Towards a Concurrent Semantics based Analysis of CC and CLP

U. Montanari* F. Rossi*
F. Bueno** M. García de la Banda** M. Hermenegildo**

*{ugo,rossi}@di.unipi.it
Universitá di Pisa
**{bueno,maria,herme}@fi.upm.es
Universidad Politécnica de Madrid (UPM)

1 Introduction

We present in an informal way some preliminary results on the investigation of efficient compile-time techniques for Constraint Logic [JL87] and Concurrent Constraint [Sar89] Programming. These techniques are viewed as source-to-source program transformations between the two programming paradigms and are based on a concurrent semantics of CC programs [MR91].

Previous work [BH92] showed that it is possible to perform program transformations from Prolog to AKL[1] [JH91], allowing the latter to fully exploit the Independent And-Parallelism (IAP) [HR93] present in Prolog programs. However, when extending the transformation techniques to the CLP paradigm [JL87, Col90, VanH89], some issues have to be initially solved. First, the notion of independence has to be extended [GHM93]. Second, compile-time tools based on the extended notions have to be developed in order to capture the independence of goals, allowing such transformation. For this purpose an analysis of the programs turns out to be needed.

Our analysis will be based on a semantics [MR91] which, although originally intended for CC programming, can be also applied to CLP, if suitably extended [BGHMR94]. Such semantics allows us to capture the dependencies present in a CLP program at a finer level of granularity than ever proposed to date in the literature. This provides the knowledge for performing a transformation of the program which will force an execution-time scheduling of processes which preserves those dependencies. When the transformed program is run in a concurrent environment, parallel execution of concurrent processes will be exploited, except for the cases where an explicit ordering has been annotated at compile-time based on the dependencies identified.

The same semantics can also be used to identify dependencies in CC programs. Based on such dependencies, an analysis of parallel and sequential threads in the concurrent computation can be performed, establishing the basis for a transformation of CC programs into parallel CLP programs (with explicit dynamic scheduling). A similar approach (although not based on program trans-

[1] AKL is a CC language based on the Extended Andorra Model, which is able to exploit the determinate-goals-first principle as well as various kinds of parallelism.

formation) has recently been proposed in [KS92], in which a static analysis of concurrent languages is proposed based on an algebraic construction of execution trees from which dependencies are identified.

The needed extension of the semantics (for dealing with CLP instead of CC programs) is non-trivial [BGHMR94]. In fact, it consists in capturing the *atomic* (instead of the *eventual*) interpretation of the tell operation: constraints are added only if they are consistent with the current store. This implies the need of having the possibility of knowing immediately if a set of constraints is consistent or not. Thus it may seem that the semantics construction would have to go back to the usual notion of a constraint system as a black box which can answer yes/no questions in one step (which is what is most generally used in all the semantics other than [MR91]). However, this is not really true. In fact, the semantic structure still shows all the atomic entailment steps of the underlying constraint system, thus allowing to derive the correct dependencies among agents.

The paper is organized as follows. Section 2 hints at the new problems arising when trying to understand the concept of goal independence in CLP programs. Then, Section 3 describes the concurrent semantics for CC, both in its eventual and in its atomic version, while Section 4 hints at its modification in order to apply it to the CLP parallelization. Section 5 then describes a meta-interpreter which creates the semantic structure for each CC program, and visualizes it, Section 6 describes how to transform a CLP program into its parallel version, and Section 7 describes the opposite transformation (from CC programs to CLP programs). For reasons of space we assume the reader to be familiar with the syntax and the semantics of both CLP [JL87] and CC [Sar89] programs.

2 Independence in CLP

The general, intuitive notion of independence between goals is that the goals' executions do not interfere with each other, and do not change in any "observable" way. Observables include the solutions and/or the time that it takes to compute them.

Previous work in the context of traditional Logic Programming languages [Con83, DeG84, HR93] has concentrated on defining independence in terms of preservation of search space, and such preservation has then been achieved by ensuring that either the goals do not share variables (*strict independence*) or if they share variables, that they do not "compete" for their bindings (*non-strict independence*).

