Design and Distributed Implementation of the Parallel Logic Language Shared Prolog

*V.Ambriola**+, *P.Ciancarini**, *M.Danelutto**

* *Dipartimento di Informatica - Università di Pisa*

+ *Dipartimento di Matematica e Informatica - Università di Udine*

Italy

ABSTRACT

The parallel logic language Shared Prolog embeds Prolog as its sequential component. A program in Shared Prolog is composed of a set of logic agents, i.e. Prolog programs, that communicate associatively via a shared workspace called blackboard.

The distinct features that characterize Shared Prolog with respect to other parallel (logic) languages are: scheduling of agents and granularity of parallelism explicitly controlled; Prolog embedded in the language; intrinsic modularity; associative send/receive of messages among agents with a simple operational semantics in terms of assert/retract of clauses. The model of communication puts Shared Prolog in the Linda family of parallel languages.

The current prototype implementation is exposed. The architecture of the system is a distributed network of extended Prolog interpreters running under Unix and communicating via Internet sockets, and was obtained refining a specification written in SP itself.

1. INTRODUCTION

Stream-based parallel logic languages like Flat Concurrent Prolog (FCP), Flat Guarded Horn Clauses (FGHC), and Parlog are not extensions of Prolog. These languages do not support programming in Prolog, apart from deterministic programs, that are immediately translated. In order to exploit the intrinsic parallelism, in general a Prolog program must be rewritten from scratch [Tick 89].

This paper describes the parallel logic language Shared Prolog (SP) that embeds Prolog as its sequential component, extending it with concurrency and associative communication. SP is based on the *blackboard* procedural interpretation of logic programming [Ciancarini 89], an interpretation different from the *process* procedural interpretation of logic programming, which is the basis of FCP, FGHC, and Parlog [Shapiro 87].

The distinct features that characterize SP with respect to other parallel logic languages are: unconstrained unification; scheduling of agents and granularity of parallelism esplicitly controlled; Prolog embedded in the language; intrinsic modularity; associative send/receive of messages with a simple operational semantics in terms of assert/retract of clauses [Brogi 89]; no streams; a simple interface to other linguistic paradigms.

The model of communication supported by SP is close to the Linda model of coordination [Gelernter 85]. Actually, SP could be considered a logic instance of the Linda family of languages [Ciancarini 90].

The intended use of SP is in applications in which it is natural and/or necessary to deal with "distributed intelligence". Two fields of application that have been developed with SP are distributed programming environments [Ciancarini 88, Ambriola 89], and expert systems based on a blackboard architecture managing multiple data streams [Brogi 89].

The paper has the following structure:

Section 2 contains a description of the language and a simple program; Section 3 shows how the language has been used in the design of its own programming environment; Section 4 describes the current implementation; Section 5 contains a comparison of SP with other parallel languages.

2. SP = LOGIC + LINDA'S IN/OUT

Given a database of clauses, the *computation* of a logic goal can be seen abstractly as a tree, where each node is labelled by a set of goals. The root is the starting goal, other nodes are labelled by derived tasks, and resolution is the inference rule.

A *procedural interpretation* is a specification of the goal evaluation. For instance, the procedural interpretation assumed in Prolog is sequential leftmost depth-first. Conversely, a parallel procedural interpretation specifies several evaluation threads. This general framework can be refined by introducing a set of control operators and extending or constraining unification. Different classes of logic languages exist, depending upon the choices about control and unification constraints.

Stream based parallel logic languages (FCP, FGHC, Parlog) rely upon the *process interpretation* of logic programming: a goal specifies a set of logic processes communicating by means of messages written in logic variables denoting partially defined shared data structures called *streams* [Shapiro 87, Ueda 87, Clark 86].

SP is a parallel logic language without shared logic variables for communication. Implicit global backtracking is not allowed, while associative communication is based on the blackboard interpretation. The key idea of the *blackboard interpretation* of logic programming is to interpret a goal as a set of logic agents exchanging messages in an associative way, via a common workspace close to Linda's Tuple Space. Associative send/receive of messages between agents have a simple and intuitive operational semantics in terms of Linda's associative in/out. They remind the metalogic assert/retract operations of standard sequential Prolog.

2.1 Syntax

A Shared Prolog program specifies a blackboard agent (a database of facts) and a set of parallel agents ruled by theories (i.e. Prolog programs activated if some conditions are satisfied within the blackboard).

A *blackboard* rule has a head without variables. The body is a bracketed set of atoms f_i that specifies the initial state of the blackboard.

```
blackboard :- {f1,...,fn}.
```

Atoms can contain variables (non-ground terms). The scope of a variable is limited to the atom in which the variable is introduced.

