Parallel Symbolic Computing with the Shared Dataspace Coordination Model

Paolo Ciancarini    Mauro Gaspari

Technical Report UBLCS-94-17
July 1994

Laboratory for Computer Science
University of Bologna
Piazza di Porta S. Donato, 5
40127 Bologna (Italy)
UBLCS Technical Report Series


94-6 Exploring the Coordination Space with LO, S. Castellani, P. Ciancarini, April 1994.

94-7 Comparative Semantics of LO, S. Castellani, P. Ciancarini, April 1994.


Parallel Symbolic Computing with the Shared Dataspace Coordination Model

Paolo Ciancarini\(^2\) \quad \text{Mauro Gaspari}\(^2\)

Technical Report UBLCS-94-17

July 1994

Abstract

Shared Prolog is a language for symbolic computing that combines distributed coordination based on a shared dataspace with sequential symbolic computation based on logic programming. Shared Prolog includes a concept of logically shared dataspace, thus introducing in parallel logic programming a coordination model different from the stream-based communication model of Flat Concurrent Prolog and Strand. We demonstrate how Shared Prolog can be used for expressing a number of different parallel symbolic computing schemata such as: pipeline, divide et impera, and distributed problem solving. We also discuss a number of topics in implementing such a programming model.

---

1. An earlier version of this work appeared under the same title in Proceedings of the ICLP'94 Post-Conference Workshop on Process-Based Parallel Logic Programming, Edited by: K.De Bosschere, J-M Jacquet, A. Brogi, June 1994. Partial support for this work was provided by the Commission of European Communities under ESPRIT Programme Basic Research Project 9102 (COORDINATION), and Italian CNR - Comitato Scienza e Tecnologia dell’Informazione under contract 94.00417.CT12.

2. Department of Mathematics, University of Bologna, Piazza Porta S. Donato 5, 40127 Bologna, Italy.
1 Introduction

Languages for programming parallel systems can be divided in two categories following opposite approaches to parallelism: user-transparent parallel languages and explicitly parallel languages. Advocates of the first field suggest that parallel control flow is much more difficult to master than sequential control flow: a compiler should be used to parallelize programs written with “normal” programming languages. Thus, parallel execution is transparent from the point of view of the programmer; there is no need to involve oneself in the intricacies and details of parallel execution; most important, reuse of existing software on novel parallel architecture is immediate.

Advocates of the second field deny the difficulty, and say that parallel programming is actually more “natural” than sequential programming, especially when the “right” parallel mechanisms are used, and even more efficient, because the programmer has more control over the underlying architecture. This approach allows for a different type of reuse, namely one where software systems can be built by combining existing sequential programs using coordination mechanisms.

The dichotomy between implicit and explicit parallelism is found also in parallel logic programming. There are now several Prolog compilers that allow a transparent parallel execution; these tools pay attention especially to opportunities for independent AND-parallelism and OR-parallelism.

On the opposite side, parallel logic languages like FCP [30] add explicit mechanisms for synchronization and communication to the logic paradigm. These languages have been based mostly on one single model of parallel computation, called the stream-based process model: processes execute logic rules that can spawn new processes; they communicate via streams; special constraints on streams rule the synchronization among producer and consumer processes [30].

From a pragmatic point of view parallel logic languages have been suggested for symbolic programming on massive parallel architectures, but their diffusion in the world outside academia is limited, possibly with one exception: Strand [21]. The main strength of Strand for symbolic computation is the accent put by its designers on coordination: it is offered as a coordination language, i.e., as a language that offers a set of parallel programming mechanisms independent from mechanisms for controlling sequential computation. In fact, various Strand dialect exist: we know at least Strand-C and Strand-FORTRAN.

Strand is not the only coordination language. Linda is an explicitly parallel programming language that introduced the concept of shared dataspace as main tool for coordination [12]. In order to support coordination, Linda offers a set of basic operations on the shared dataspace, that is called tuple space. These operations have been coupled to several “host languages”, like C, FORTRAN, Lucid, Lisp, Scheme, Eiffel, Pascal, Prolog, Ada, etc. All the Linda operations involve a single tuple: \texttt{in} retrieves (deletes) one tuple from the tuple space, \texttt{read} reads one tuple, \texttt{out} adds one tuple to the tuple space, and \texttt{eval} starts an active tuple (i.e., a process). Linda can be considered as an assembler language to interact with a shared dataspace.

There are several models and languages which improve on Linda introducing more high level primitives: Gamma [8], Actor Spaces [1], Linear Objects [6], and ProSet [26], are only a few. We believe that none of these proposals covers the whole space of possible coordination concepts, and surely there is still much room to investigate new concepts.

In this paper we illustrate a coordination model (called PoliS) for parallel symbolic computing based on a shared dataspace; the model includes high level mechanisms to support coordination. PoliS extends Linda with multiple shared dataspaces and a precise interaction
A distinguishable feature of PoliS is the clear distinction between the coordination component and the sequential component, which allows to easily reuse existing sequential programs. Our claim is that PoliS allows to port effectively and economically programs written in a sequential symbolic language, i.e., Prolog, to a general purpose distributed architecture, namely a network of workstations.

We also present a family of languages based on this model and we demonstrate the expressiveness of them in a number of programming examples where coordination plays a major role. The first language, based on a monolithic tuple space, is Shared Prolog, a coordination language that combines Linda and Prolog. Then we will introduce; a) ESP, that extends SP with multiple tuple spaces; b) PESP, that adds some declarative synchronization constraints based on path expressions; and c) COOLL, that adds to multiple tuple spaces new coordination mechanisms useful for object oriented programming.

The structure of this paper is as follows: Sect.2 describes our basic model for coordination, that extends Linda with multiple shared dataspaces. Sect.3 overviews Shared Prolog syntax and semantics, whereas in Sect.4 a number of programming techniques in SP will be exposed. In Sect. 5 we will show the other languages which are instance of PoliS. Sect.6 discusses some implementation issues, and Sect.7 compares SP with other parallel symbolic computing languages.

2 A coordination model

A coordination model is a set of abstract primitives for handling processes and communications that can be added to a sequential language to obtain an explicitly parallel programming language [12]. In short, we can define a coordination model describing three concepts: the coordinating entities, the coordination medium, and the coordinating rules. In a classic view the coordinating entities are processes and messages; here however we are more liberal, and allow several kind of entities: large-grain processes, threads, concurrent objects, active/passive tuples, etc. The coordination media can be shared variables, message channels, private ports, shared dataspaces, etc. Coordination rules state how coordinating entities interact: by synchronization primitives, by sending/receiving messages, by associative access to a dataspace, by complex protocols, etc.