Recently, the concept of independence has been extended to CLP [GHM93]. It has been shown that search space preservation is no longer sufficient for ensuring the efficiency of several optimizations when arbitrary CLP languages are taken into account. The reason is that while the number of reduction steps will certainly be constant if the search space is preserved, the cost of each step will not: modifying the order in which a sequence of primitive constraints is added to the store may have a critical influence on the time spent by the constraint solver algorithm in obtaining the answer, even if the resulting constraint is consistent (in fact, this issue is the core of the reordering application described in [MS92]). This implies that optimizations which vary the intended execution order established by the user, such as parallel or concurrent execution, must also consider an orthogonal issue – *independence of constraint solving* – which characterizes the properties of the constraint solver behavior when changing the order in which primitive constraints are considered.

3 A Concurrent Semantics for CC and CLP

Usually the semantics of CC programs [Sar89] is given operationally, following the SOS-style operational semantics, and thus suffering from the typical pathologies of an interleaving semantics. On the other hand, the concurrent semantics approach introduced in [MR91] presents a non-monolithic model of the shared store and of its communication with the agents, in which the behavior of the store and that of the agents can be uniformly expressed by context-dependent rewrite rules (i.e. rules which have a left hand side, a right hand side and a context), each of them being applicable if both its left hand side and its context are present in the current state of the computation. An application removes the left hand side and adds the right hand side. In particular, the context is crucial in faithfully representing asked constraints, which are checked for presence but not affected by the computation.

From such rules a semantics structure is then obtained. Such structure is called a contextual net [MR93] and it is constructed by starting from the initial agent and applying all rules in all possible ways. A contextual net is just an acyclic Petri net where the presence of context conditions, besides pre- and post-conditions, is allowed. In a net obtained from a CC program, transitions are labelled by the rule applied for them.

Three relations can be defined on the items (conditions and events) of the obtained net: two items are *concurrent* if they represent objects which may appear together in a computation state, they are *mutually exclusive* if they represent objects which can not appear in the same computation, and they are *dependent* if they represent objects which may appear in the same computation but in different computation steps.

For each computation of the CC program, the net provides a partial order expressing the dependency pattern among the events of the computation. As a result, all such computations are represented in a unique structure, where it is possible to see the maximal degree of both concurrency (via the concurrency relation) and indeterminism (via the mutual exclusion relation) available both at the program level and at the underlying constraint system.

Nevertheless, such semantics is not able to handle failure, in the sense of detecting inconsistencies generated by tell operations, since constraints are added without any consistency check (i.e., the "eventual" interpretation of the tell operation is modelled). We extended such semantics to include the case of failure [BGHMR94]. We showed that the new semantics can be obtained from the old one either by pruning some parts of the semantic structure, or by not generating them at all. On one hand, the semantic structure can be built up by first generating the net as before, and then propagating the failure information through the net by introducing a notion of *mutual inconsistency* between items. The inconsistent items are then pruned out. On the other hand, the net can be generated from scratch with a new computation rule for the semantics which takes mutual inconsistency into account.

The mutual inconsistency relation extends the mutual exclusion relation, in the sense of capturing more objects which are not allowed to be present in the same computation. In fact, in the original semantics, if two objects were mutually exclusive, they could not be present in the same deterministic computation, even at different computation steps, because they belonged to two different nondeterministic (in the sense of "don't-care" nondeterminism, or indeterministic) branches of the program execution. Now, two items exclude one another also when they are mutually inconsistent, that is, when they represent (or generate) objects which are inconsistent.

When introducing an explicit representation for failure in the original semantics, what is achieved in fact is a faithful model for capturing backtracking. In other words, failing branches in a computation are also captured, allowing us to exchange nondeterminism for indeterminism. In the extended semantics, two different branches will be mutually inconsistent if they (together) lead to failure, otherwise, if they are mutually exclusive they will represent two different deterministic computations yielding distinct solutions, i.e. a nondeterministic choice: now mutual exclusion no longer represents commitment, but backtracking.

