates the two semantics for $\mathcal{L}$. This method of uniform abstraction gives in our opinion more insight into the semantic models than a direct definition would give. A direct definition yields a transition system in the style of [Saraswat 1987] and a denotational semantics as in [Kok 1988]. The proof of the equivalence between two such semantic definitions would be more difficult to understand (due to the interpretation of the abstract sets).

8 Acknowledgements

We acknowledge fruitful discussions on our work in the Amsterdam concurrency group, including Frank de Boer, Arie de Bruin, John-Jules Meijer, Jan Rutten and Erik de Vink. We thank Erik de Vink and Katiuscia Palamidessi for the comments and suggestions made during the work.

9 References

[America and Rutten 1988] P. America, J.J.M.M. Rutten, *Solving reflexive domain equations in a category of complete metric spaces*, Proc. of the Third Workshop on Mathematical Foundations of Programming Language Semantics, Lecture Notes in Computer Science, Vol. 298, Springer (1988) 254-288.

[Apt 1987] K.R. Apt, *Introduction to logic programming*, Report CS-R8741, Centre for Mathematics and Computer Science, Amsterdam (1987), to appear as a chapter in Handbook of Theoretical Computer Science, North-Holland.

[Apt and van Emden 1982] K.R. Apt, M.H. van Emden, *Contributions to the theory of logic programming*, JACM Vol. 29, No. 3, July 1982, pp. 841-862.

[Apt and Plotkin 1986] K.R. Apt, G. Plotkin, *Countable nondeterminism and random assignment*, JACM Vol. 33, No. 4, October 1986, pp. 724-767.

[Beckman 1986] L. Beckman, *Towards a Formal Semantics for Concurrent Logic Programming Languages*, Proc. of the Third International Conference on Logic Programming, Lecture Notes in Computer Science, Vol. 225, Springer (1986) 335-349.

[de Bakker and Zucker 1982] J.W. de Bakker, J.I. Zucker, *Processes and the denotational semantics of concurrency*, Inform. and Control 54 (1982) 70-120.

[Bakker and Meyer 1987] J.W. de Bakker, J.-J.Ch. Meyer, *Metric semantics for concurrency* , Report Free University, Dept. of Computer Science, IR-139 (1987). to appear in BIT.

[de Bakker 1988] J.W. de Bakker, *Comparative semantics for flow of control in logic programming without logic*, Report CS-R88.., Centre for Mathematics and Computer Science, Amsterdam (1988) to appear.

[Engelking 1977] R. Engelking, *General topology*, Polish Scientific Publishers 1977.

[Falaschi and Levi 1988] M. Falaschi, G. Levi, *Operational and fixpoint semantics of a class of committed-choice languages*, Techn. Report, Dipartimento di Informatica, Universita di Pisa, Pisa (1988).

[Gerth et al. 1988] R. Gerth, M. Codish, Y. Lichtenstein, E. Shapiro, *Fully abstract denotational semantics for Concurrent Prolog*, Proc. Logic In Computer Science, (1988) 320-335.

[Kok 1988] J.N. Kok, *A compositional semantics for Concurrent Prolog*, Proc. Symp. on Theoretical Aspects Computer Science (R. Cori, M. Wirsing eds.), Lecture Notes in Computer Science Vol. 294, Springer (1988) 373-388.

[Kok and Rutten 1988] J.N. Kok, J.J.M.M. Rutten, *Contractions in comparing concurrency semantics*, Proc. International Colloquium Automata, Languages and Programming (T. Lepistö, A. Salomaa eds.), Lecture Notes in Computer Science Vol. 317, Springer (1988) 317-332.

[Levi 1988] G. Levi, *A new declarative semantics of Flat Guarded Horn Clauses*, Techn. Report, ICOT, Tokyo (1988).

[Levi and Palamidessi 1985] G. Levi, C. Palamidessi, *The declarative semantics of logical read-only variables*, Proc. Symp. on Logic Programming, IEEE Comp. Society Press (1985) 128-137.

[Levi and Palamidessi 1987] G. Levi, C. Palamidessi, *An approach to the declarative semantics of synchronization in logic languages*, Proc. 4th Int. Conference on Logic Programming, Melbourne, (1987) 877-893.

[Lloyd 1987] J.W. Lloyd, *Foundations of Logic Programming*, Springer (1984), (Second edition 1987).

[Plotkin 1981] G.D. Plotkin, *A structural approach to operational semantics*, Report DAIMI FN-19, Comp. Sci. Dept., Aarhus Univ. 1981.

[Saraswat 1987] V.A. Saraswat, *The concurrent logic programming language CP: definition and operational semantics*, in: Conference Record of the Fourteenth Annual ACM Symposium on Principles of Programming Languages, Munich, West Germany, January 21-23, 1987, pp. 49-62.

[Shapiro 1983] E.Y. Shapiro, *A subset of Concurrent Prolog and its interpreter*, Tech. Report TR-003, ICOT, Tokyo (1983).

[Shapiro 1987] E.Y. Shapiro, *Concurrent Prolog, a progress report*, in Fundamentals of Artificial Intelligence (W. Bibel, Ph. Jorrand, eds.), Lecture Notes in Computer Science, Vol 232, Springer (1987).

[Vink 1988] E. de Vink, *Equivalence of an Operational and a Denotational Semantics for a Prolog-like Language with Cut*, Report IRR-151, Free University, Amsterdam (1988).