We now describe in turn each component of our model, that is called PoliS.

2.1 Coordinating entities

Coordinating entities are tuples. A tuple is a structured data object that includes a sequence of values. It is produced by some agent, and it remains in the dataspace until some agent consumes it. A tuple can be “copied” (read) or “consumed” (read and deleted). Access to a tuple is associative, i.e., it is done “by contents”. The particular access mechanism chosen is a degree of freedom in our model: we can accommodate either a mechanism based on typed pattern matching, as in Linda [25], or a mechanism based on unification, as in a logic language.

Some tuples are active: they denote computations. Tuples that are not active are passive: they are data, or messages. We call active tuples also “agents”: an agent is an execution thread, i.e., it is an abstraction of a running program. Each agent is able to perform some operations on tuples, both passive and active.

- associative test of a tuple contained in the dataspace; in our model the test operation has a broader semantics than read in Linda: a number of predefined tests on the dataspace
are allowed, depending on the chosen type system for tuple arguments.
- associative consumption of a tuple contained in the dataspace; this operation also has a broader semantics than in in Linda: several tuples can be collected and consumed by special operators that return multisets.
- asynchronous creation of a tuple inside the dataspace.

2.2 Coordination rules
The coordination of agents is defined by rules. In fact, programs in this model consist of sets of coordination modules; each module is a set of coordination rules.

A coordination rule consists of a protocol including four steps:
- i. A rule atomically tests and consumes tuples in the dataspace: this is a guard that activates an agent.
- ii. When the guard is satisfied, the agent “reacts” and starts a “local” computation that has no effect on the dataspace; such a computation is expressed in a sequential programming language. The coordination model does not prescribe any specific language, so that agents written in many different sequential languages can coexist.
- iii. When the local execution terminates the agent creates tuples in the dataspace.
- iv. Eventually, the agent terminates and disappears from the dataspace (however, we can specify an ever-lasting agent by inserting among its outputs the creation of a tuple that is a “copy” of itself).

Conceptually, each agent evaluates in OR-parallel several coordination rules; these may be thought of as methods that allow to activate the services offered by agents.

2.3 Coordination media
The coordination medium is the tuple space, namely an unbounded multiset of tuples.

In PoliS we generalize the Linda’s Tuple Space concept, allowing multiple tuple spaces as coordination media. A collection of tuple spaces is called a polispace. The simplest polispace contains one dataspace only: this is the monolithic tuple space as it is found in Linda (and Shared Prolog). Multiple tuple spaces can be organized in several ways. Each dataspace has a name and possibly a set of attributes, that rule the exchanges of tuples among them. Another attribute that a dataspace can have is an “invariant”, that is a predicate on its contents that has to be true for all the life of the dataspace itself. When it is violated, the dataspace terminates.

3 Shared Prolog
When it was introduced in [9], Shared Prolog was a language based only partially on the coordination model presented above. In the past years there have been several adjustments both in syntax and semantics. To present the language in its current form, we show what are the coordination medium, the coordinating entities, and the coordinating rules.

The coordination medium in SP is a dataspace containing logic tuples. Formally, the shared dataspace is a multiset of logical tuples that have the form of Prolog terms: variables as arguments of terms are allowed, and they only scope for the term they belong to.

The entities to be coordinated are agents that execute Prolog programs encapsulated in modules called theories, which also include the coordination rules. More precisely, a theory defines the coordination of an agent with respect to the dataspace, using rules that follow the PoliS protocol described in Sect.2.2.
3.1 The logic dataspace

A logic dataspace is a container of logic tuples, i.e., Prolog atoms (constants or terms). The initial contents of a dataspace are described by a special goal that is introduced by the keyword `dataspace` and specifies a (possibly empty) bracketed multiset of tuples.

\[
\text{dataspace } \{ a_1, \ldots, a_n \}.
\]

Logic variables in the dataspace have scope only for the tuple to which they belong.

Example:
The following dataspace contains three logic tuples:

\[
\text{dataspace}\{i(5). \; s(X). \; buffer(4.\{2,3\})\}
\]

The second tuple includes a variable.

3.2 Theories

A program is composed of a set of theories. Each `theory` has the following syntactical structure:

\[
\begin{align*}
\text{theory } & \text{name}(V_1, \ldots, V_n) \\
& \text{eval } \text{rule}_1# \ldots # \text{rule}_k \\
& \text{with } \text{Prolog program}
\end{align*}
\]

A theory is identified by a name: a functor with zero or more arguments \( V_i \) that are logic variables that scope over the rules. The `theory interface` follows the keyword `eval` and includes some rules, separated by the symbol `#`; the `theory implementation` is the Prolog program that follows the keyword `with`. The set of rules of a theory can be considered the interface "visible from the outside" of a module, while the Prolog program is the "invisible" implementation of the module itself. Agents react to the contents of the dataspace according to the theory interface.

Rules include a `guard`, which is composed by a sequence of test and consume operations, a `goal` to be evaluated with respect to the `Prolog program`, and finally some `out` operations.

\[
\text{Guard } \rightarrow \text{Goal}\{\text{Success}\}\text{ fail}\{\text{Failure}\}
\]

Test operations are written as goals, whereas consumption and creation operations are put between braces, to mean that they have side effects on the dataspace.

If the `Guard` is satisfied, i.e., when all its test and consume operations are completed, the rule can commit: then the `Goal` is evaluated with respect to the Prolog program. To deal with the possibility of a failure of such a Prolog goal, creation operations are partitioned in two sets separated by the keyword `fail`: if the goal evaluation succeeds the `Success` out-set is added to the tuple space, otherwise the `Failure` out-set is added to the tuple space.

A notable feature of SP is that control flow of `test` and `consume` operations is ruled by backtracking. Each `test` or `consume` operation either succeeds or fails; a failure activates backtracking to the preceding operation. The formal semantics of such a mechanism has been studied in [9, 23].

Tests can also include built-in predicates which are:

- the set of relational operators;
- the `is` predicate, with the same meaning it has in Prolog;
- the `all(Tuple, List)` operator which takes two arguments and returns in the second the list of all the tuples which unify the first;
- the `var` and `nonvar` predicates with the same meaning they have in Prolog.

\[
\text{Guard } \rightarrow \text{Goal}\{\text{Success}\}\text{ fail}\{\text{Failure}\}
\]
The body is a normal Prolog goal to be evaluated with respect to the knowledge base. The knowledge base after with includes a set of Prolog procedures.