Thus the new semantics, although originally intended for CC programs, can be used also for describing the behavior of (pure) CLP programs. The only difference is the interpretation of the mutual exclusion relation, which expresses indeterminism when applied to CC programs, and nondeterminism when applied to CLP programs.

4 Local Independence and CLP Parallelization

The semantics obtained above, while being maximally parallel, could be very inefficient if implemented directly as an operational model for CLP. One reason for this is that branches of the search tree may be explored which would have been previously pruned by another goal in the sequential execution. The general problem of finding a rule to avoid the exploration of such branches is directly related to the concept of independence and has been previously addressed in Section 2. In order to avoid such efficiency problems we propose to apply those independence rules, but at the finest possible level of granularity (as proposed in [BGH93]). This is now possible because we have a structure in which all intermediate atomic steps in the execution of a goal and their dependencies are clearly identifiable.

Capturing independence is achieved by identifying dependencies which occur due to subcomputations which affect each other, in the sense of the constraint independence notions above. In our nets, these notions are applied not only at the level of whole computations of different goals, but also at the finer level of subcomputations of those goals, i.e. the actual subcomputations which can affect each other. This new notion of independence (*local independence*) is, to our knowledge, the most general proposed so far (in the sense that it allows the greatest amount of parallelism) which, at the same time, preserves the efficiency of the sequential execution.

A drawback of local independence is that it requires an oracle, since mutual inconsistency of branches is not known a priori, and thus suitable scheduling strategies for AND-OR parallelism must be devised which make sure that the added dependency links are respected (i.e. the strategy is *consistent*), while still taking advantage of the remaining parallelism (i.e. the strategy is, more or less, *efficient*). Such an oracle can be devised at compile-time by means of abstract interpretation based analysis, and a scheduling strategy can be obtained for instance by a suitable program transformation (as that presented in Section 6).

5 A Meta-interpreter of the Concrete Semantics

A meta-interpreter has been implemented which takes as input a CC program and a concrete query, and builds up the associated contextual net as defined by the true concurrency semantics of [MR91], presented in Section 3. The computation of the concrete model is performed in several steps:

1. A program is read in and transformed into a suitable set of context-dependent rules.
2. Starting from the initial (concrete) agent – the query – rules are applied one at a time, until no rule application is possible.
3. Relations of mutual exclusion, causal dependency and concurrency are constructed from the structure given by the previous step.
4. The contextual net giving the program semantics can be visualized in a windows environment, as well as the resulting relations.

Although the method based on rule application to construct the structure is completely deterministic, a fixpoint computation based on memoization is performed in order to ensure termination (whenever the semantics model is finite).

Once the computation is finished, the structure giving the model of the program resembles an event structure [Ros93]. An event structure is a set of events (together with *conflict* and *dependency* relations), where each event represents a single computation step, i.e. a rule application, and contains all the history of the subcomputation leading to the particular step represented. The events represent either program agents, which will be consumed by applying the program rules, or constraint tokens which will be asked for in such rule applications. The former are represented by usual conditions in the net, the latter by *context* conditions.

For simplicity, the current implementation only implements the Herbrand constraint system, leaving to the underlying Prolog machinery much of the entailment relation.

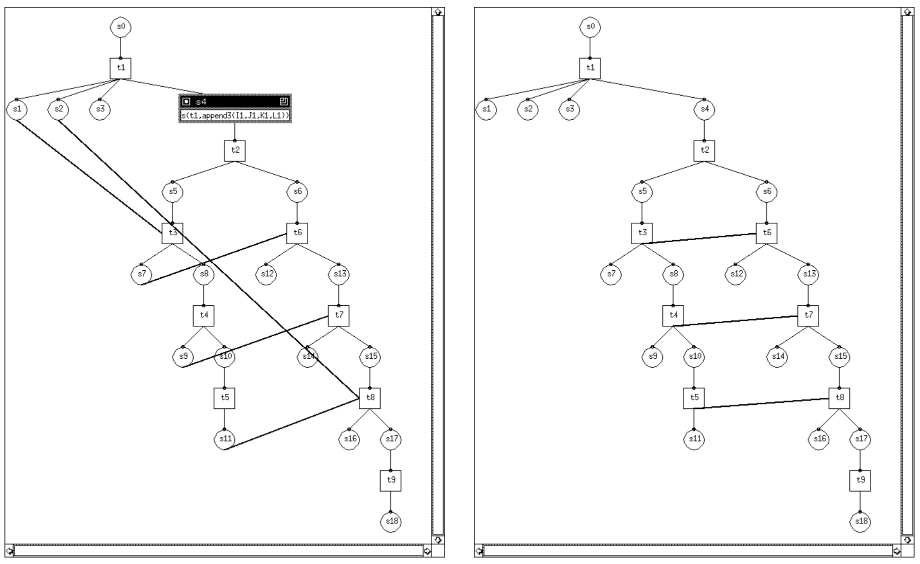


Fig. 1. Contextual Net of the `append3/4` example.

As an example, consider the following definition of `append/3`, which appends two lists into another one, and then splits it into another two. It can be run either first appending and then splitting or "backward" (first splitting and then appending).

```
:- tell(X=[1,2]), tell(Y=[3]), tell(Z=[4]), append3(X,Y,Z,W).

append3(A, B, D, E) :- app(A, B, C), app(C, D, E).

app(X, Y, Z) :-  ask(X = []),    tell(Y = Z).
app(X, Y, Z) :-  ask(X = [A|B]), tell(Z = [A|D]), app(B, Y, D).
app(X, Y, Z) :-  ask(Z = []),    tell(X = []),    tell(Y = Z).
app(X, Y, Z) :-  ask(Z = [_|_]), tell(X = []),    tell(Y = Z).
app(X, Y, Z) :-  ask(Z = [A|D]), tell(X = [A|B]), app(B, Y, D).
```

A query has been included which performs the "forward" computation, where the second `app/3` goal in the body of the `append3/4` clause has to wait on the first goal to proceed at each step while the resulting list `C` is being constructed to consume it. The semantic structure resulting for the computation with this query can be seen in Figure 1.

Circles in the figure correspond to agents (either program agents or tokens) and squares correspond to steps. Thin lines correspond to dependency links, and thick lines to context links. Each element (either a circle or a square) is internally represented as a term with his "history", that is, the set of other elements it depends on. Such term can be explicitely seen by clicking on the element itself (like it can be seen for element s_4 in Figure 1 (left picture)).

The partial order derivable from the net corresponds to the causal dependency relation, plus additional dependencies due to the "use" of contexts. Such order appears in Figure 1 (left picture).

In this way, the causal dependency relation captures an optimal scheduling of processes based on producer/consumer relations on the tokens added to the store. This can be augmented with the local independence relation (as explained in Section 4) to capture and-parallel scheduling based on mutually inconsistent computations.

6 Parallelization of CLP via Program Transformation

One possible application of our semantics can be achieved by program transformation from CLP to CC. The purpose of the transformation will be to allow CLP programs to run under CC machinery with an optimal scheduling of processes which ensures no-slowdown and allows for maximal parallelism. In doing this, the target language should allow for the features of CC, including synchronization and indeterminism (although this latter is not needed for our purposes), and also for additional nondeterminism (in the sense of backtracking - which is indeed needed to embed CLP). Examples of such languages are AKL[2] and concurrent (constraint) Prologs (i.e. Prologs with explicit delay) such as PNU-Prolog [Nai88] and CIAO-Prolog [Her94].

The transformation will proceed as follows. First, the CLP program is rewritten into a CC program. This first step will embed a CLP program into CC syntax, by (possibly) normalizing goals and head unifications, and make all constraint operations explicit as tell agents. Second, inconsistency dependencies are identified within the (abstract) semantics via program analysis, and then the program is augmented with sequentialization arguments where required, and suitable ask and tell operations for this are incorporated to the program clauses.