The goal in the body can be either normal, i.e., only one solution is returned, if any, or an all-solutions generator goal. The formal semantics of a generator goal is described in [23]; briefly, a generator goal generates all the terms that solve a given goal exploiting Prolog backtracking and for each solution adds the corresponding term to the tuple space; finally, when failure occurs it adds the failure out-set to the dataspace.

3.3 Initial goal
Each agent executes a theory, whose heading matches the active tuple representing the agent itself. The set of active agents is statically defined by an initial goal rule, which contains a list of agents separated by the symbol “||”.

?- TheoryName1 || ... || TheoryNameN.

Theory names in the initial goal are matched at compile time with the theory definition and the code of the active agent is determined. Several agents instances of the same theory can be active at the same time.

4 Some programming techniques in SP
In this section we show some programming techniques in SP. First we show how simple synchronization mechanisms are implemented in SP. Then we describe the language expressiveness giving solutions to the N-queens problem based on three different schemas of parallel programming: pipeline, divide et impera, and distributed problem solving. The first two parallel programs are obtained from sequential programs by adding coordination rules to the sequential Prolog predicates.

4.1 Expressing concurrency
4.1.1 Mutual exclusion
As first programming example, we show a solution to the classic Dining Philosophers problem. In the initial dataspace there are the forks and the number of philosophers. The initial goal and the initial dataspace are defined as follows.

% initial dataspace
dataspace {n(5),fork(0),fork(1),fork(2),fork(3),fork(4)}.
% initial goal
philo(0) || philo(1) || philo(2) || philo(3) || philo(4).

This goal starts five agents, all executing the theory philo(X). A philosopher is identified by the argument X.

The theory philo(X) is specified as follows:

theory philo(X) % a dining philosopher
  eval
    not ready(X)
    ->
    thinks
    {ready(X)}
Such a theory includes two rules only: the first one is used to think, the second one to eat. Initially all the agents are not ready. The first rules says that when the tuple \texttt{ready}(X) is not present in the tuple space an agent \texttt{philo}(X) can only think; after thinking the tuple \texttt{ready}(X) is inserted in the tuple space and the agent is ready to eat.

The guard of the second rule says that if the agent is ready and both forks can be grabbed, the agent can start eating. After eating the agent puts back the forks in the tuple space. It is easy to see that this program avoids deadlock because the agent gets both forks at the same time as an atomic action.

### 4.1.2 Centralized bounded buffer

Direct communication between two set of agents, \textit{i.e.}, producers and consumers, can be obtained using a buffer represented by a tuple accessible by both the producers and the consumers. The dataspace initially contains an empty buffer; \texttt{N} is the max number of messages. The initial goal activates two producer and two consumer agents.

\begin{verbatim}
% datasource rule
dataspace {buffer([], N).
% initial goal
producer || producer || consumer || consumer.

% theories
theory producer
eval
  {buffer(L,N)}, N>0, N1 is N-1
  →
  produce(E)
  {buffer([E|L],N1)}
  with
 produce(E):- ...

theory consumer
eval
  {buffer([E|L],N)}, N1 is N+1
  →
  consume(E)
  {buffer(L,N1)}
  with
consume(E):-...
\end{verbatim}

If the buffer has length equal to one, this form of communication is equivalent to rendez-vous communication. In this implementation the buffer is fully retrieved from the dataspace for every communication step: this may limit the potential parallelism of the system.
4.1.3 Distributed bounded buffer

We show now an alternative implementation of the bounded buffer which use tuples to represent buffer elements, and indexes to manage a circular structure.

```
% dataspace rule
dataspace {first(0), last(0), max(N)}.
% initial goal
producer || producer || consumer || consumer.

% theories
theory producer
eval
  max(N), first(F), last(N), N1 is N+1 mod M, F =\<\< N1
  \rightarrow
  produce(E),
  {buffer_elem(E,N), last(N1)}
with
  produce(E):- ... .

theory consumer
eval
  max(N), first(F), buffer_elem(E,F), F1 is F+1 mod M,
  \rightarrow
  consume(E),
  {first(F1)}
with
  consume(E):- ... .
```

The buffer is implemented as a circular array managed with a FIFO discipline. The dataspace initially contains two tuples `first(F)` and `last(N)` indicating respectively the first and the last element in the array. The producer can send messages only if the buffer is not empty; this is true if the index `F` is different from `N+1`. The consumer retrieves just one element and increments the `first(F)` index.

4.1.4 Synchronous Communication

We have shown how to coordinate two agents via a bounded buffer of size 1. It is possible to describe rendez-vous synchronization directly in the rules of a theory. The general schema is the following: the receiver waits for messages in each rule, whereas the sender checks that no pending messages are on the dataspace.

```
theory sender
eval
  not msg(...) % checks for pending messages
  \rightarrow
  produce(msg(...)),
  {msg(...)} % writes the message
with
  produce(...).

theory receiver
eval
```

8
For instance, suppose we program two processes communicating synchronously by messages: agent \texttt{player(1)} after having received a ping tuple sends a pong tuple, whereas process \texttt{player(2)} after having received a pong sends a ping.

```
{msg(..., ...)}  % waits for the message
\rightarrow ...
```

```
{msg(..., ...)}  % waits for the message
\rightarrow ...
```

```
with ...
```

There is only one theory and one rule; the different instantiations of variable \(X\) allow to build two different player agents both using such a theory.

### 4.1.5 An anonymous server

A set of clients can invoke the services of a set of homogeneous servers (this problem is taken from [24]). A client invokes a server issuing the tuple \texttt{connect(Client)}. A free server accepts the message and answers to the client with a message \texttt{connected(Client)}. Now the client can ask for service, starting a session consisting of message exchanges. The session ends with a message \texttt{disconnect(Client)} sent by the client.

The following program includes \(N\) client agents and \(M\) server agents.

The dataspace initial contents and the initial goal are as follows:

```
% initial datasetspace
dataspace \{\texttt{free_server(1),..., free_server(n)}\}.
```

```
% initial goal
server(1) || ... || server(n) || client(1) || ... || client(m).
```

Each server is implemented by the following theory:

```
theory server(K)
eval  
{free_server(K), connect(U)}
\rightarrow % start connect  
{connected(U, server(K))}
#
```
4.2 Exploring coordination

In this section we show three programs, providing different solutions to the N-queens problem. This is a classic problem in which given a chessboard of side N we have to place N safe queens. A queen is safe if it is not attacked by another queen. Two queens attack each other if they are on the same row, column, or diagonal. This problem has been largely used as a benchmark to compare and test the efficiency of language implementations, especially in parallel logic programming [32, 33].