Consider for example the following CLP program (where tell operations are explicitely specified), with the query `:- p1,p2`.

```
p1 :- tell(c1).
p1 :- tell(c2).
p2 :- tell(c3).
p2 :- tell(c4).
```

and assume that $\{\mathtt{c2}, \mathtt{c3}\}$ is an inconsistent set of constraints. Then the transformed program, containing the required sequentialization, is:

```
p1 :- tell(c1), tell(c).
p1 :- tell(c2).
p2 :- ask(c) -> tell(c3).
p2 :- tell(c4).
```

[2] However, in AKL computations are encapsulated in the so called *deep* guards, an issue that our semantics does not capture yet.

In such a new program, the first alternative (assuming a top-down choice of the clauses) of `p2` is allowed to be executed only if `p1` chooses its first alternative. In this way any of the alternatives for both `p1` and `p2` can be executed in parallel without interaction. In other words, the only interaction needed among such alternatives is explicitely specified by the added ask-tell operations over dummy new constraints (`c` in this case).

The transformed program will allow for or-parallelism (which is captured in the semantics by the mutual exclusion relation) and *locally independent* and-parallelism (which is captured by means of relations derived from the mutual inconsistency relation). An efficient strategy for parallel execution is thus achieved.

7 Static Scheduling in CC via Program Transformation

Another complementary application of the independence detection based on our semantics is schedule analysis. We propose to perform the linearization associated to schedule analysis by means of program transformation from CC to CLP, achieving in addition an efficient parallelization of concurrent goals. In order to do this the intended target language should allow "delay" features able to support concurrency.

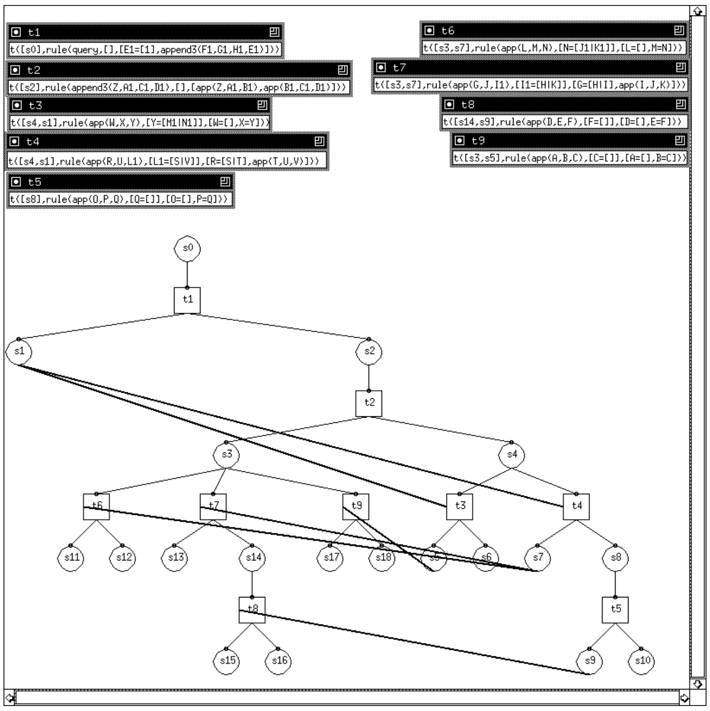
Such features allow dynamic scheduling of processes *a la* concurrent logic programming in (otherwise) sequential languages. One such feature is the `when` declaration [Nai88]. This declaration delays execution of a goal until some conditions are met. Usually conditions relate to meta-logical features of the language and are formed of: `nonvar/1` (true if argument is not a variable), `var/1` (true if it is), `ground/1` (true if argument is ground), etc.

The basic idea of our tranformation is related to the approach of [BGH93] and QD-Janus [Deb93]. However, we propose to perform a more "intelligent" transformation (see also [BGH93]), which is based on the results of the analysis performed over the CC program.