In order to show the flexibility of Shared Prolog in defining different coordination protocols, we shall give three different solutions to this problem: one based on a generate and test technique exploiting pipeline parallelism, a second based on a divide et impera technique exploiting master-worker parallelism, and finally a solution based on a distributed problem solving model.

The first two programs are based on well known sequential algorithms: we add coordination rules to obtain the parallel version of the algorithm leaving the main predicates unchanged. The third program shows how Shared Prolog rules can express explicitly distributed backtracking.

4.2.1 Pipeline Parallelism

In this section we describe an implementation based on the Bruynooghe’s naive generate-and-test algorithm [32] which finds all the possible solutions, represented as permutations of a list of N integers. The Nth position in the list indicates the Nth column on the chessboard. An integer R in position Nth inside the solution states that the queen is placed on the Nth column and on the Rth rows on the chessboard.

```prolog
queens(N):- range(1,M,L),permutation(L,Qs),safe(Qs).
permutation(Xs,[Z|Zs]):-
   select(Z,Xs,Ys),permutation(Ys,Zs).
permutation([],[]).
select(X,[X|Xs],Xs).
select(X,[Y|Ys],[Y|Zs]):=select(X,Ys,Zs).
range(M,N,[M|N|Ns]):= M < N, M1 is M + 1, range(M1,N,Ns).
range(M,N,[M|N]).
safe([Q|Qs]):=safe(Qs), not attack(Q,Qs).
safe([]).
attack(X,[Xs]):=attack(X,1,Xs).
attack(X,[Y|Ys]):=Y+X; X is Y-N.
```
The predicate permutation/2 generates all the permutations of a list, while range/3 is used to build the initial list (e.g., range(1,N,X) generates the list X=[1,...,N]). The predicate attack/2 takes two arguments: a queen Q and a list of queens and checks the one of the queens in the list attack Q.

In the SP solution we present there are two theories: the first one is a generator theory generating all the possible permutations of the list [1,2,3,...,9]; the second theory is safe, and it checks if a proposed permutation is also safe, thus finding a solution of the problem.

\[
\text{theory generator}(N) \\
\text{eval} \\
\{\text{start}\} \rightarrow \text{range}(1,N,L) \{\text{root}(L)\} \\
\# \\
\{\text{root}(L)\} \rightarrow \\
*\{\text{permutation}(L,L)\} \{\text{perm}(L)\} \\
\text{with} \\
\text{permutation}... \\
\text{select}... \\
\text{range}...
\]

The first goal of the generator(N) theory produces a template list [1,...,N], that is used in the second rule. The second rule starts a generator goal which generates all the permutations in form of tuples and inserts them to the tuple space.

\[
\text{theory safe} \\
\text{eval} \\
\{\text{perm}(X)\} \rightarrow \text{safe}(X) \{\text{found}(X)\} \\
\text{with} \\
\text{safe}... \\
\text{attack}...
\]

Agents executing the safe theory retrieve permutations and test if they represent safe solutions.

\[
\% \text{ initial dataspace} \\
\text{dataspace} \{\text{start}\}. \\
\% \text{ initial goal} \\
\text{generator}(9) \parallel \text{safe}.
\]

The initial dataspace contains only the atom start. The initial goal activate a generator agent and a safe agent. We could even activate several safe agents, i.e., as many as the processors that are available.

### 4.2.2 Master-Worker

The example in the previous section is not an efficient solution to the problem, because the messages that are exchanged are proportional to the permutations to be tested. The solution we show now is based on the Bruynooghe’s optimized (fused) algorithm.

\[
\text{queen}(\{}P,R,P\}. \\
\text{queen}(\{}R|T\},R,P)\rightarrow \text{select}(A,\{}R|T\},L). \\
\text{safe}(R,A,I).\text{queen}(L,\{}A|R\},P).
\]
The SP solution is based on a *divide et impera* approach, exploiting the generator goal inside a master agent that distributes tasks to some workers. The theory `master` is used by a master agent to generate a list of subproblems to be solved.

```prolog
theory master(N)
  eval
  {start}
  →
  range(1..N) % creates the ordered list
  {root(L)}
  #
  {root(L)}
  →
  *(divide(L,L1)) % creates permutations
  {perm(L1)}
  with
  divide(Xs,[Z|Ys]):=select(Z,Xs,Ys).
  select... range...
```

The `worker` theory describes the job of each worker: it takes a subproblem and returns the list of solutions it found.

```prolog
theory worker
  eval
  {perm([H|T])} →
  fq(T,H,L) {found(LP)}
  with
  fd(T,H,L):-findall(X,queen(T,[H],X),L).
  queen...
  select...
  safe...

% initial dataspase
dataspase {start}.

% initial goal
master(9) || worker || worker || worker || worker || worker || worker || worker || worker.
```

The initial dataspase contains only the atom `start`. The initial goal activates a `master` agent and nine `worker` agents, *i.e.*, as many workers as queens to be placed. The workers retrieve subproblems from the dataspase, solve them, and put back the solution in the dataspase.

**4.2.3 Distributed Problem Solving**

In this solution a set of agents cooperate by backtracking on the state of the problem solving process that is represented explicitly on the dataspase. There are three kinds of theories:

- *queen*: places a queen on the chessboard;
unsafe discover unsafe queens;

bktrack backtrack the position of a queen.

The problem solving activity is started by queen agents, which place queens on the chessboard. At the same time unsafe agents check for unsafe queens and remove them from the chessboard. When no queen can be placed and there is still an empty column a bktrack agent removes some queens and the problem solving activity starts again. When a safe position is reached no rule can be applied and the problem solving process terminates.

The chessboard is represented on the dataspace, for each row of the chessboard we have the following data:

- go(Q,X) means that the queen Q is currently not on the chessboard; X records the last position used for that queen.
- pos(Q,Y) means that queen Q is on column Y.
- list(Q,L) records other tries to attempt if this position will be discarded.

The queen theory places queens identified by the go/2 tuple. The first rule deals with the general case while the second and the third rules are introduced respectively to place a queen on the first or on the last column.

```
theory queen
val
  {go(X,Y) : Y is X + 1,
   list(X,[E|L1]),list(Y,L2)}
  -> delete unsafe(E,L2,L3)
     {pos(E,X), list(X,L1), list(Y,L3)}
#  {go(X,Y) : dim(X), list(X,[E|L1])}
  -> {pos(X,E), list(X,L1)}
#  {go(1,Y) : list(1,[E|L1])}
  -> {pos(1,E), list(1,L1)}
with
  delete unsafe(E,[]).
  delete unsafe(E,[E|T],T1) :-
    !, delete unsafe(E,T,T1).
  delete unsafe(E,[H|T],T1) :-
    E is H - 1, !, delete unsafe(E,T,T1).
  delete unsafe(E,[H|T],T1) :-
    E is H + 1, !, delete unsafe(E,T,T1).
  delete unsafe(E,[H|T],[H|T1]) :-
    delete unsafe(E,T,T1).
```

Two queens attack each other if they are on the same column (first rule), or they are on the same diagonal (second and third rule).

```
theory unsafe
val
  pos(X1,Y), {pos(X2,Y)}, X1 < X2
  -> {go(X2,Y)}
#  pos(X1,Y1), {pos(X2,Y2)}, X1 < X2,
    Z1 is X1+Y1, Z2 is X2+Y2, Z1=Z2
  -> {go(X2,Y2)}
```
If a queen Q has to be placed (no atom $pos(Q,P)$) and no position has to be tried (list $(Q,[])$), the upper queen $(pos(Q-1,Y))$ must be moved, inserting in the dataspace the tuple $(go(Q-1,Y))$. As a consequence, all the queens that follow, *i.e.*, such that $(pos(K,X))$ and $(K \geq Q)$, must be replaced, thus, the tuples $(go(K,X))$ are inserted in the dataspace.

```
theory bktrack
  eval
  bblist(L). {list(X,[])}. not pos(X,Y).
  Z is X - 1. {pos(Z,Y)}
  → {go(Z,0),list(X,L),btrk(X)}
#
  bblist(L). {btrk(X)}. Z is X + 1.
  {pos(Z,P),list(Z,[])}
  → {go(Z,0),list(Z,L),btrk(Z)}
#
  {btrk(Z),dim(Z)} → {dim(Z)}.
```

The initial dataspace and the initial goal are defined as follow:

```
% initial dataspace
dataspac{dim(N),bblist([1,...,N])}.
  go(1,0)....go(N,0).
  list([1,...,N])....list([1,...,N]).
% initial goal
queen || queen || queen || unsafe || unsafe || bktrack.
```

Such a solution is very inefficient, because many messages are exchanged to support distributed problem solving and backtracking. However, it is a model example for problems where it is necessary to define a backtracking strategy that works in a distributed setting.

### 4.2.4 Performance comparison

We have recorded the performance achieved in the three different solutions. The programs run on a workstation cluster consisting of 9 Sun Sparcstations 1 on a Ethernet LAN. The implementation we have used is presented in [23]; it is based on a coordinator process which evaluates coordination rules and manages the tuple space. We have tested two runtime configurations: in the first the coordinator process runs on a Sparcstation 1, in the second it runs on a Sun Sparc ELC which is about 1.7 times faster.

As expected, there are large differences among the three solutions we have reported. In particular, it should be noted that the programs do not scale when the number of workers is greater than 6. This is due to the high cost of communications in reading and writing tuples in the dataspace. This problem can be solved by providing a better coordination medium or a faster machine for the coordinator, as shown in the table when we use an ELC.

In order to give an idea of comparable solutions implemented with parallel logic languages, Fig. 2 contains the results obtained with a variant of the *divide et impera* algorithm which does not return the solutions in the dataspace. We compare our results with data reported in [33] running a) an or-parallel Prolog algorithm (The “Bruynooghe fused generate-and-test”), and
<table>
<thead>
<tr>
<th>Workers</th>
<th>2+1</th>
<th>4+1</th>
<th>6+1</th>
<th>8+1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Master-Worker</td>
<td>0.566</td>
<td>0.475</td>
<td>0.468</td>
<td>0.446</td>
</tr>
<tr>
<td>Master-Worker + (ELC)</td>
<td>0.465</td>
<td>0.416</td>
<td>0.315</td>
<td>0.308</td>
</tr>
<tr>
<td>Pipeline</td>
<td>170.905</td>
<td>154.915</td>
<td>156.223</td>
<td>158.552</td>
</tr>
<tr>
<td>Pipeline + (ELC)</td>
<td>128.993</td>
<td>104.007</td>
<td>104.337</td>
<td>103.615</td>
</tr>
<tr>
<td>Distr. Prob. Solving (1 sol)</td>
<td>36.023</td>
<td>34.438</td>
<td>37.706</td>
<td>37.856</td>
</tr>
</tbody>
</table>

Figure 1. Comparing the 7-Queens programs (times in seconds)

<table>
<thead>
<tr>
<th>Workers</th>
<th>2+1</th>
<th>4+1</th>
<th>6+1</th>
<th>8+1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Master-Worker (9 queens)</td>
<td>2.013</td>
<td>1.720</td>
<td>1.690</td>
<td>1.675</td>
</tr>
<tr>
<td>Master-Worker (9 queens) + (ELC)</td>
<td>1.506</td>
<td>1.214</td>
<td>1.207</td>
<td>1.154</td>
</tr>
<tr>
<td>Master-Worker (9 queens) no read</td>
<td>1.251</td>
<td>0.770</td>
<td>0.746</td>
<td>0.597</td>
</tr>
<tr>
<td>Prolog on Sequent Symmetry (9 queens)</td>
<td>12.4</td>
<td>6.8</td>
<td>-</td>
<td>5.3</td>
</tr>
<tr>
<td>FGHC on Sequent Symmetry (9 queens)</td>
<td>25.2</td>
<td>12.8</td>
<td>-</td>
<td>6.6</td>
</tr>
<tr>
<td>Master-Worker (11 queens) no read</td>
<td>16.466</td>
<td>8.595</td>
<td>6.412</td>
<td>5.386</td>
</tr>
</tbody>
</table>

Figure 2. Comparing performances (times are given in seconds)

b) the same algorithm translated in FGHC and executed on a Sequent Symmetry (a shared memory architecture with eight 80386 processors). We report the running time for the 9 queens problem which has 352 solutions. The running time of the sequential algorithm is 1.150 sec on a Sparcstation 1 and 24.4 on a single processor of the Sequent Symmetry.

In the same table we report the performance of our algorithms exploring the 11 queens problem. The sequential program takes 27.020 seconds on a Sparcstation 1 to find all 2680 solutions. It is clear that the hardware we have chosen imposes high communication costs, and that good performances can be expected only if the programmer is able to exploit large-grain parallelism.

5 A family of languages based on PoliS

The main application of SP was as a specification language for multiuser software development environments able to support the software process [17]. In this section we describe three languages based on PoliS; they extend SP to handle different application domains.

Extended Shared Prolog (ESP) [10] is an extension of Shared Prolog which includes multiple tuple spaces. ESP has been chosen as the design language for Oikos, a multiuser distributed software development environment [3].