Let us illustrate our approach with the `append3/4` example of Section 5. Assume the following query:

```
:- tell(W=[1]), append3(X,Y,Z,W).
```

The resulting contextual net given by our meta-interpreter is that of Figure 2, where the context dependencies links are shown, and the information corresponding to each rule application ($t_1, t_2, \ldots$) appears explicitly at the top. In the net, it can be seen that only the "backward" version of the predicate `app/3` is used: while the second `app/3` goal in the body of the `append3/4` clause (corresponding to agent s_4) can proceed without suspending, as no context other than the told constraints in the query is needed, the first goal and the goals occurring in its subcomputation always suspend until the third argument becomes instantiated. An identical behavior will occur in all queries in which the three first arguments of `append3/4` are free and the forth is instantiated to a non-incomplete list. With this knowledge the following transformed CLP program can be obtained:



```
t1
t([s0],rule(query,[],[E1=[1],append3(F1,G1,H1,E1)]))
t2
t([s2],rule(append3(Z,A1,C1,D1),[],[app(Z,A1,B1),app(B1,C1,D1)]))
t3
t([s4,s1],rule(app(W,X,Y),[Y=[M1|N1]],[W=[],X=Y]))
t4
t([s4,s1],rule(app(R,U,L1),[L1=[S|V]],[R=[S|T],app(T,U,V)]))
t5
t([s8],rule(app(O,P,Q),[Q=[]],[O=[],P=Q]))
t6
t([s3,s7],rule(app(L,M,N),[N=[J1|K1]],[L=[],M=N]))
t7
t([s3,s7],rule(app(G,J,I1),[I1=[H|K]],[G=[H|I],app(I,J,K)]))
t8
t([s14,s9],rule(app(D,E,F),[F=[]],[D=[],E=F]))
t9
t([s3,s5],rule(app(A,B,C),[C=[]],[A=[],B=C]))
```

Fig. 2. Contextual net for append3/4 running backward.

```
append3(A, B, D, E) :- when(nonvar(C),app(A, B, C)), app(C, D, E).

app(X, Y, Z) :- X = [],    Y = Z.
app(X, Y, Z) :- Z = [A|D], X = [A|B], when(nonvar(D), app(B, Y, D)).
```

In the schedule analysis of [KS92], a dependency relation among literals of the program is used to find optimal sequences of the program clause bodies where the efficient compilation techniques of sequential implementations can be applied. Each such sequence is a *thread.* Threads should not compromise the termination properties of the program. Therefore, where dependencies do not allow to figure out a total ordering of the literals, different single threads must be allocated. Threads will then be dynamically scheduled, while in each single thread, one would like to statically schedule the producer(s) before the corresponding consumer(s), so that the consumers do not need to be suspended and then woken up later. In the specific case of CC programs, the producers are the tell operations and the consumers are the ask operations, so this desirable property of each thread here means that some ask operations could be deleted, if we can be sure that when they will be scheduled the asked constraint has already been told.

By using our semantic structures it is easy to see how this can be done. The order between two goals in the body of a clause can be easily decided by looking at the contextual net describing the behaviour of the original CC program: if the subnets rooted at these two goals are linked by dependencies which all go in the same direction (from one subnet to the other one), then this direction is the order to be taken for the scheduling; if instead the dependencies go in both directions, then the two goals must belong to two different threads; otherwise (that is, if there are no dependency links between the two subnets), we can order them in any way. Once the order has been chosen, each ask operation which is scheduled later than all the items of the net on which it depends on can safely be deleted. Following our example, we reorder `app/3` goals in the `append3/4` clause, obtaining:

```
append3(A, B, D, E) :- app(C, D, E), app(A, B, C).

app(X, Y, Z) :- X = [],    Y = Z.
app(X, Y, Z) :- Z = [A|D], X = [A|B], app(B, Y, D).
```

Our aim is to develop an analysis able to infer such invariants based on the semantics. Such analyzer will guarantee that the transformations applied to a CC program in the spirit above are correct.

Acknowledgments

This research has been partially funded by the ACCLAIM Basic Research Esprit project n.7195 and by the COMPUGRAPH Esprit working group n. 7183.