PESP [28] is a variant of SP enriched with path expressions and and used as prototyping (“animation”) language for specifications written in the Z specification language. The problem of enriching theories with path expressions has been studied also in [5], where the language PATE’ is presented. PATE’ as been chosen as the successor of ESP in Oikos.

Finally, COOLL [15] demonstrates how an object oriented programming style can be easily obtained as an instance of PoliS.
5.1 Programming with multiple tuple spaces

The most obvious extension to a language with a unique dataspace as a coordination medium is to have multiple tuple spaces, in order to increase modularity of the system. This is the idea at the basis of Extended Shared Prolog.

5.1.1 Extended Shared Prolog

An Extended Shared Prolog (ESP) program defines a hierarchical system of nested dataspaces [10]. Zero or more agents can be “connected” to each dataspace. An agent can read only the contents of the dataspace to which it is connected, whereas can write tuples everywhere in the dataspace system.

There are two distinct sets of names: a set of dataspace names, and a set of theory names. Each dataspace has a distinct name, while agents are identified by the name of their theory and the name of the dataspace they belong to. A dataspace name is a path in Unix style, e.g., /os/users/user1 is the name of a dataspace user1 that is nested in dataspace user that is nested in dataspace os. Constants prefixed with the symbol @ denote addresses, i.e., they are names of dataspaces.

Differently form Shared Prolog, ESP is a dynamic language where it is possible to create at runtime both new dataspaces and agents.

This is the syntax of the ESP construct to create a dataspace and some agents:

\[
\text{act Agent}_1 \parallel \ldots \parallel \text{Agent}_n \{a_1, \ldots, a_n\}@\text{DataspaceName}
\]

In ESP there is still no concept of active tuple and “chemical evaluation” as in PoliS: agents have to be explicitly created with an ad hoc construct (act).

5.1.2 Invariants

Whereas agents are ephemeral and stateless, a dataspace can be seen as an object that is persistent and has a state. Hence, in ESP a dataspace is not a passive entity, a mere repository of tuples or a channel for messages. There is a way of controlling the activities that take place inside a dataspace. In fact, for each dataspace we can define one or more invariants, i.e., constraints that must hold for all the dataspace life span. Whenever an invariant is violated, the dataspace stops all activities and terminates.

Invariants are defined by special rules where the heading is substituted by the keyword invariant.

\[
\text{invariant} :- \text{Test} \rightarrow \text{Goal} \{\text{Out}\}
\]

The Test section defines a condition on the dataspace that, when verified, terminates the dataspace itself. The Goal and the Out sections can be used to compute and communicate any results to other dataspaces.

Example:

The following is an invariant:

\[
\text{invariant} :- \text{acceptable(result)} \rightarrow \{\text{start(newprocess)@manager}\}
\]

when tuple acceptable(result) is produced, the dataspace terminates, communicating tuple start(newprocess) to the tuple space named manager.
The invariant concept is not present in Linda. In ESP it is useful to specify the intended semantics of a dataspace: when the condition specified by the invariant is verified, a "result" has been obtained; it can then be passed to some other dataspace.

5.1.3 Programming in ESP

We show now a simple example that deals with logical distribution in ESP: a bank network system.

A client uses a tty and an identification card to withdraw or deposit money. The tty is a cash dispenser that can belong either directly to his own bank (the customer account is local) or to another one (the customer account is remote, i.e., belonging to another bank). The tty is ruled by a teller agent, that decides if the client is trusted and can obtain the required money.

The bank network system is represented by a polispaces including three banks. Each bank is itself a dataspace and contains client accounts (represented by tuples).

The interaction of a customer with a terminal is ruled by a shell theory called tty. The initial goal specifies a set of nested tuple spaces that represent the central bank office and its branches. To each branch is connected a tty agent and a teller agent. The switch and create_new agents are both connected to the tuple space representing the central office.

A tty agent is ruled by the following theory:

```plaintext
definitions:
theory tty/(X/)
eval
  self/(Dispenser/) {request/(Name,Query/)}
  └→ % get the request and send it to the teller agent
      check/(Query/)
      {query/(Dispenser/,Name,Query/)}
  % {give/(Money/)}
  └→ % give money to the user
      {output/(Money/)}
  #
  self/(This_bank/) {end/}
  └→ % detects local termination
      {terminate/(This_bank/)}
  #
  terminate/(Z/,not done/)
  └→ % sends a termination msg to the central bank
      mother/(Z/,M/) {done/,terminate/(Z/)@M}
      with
      check/(deposit/(Amount/)).
      check/(withdraw/(Amount/)).
      mother/(X/,[]):-!.
      mother/(H|T/,[H|TN/]):-mother/(T,TN/).
```

The check predicate states the possible operations. The mother predicate is used to access to the parent dataspace in the hierarchy. Accounts (either local or remote) are handled by teller agents, declared as follows.

```plaintext
definitions:
theory teller/(X/)
eval
  {query/(Bank,Name,withdraw/(Amount/),account/(Name,Balance/)}
```

17
Amount =< Balance, NewBalance is Balance - Amount
→ % local account withdraw
  write(account(Name,NewBalance))
  \{account(Name,NewBalance). give(Amount)@Bank\}
#
  \{query(Bank,Name,withdraw(Amount)))\ account(Name,Balance). Amount >= Balance
→ % incorrect operation
  \{overdraft(Amount)@Bank\}
#
  \{query(Bank,Name,deposit(Amount)), account(Name,Balance)\}
  NewBalance is Balance + Amount
→ % local account
  write(account(Name,NewBalance))
  \{account(Name,NewBalance). certificate(Amount)@Bank\}
#
  \{query(Bank,Name,Query)) not account(Name,_)\n→ % non local account, send it to the central bank
  \{query(Bank,Name,Query)@[banknetwork]\}

The remote answers are directly dealt with by the tty theory.
An out-set can be sent to an address different from the default dataspace on which the agent is activated, i.e., the dataspace that is currently visible (as identified by the self predicate).
The teller transmits a non local request to the top level (network) dataspace. A number of switch agents rule the behavior of the network in case of non local requests.

theory switch(X)
eval
  \{query(Source,Name,Query)} customer(Name,Bank), self(Root)
  →
  append(Root,Bank,Dest)
  \{query(Source,Name,Query)@Dest\}
with ...

New bank branches can be also started from the central office.

theory create new
eval
  \{newbank(BankName,Pos,D),banknumber(X)\} Xi is X + 1
  →
  \{act tty(X1) || teller(X1) || interface(D,Pos) @BankName\}
  \{acth(switch(X1))\}
  \{banknumber(X1)\}
#
  self(ME) \{client(Name,BankName)\}
  →
  append(ME, [BankName], Address)
  \{customer(Name,[BankName]), account(Name,0)@Address\}
with ...

The goal
activates a new dataspace representing the new branch, whereas the goal
\{acth(switch(Xi))\}
activates another switch agent.

5.2 Imposing constraints on coordination rules

Programming practice has shown that the use of asynchronous communication as the only coordination mechanism in Shared Prolog rules may compel programmers to be too verbose: the programmer has to introduce several special tuples in the theories with the unique aim of introducing synchronization among rules. This problem can be avoided introducing a mechanism of coordination to declare mandatory sequences of rule activations: such a declarative mechanism is the path expression [11]. The idea is to enrich the interface of a theory with a path expression describing the allowed sequence of synchronizations of coordination rules with respect to the shared dataspace. This is the basis of Path Expression SP (PESP).

Each coordination rule in PESP has a name, and the path expression determines some synchronization constraints on the sequence of activations of the rules in the theory.

theory name(V1, ..., Vn)
select pathexpression
eval name1 : rule1#...#namek : rulek
with Prolog_program

The path expression after keyword select is a regular expression over names of coordination rules. For instance, a PESP theory for a philosopher is the following:

theory philosopher(J)
select Thinks ; Takesforks ; Eats
eval
Thinks :
\{thinks(J)\}
\rightarrow
\{takes(J)\}
#
Takesforks:
\{takes(J)\}, K is (J+1) mod 5, \{fork(J), fork(K)\}
\rightarrow
\{eats(J)\}
#
Eats:
\{eats(J)\}, K is (J+1) mod 5
\rightarrow
\{fork(J), fork(K), thinks(J)\}

The use of the path expression simplifies the rules, because now synchronizations among rules of the same theory via “private” tuples are dealt with the path expression.

PESP has been introduced as a tool for refining specifications written in the Z specification language. Since Z is not executable, some researchers have attempted to develop methods to “animate” Z specifications, i.e., to obtain executable prototypes that correspond to the specification; one of the most used languages for Z animations is Prolog. PESP has been introduced to support distributed animation of specifications written in Z [28]. A forthcoming paper will describe the method with more details.
5.3 Towards rule based object-oriented coordination

Similarly to ESP, COOLL (Coordination, Objects, and Linear Logic) is based on multiple tuple spaces [15]. However, its basic computing model is different because it is based on an abstract notion of active objects inherited from the LO (Linear Objects) language [6].

A COOLL program is composed of a set of classes; each class is a collection of methods (rewriting rules) specifying the behaviour of an object. A computation in COOLL can be thought of as the evolution of a system of communicating objects, where each object has a private dataspace; the object state is represented by the contents of the associated dataspace, consisting of a multiset of atoms; transitions are expressed by multiset rewritings. Each object can test and modify its own dataspace, but cannot access the dataspace of other objects. Objects are dynamic entities, in fact they can terminate and can be dynamically created. An COOLL object can force other objects to terminate, replacing them with a new object.

The class construct in COOLL is very similar to a theory in SP; however, COOLL methods are more structured than SP coordination rules. A method includes three parts; each part specifies a different phase of the application of such a method. The first part is the guard that specifies conditions for method triggering; in particular an object can read without consume tuples into its context. The second part specifies communications, and the third part specifies a transition to a new dataspace configuration.

Two forms of concurrency are featured in COOLL: “inter”- and “intra”-object concurrency; these two forms of concurrency have been studied in [7]. Inter-object concurrency is similar to independent AND-parallelism in parallel logic languages: each dataspace is independent from other dataspaces. Intra-object concurrency is similar to multiset rewriting in Gamma [8]: each object executes a program consisting of methods that are simultaneously active in the private dataspace.

Two forms of communications rule agents coordination: one-to-many (broadcasting), like in LO, and one-to-one (named). Named communication has a Linda “flavour”: it is associative and persistent. A message will be received when the receiver will be found in the agent system, even if it was absent when the message was sent.

We outline a COOLL solution of a problem of Dining Philosophers. The program consists of a single class: the table.

```
class table /</>:
    think-phil(1) @ fork @ fork /</> = "eat-phil(1) @" = "eat-phil(1) #
    eat-phil(1) @ = "end-phil(1) @" = "fork @ fork.
```

and an initial goal for three philosophers has the following form:

```
*table @ fork @ fork @ fork @ think-phil(0) @ think-phil(1) @ think-phil(2).
```

This example shows that COOLL is very similar to SP: the only difference with respect to the SP program is the presence of the *table atom, that denotes the theory to be used in this context.

However, the COOLL solution is much more flexible and adaptable inside an “open system”. In fact, suppose we need to define a whole restaurant of dining philosophers, with several tables and waiters that allocate philosophers to tables. Such a system is easily modeled and coordinated in COOLL, using one shared dataspace for each table and a new theory for describing what waiters do. For a complete solution, see [14].
6 Implementation issues

Several implementation strategies are possible for parallel logic languages based on the PoliS coordination model. The implementation strategy depends on the granularity of parallelism that one would like to exploit. Some possible scenarios are:

Centralized Coordination, Distributed Computation: The coordination rules are scheduled by a single process for all agents; such a scheduling process also manages the shared dataspace, whereas sequential computations are distributed and executed in parallel by a set of workers (see Fig. 3-left). The run-time support includes a rule scheduler which evaluates rules and manages the tuple spaces; the workers evaluate Prolog goals for agents, that are distributed and perform their internal derivation (local computation) in parallel. The implementations described in [2, 10] are all based on this approach. The target architecture for all these systems is a cluster of workstations.

Distributed Coordination: Agents are fully distributed; each agent is an autonomous worker including both the coordination rules and the Prolog program; the dataspace can be either centralized or distributed among workers (see Fig. 3-right). Agents perform both the local sequential computation and evaluate coordination rules asking tuples to be tested or consumed to other agents. In the implementation described in [13] tuples are produced only locally (similarly to Linda implementations where out is local and read-in are broadcasted).

Dataflow Coordination: In a fine-grained parallel scenario agents are evaluated in parallel and their rules are evaluated exploiting OR-parallelism with a dataflow strategy. More details are given in [13].

The first SP implementation was based on a main tuple manager process that managed both the tuple space and the evaluations of coordination rules, while Prolog goals were executed by separate worker processes residing in different workstations [2]. The number of workers was fixed, reflecting the number of parallel agents allowed in an initial goal. Low level communication was supported by a message-passing library of written in C under SUN-OS: it was based on a central mailbox and exploited socket communication.