References

[BGH93] F. Bueno, M. García Banda, and M. Hermenegildo. Compile-time Optimizations and Analysis Requirements for CC Programs. Technical Report CLIP6/93.0, T.U. of Madrid (UPM), July 1993.

[BGHMR94] F. Bueno, M. García Banda, M. Hermenegildo, U. Montanari, and F. Rossi. From Eventual to Atomic and Locally Atomic CC Programs: A Concurrent Semantics. In *Fourth International Conference on Algebraic and Logic Programming*, Springer–Verlag LNCS, September, 1994. To appear.

[BH92] F. Bueno and M. Hermenegildo. An Automatic Translation Scheme from Prolog to the Andorra Kernel Language. In *International Conference on Fifth Generation Computer Systems*, pages 759–769. Institute for New Generation Computer Technology (ICOT), June 1992.

[Col90] A. Colmerauer. An Introduction to Prolog III. *CACM*, 28(4):412–418, 1990.

[Con83] J. S. Conery. *The And/Or Process Model for Parallel Interpretation of Logic Programs*. PhD thesis, The University of California At Irvine, 1983. Technical Report 204.

[Deb93] S.K. Debray. QD-Janus: A Sequential Implementation of Janus in Prolog. Technical Report, University of Arizona, 1993.

[GHM93] M.García de la Banda, M. Hermenegildo, and K. Marriott. Independence in Constraint Logic Programs. In *International Logic Programming Symposium.* MIT Press, Boston, MA, October 1993.

[DeG84] D. DeGroot. Restricted AND-Parallelism. In *International Conference on Fifth Generation Computer Systems*, pages 471–478. Tokyo, November 1984.

[Her94] M. Hermenegildo. Towards CIAO-Prolog - A Parallel Concurrent Constraint System. In *Workshop on the Principles and Practice of Constraint Programming*, LNCS, Springer-Verlag, 1994.

[HR93] M. Hermenegildo and F. Rossi. Strict and Non-Strict Independent And-Parallelism in Logic Programs: Correctness, Efficiency, and Compile-Time Conditions. *Journal of Logic Programming*, 1993. To appear.

[JL87] J. Jaffar and J.-L. Lassez. Constraint Logic Programming. In *ACM Symp. Principles of Programming Languages*, pages 111–119. ACM, 1987.

[JH91] S. Janson and S. Haridi. Programming Paradigms of the Andorra Kernel Language. In *International Logic Programming Symposium*, pages 167–183. MIT Press, 1991.

[KS92] Andy King and Paul Soper. Schedule Analysis of Concurrent Logic Programs. In *Joint International Conference and Symposium on Logic Programming*, pages 478–492, Washington, USA, 1992. The MIT Press.

[MS92] K. Marriott and P. Stuckey. The 3 R's of Optimizing Constraint Logic Programs: Refinement, Removal, and Reordering. In *19th. Annual ACM Conf. on Principles of Programming Languages.* ACM, 1992.

[MR91] U. Montanari and F. Rossi. True-concurrency in Concurrent Constraint Programming. In *International Symposium on Logic Programming*, pages 694–716, San Diego, USA, 1991. The MIT Press.

[MR93] U. Montanari and F. Rossi. Contextual Occurence Nets and Concurrent Constraint Programming. Technical report, U. of Pisa, Computer Science Department, Corso Italia 40, 56100 Pisa, Italy, May 1993.

[Nai88] L. Naish. Parallelizing NU-Prolog. In *International Conference and Symposium on Logic Programming*, pages 1546–1564, August, 1988. The MIT Press.

[Ros93] Francesca Rossi. *Constraints and Concurrency.* PhD thesis, Università di Pisa, April 1993.

[Sar89] V. Saraswat. *Concurrent Constraint Programming Languages.* PhD thesis, Carnegie Mellon, Pittsburgh, 1989. School of Computer Science.

[Sha87] E.Y. Shapiro, editor. *Concurrent Prolog: Collected Papers.* MIT Press, Cambridge MA, 1987.

[VanH89] P. Van Hentenryck. *Constraint Satisfaction in Logic Programming.* MIT Press, 1989.