The first implementation of multiple logic tuple spaces was described in [10]; its structure is depicted in Fig. 4. Each logic tuple space is implemented as a pair of Prolog processes: a tuple space manager and an agent manager; dynamic creation of both new tuple spaces and agents was allowed; support for communications among tuples spaces was based on a name server. Such an implementation currently includes a fully fledged programming environment called EXPO that is integrated with an X-window based interface that allows to visualize the contents of the tuple spaces [4].

In Fig. 5 the leftmost top window shows the current contents of the blackboard. A number of buttons allow the manipulation of the window. The leftmost bottom window is used to edit “shell patterns”, i.e., rules that are inserted and evaluated “on the fly” when the user is interested in some events that happens in this blackboard, typically when some tuples appear or disappear. The rightmost top window shows the history of user actions. The rightmost bottom window shows the name of the blackboard, and three buttons used to manipulate the windows.

Another ESP implementation is described in [18], and it is based on Network Linda for coordinating multiple tuple spaces. The run time system is a Linda program that implements a metatuple space that supports communication and tuple space creation.

All these implementations are based on interpretation, thus excluding any optimizations.
Figure 3. Architectures of two SP distributed interpreters: left) centralized coordination; right) distributed tuple space

Figure 4. Basic architecture of ESP implementation
obtained by compilation. However, this kind of languages offer several opportunities for compilation. For instance, in [13] optimization of rule distribution in Shared Prolog was addressed, and a compiler that could partially evaluate guards and communication flows was described. Such a compiler is targeted at a run-time system where agents execute theories in parallel and the blackboard is distributed among agents using replication techniques. The compiler uses static analysis techniques in order to minimize communications and conflicts among agents.

In [23] a different compilation approach is described which performs optimization of rule activation using indexing techniques on the tuple space. The compiler is based on a distributed runtime support with meta-level features written in C; both the scheduler and the indexing based on bitmap operations are implemented in C.

Possible issues for future research on improving implementations are:
- to analyze a dataflow implementation model, that is best suited to a massively parallel machine as a target;
- to apply the compilation techniques and indexing techniques described in [23] to a framework based on the compilation model described in [13];
- to use Linda for programming the coordination and creation of multiple tuple spaces; this seems to be a promising direction for improving load balance and integrating multiparadigm tuple spaces [18].

Figure 5. The ESP user interface
7 Comparison with other coordination models and languages

The family of languages we have described combines shared dataspace coordination with logic programming computation. They are extensions of Prolog in the same sense that C-Linda is an extension of C; they all add to Prolog coordination mechanisms based on PoliS. In comparison to Linda, the main differences are that PoliS is based on a precise coordination protocol expressed by logic rules.

The first suggestion concerning the use of shared dataspaces in parallel logic programming can be found in a paper presenting Polka, an extension of Parlog with the blackboard concept [19]. Shared Prolog was initially proposed in [9], where a comparison with Linda was given. Then PoliS [16] was proposed as a model for coordination based on multiple tuple spaces.

How can we compare PoliS with other logic languages and models? An interesting dimension for comparing parallel logic languages is the granularity of parallelism. Parallel logic programming systems may be classified in two different classes:

i. Fine-grain parallel logic languages evaluate implicitly parallel atomic goals. These languages include AND-parallel, OR-parallel and combined AND-OR-parallel languages. Examples in this category are the family of Concurrent Logic Programming languages [30] and Strand [21]. A new language that uses such a model, combining it with a constraint store, is OZ [27].

ii. Coarse-grain parallel logic languages evaluate multiple sequential programs, with some mechanisms being provided for inter-process communication [34]. In this class there are also two systems that were inspired by Linda: both Multi-Prolog [20] and Prolog-D-Linda [31] are Prolog extensions based on the Linda model. They are both different from SP because they do not impose any constraints on control of the communication primitives in the language: a dataspace operation can appear everywhere in a Prolog program.

SP has been implemented as a language of the latter class, augmenting Prolog with a number of Linda-like communications mechanisms. However, it could be considered for fine-grained parallel evaluation of coordination rules, using techniques developed in [13], where a compiler was described for SP whose output is a program optimized for a dataflow-like evaluation.

The advantage of SP with respect to the first class of languages is that programs can be easily built reusing existing Prolog programs, whereas the translation of Prolog to a stream-based language is difficult and moreover the resulting program loses its declarative reading. With respect to the second class, SP provides higher-level coordination primitives that improve the expressivity of the language, as shown in the distributed problem solving solution.

Other models and languages that have some affinity with PoliS and SP are LO, AbstrAct, Gamma, and Swarm.

For a discussion on the relationship between LO and SP, see [7].

AbstrAct is a logic programming language by Porto and Rosado [29]. In AbstrAct the blackboard coordination model is used: a set of tasks called agenda share a dataspace that is a set of logic terms. Non-monotonic update is supported. AbstrAct has not been implemented and it is still in evolution: the main concern of its designers is on semantic “cleanliness” of the model, that has to manage both classic logic programming and coordinated concurrency.

Gamma is a programming model based on multiset rewriting coordinated by the /BnZr operator [8]. In Gamma, the computation is a succession of non-deterministic applications of rules which consume elements of a multiset while producing new elements. Gamma programs
do not have any sequential structure and can be implemented easily on parallel machines. The main difference with respect to SP is that Gamma does not have a sequential program associated to rewriting rules. Gamma can be seen as an intermediate language between specifications and programs. High level specifications, for instance written using first order logic, can be translated into Gamma programs and then compiled into a parallel programs.

Another model that is based on a shared dataspace of tuples is Swarm [22]. However, in contrast with models which employ multisets the Swarm dataspace is not a multiset. The dataspace is partitioned into the tuple space, a set of ground tuples representing the working memory, and a transaction space, which corresponds to a set of rewriting rules. Transactions do not include deletion in the left hand side; moreover, it is acceptable for a transaction to delete an instantiated tuple that does not exists. SP has a different operational semantics for guard evaluation and commitment with respect to deleted tuples: the existence of tuples to be deleted is required before commit.

Swarm transactions are defined by introducing the type specifying the structure of the transaction; such a type may be simple or complex. The concept of transaction type presented in Swarm is similar to the concept of agent in SP, the main differences being: Swarm transactions are dynamic whereas SP agents are statically defined; the subtransactions of a Swarm complex transaction are applied all together whereas only one rule associated to a SP agent can commit in a single execution step.

Acknowledgments. Several people cooperated in developing the family of languages described in this paper. Here we want to thank especially S.Castellani, who is contributing to the development of COOLL.

References