The head of a *theory* rule may contain variables. The body of a theory is a set of activation patterns separated by "♦", plus an optional Prolog program (following the keyword **with**) consisting of a set of clauses. Patterns and clauses of a theory are not visible from other theories.

```
theory(V1,...,Vj):-
  pattern1 ♦ ...♦ patternk
  with Prolog_program.
```

The variables introduced in the theory head are local to the theory, but global to all its patterns. These variables are useful to parametrize the theory, as shown in Section 2.3 (theory `tty(X)`).

The patterns specify the messages that activate the theory. A *pattern* has the following structure:

```
Read_Guard {In_Guard} | Body {Out}
```

where Read_Guard, In_Guard, Body, and Out_Set are set of atomic goals. The vertical bar is a symbol of *commitment*.

2.2 Semantics

An *initial goal* rule contains the name of a blackboard and a (possibly empty) set of theory invocations separated by "||". No variables can be shared between different theory invocations.

```
blackboard || theory1 || ... || theorym
```

The initial goal evaluation corresponds to an infinite computation of the specified set of processes. A final condition (a bracketed set of atoms) can be expressed using another form of initial goal rule.

```
blackboard || theory1 || ... || theorym :-
  {final_condition}.
```

When a final condition is specified, the initial goal evaluation corresponds to an attempt of proving the final condition by forward chaining.

The evaluation of the initial goal creates and activates as many parallel processes as are the theories. No backtracking takes place in the initial goal. The evaluation of a theory corresponds to the evaluation of its patterns, and it is totally independent from the evaluation of other theories.

The structure of a pattern is:

```
Preactivation | Postactivation
```

A preactivation is a guard (in the sense of Hoare's CSP) that must be satisfied before the theory commits. It includes two components:

- *Read_guard*: A set of conditions concerning the facts to be found on the blackboard before the theory activation. Positive conditions, negative conditions, or built-in predicates can be used in a Read_guard.
- *In_guard*: A set of facts to be found in the blackboard and consumed before the theory activation.

Read_guard and In_guard are evaluated by unification. If the preactivation of a pattern is satisfied, the theory commits to that pattern and the postactivation is executed. The postactivation is composed of two parts:

- *Body*: The initial goal of the Prolog program of the theory.
- *Out_set*: A set of facts that will be written on the blackboard at the end of the computation of the body.

The evaluation of the preactivation is an atomic action. As soon as a pattern is satisfied, the evaluation of the other patterns of the same theory is halted, and the theory commits to the body of the satisfied pattern. This starts a local Prolog computation whose final effect is to write the Out_set on the blackboard, and to restart the competition among patterns.

A running SP program consists of a set of independent processes which cooperate via the blackboard. The initial goal activates as many parallel processes as theories. There is no internal parallelism in a theory.

Another form of parallelism in Shared Prolog is OR-parallelism. The step of theory activation can be partitioned into two sub-steps: first the satisfaction of a pattern is checked and then, if it is satisfiable, the pattern succeeds. This way, a number of inactive theories may simultaneously try to satisfy their guards.

2.3 An Example

This section describes how to write in SP a system that manages a bank database containing customer accounts. Customers can connect to one of N tty's either to read their own balance, or to withdraw/deposit some amount of money.

The design of a corresponding SP program starts with the definition of the initial goal. This goal specifies one blackboard (an account database called `safe`), and a set of theories (two bank servers, and N tty's for the customers).

```
safe ||
bank_server(1) || bank_server(2) ||
tty(1) || ... || tty(n).
```

The account database initially contains two accounts only. This is stated by the following rule that defines the initial state of the blackboard.

```
safe:-
  {account(paul,1000),
  account(mark,1200)}.
```

The bank_server theory is shown in Figure 1.

```
bank_server(_):-
  Amount≤Balance
  {query(Tty,Name,withdraw(Amount)),
  account(Name,Balance)}
  |                            % pattern 1
  NewBalance is Balance-Amount
  {account(Name,NewBalance)}.
♦
  {query(Tty,Name,deposit(Amount)),
  account(Name,Balance)}
  |                            % pattern 2
  NewBalance is Balance+Amount
  {account(Name,NewBalance)}.
♦
  {query(Tty,Name,new(Amount))}
  |                            % pattern 3
  {account(Name,Amount)}.

  % no with part in this theory
```

Figure 1 *Bank_server Theory*

A `bank_server` theory can accept three different messages (corresponding to three theory patterns):

- `withdraw(Name,Amount)`
- `new(Name,Amount)`
- `deposit(Name,Amount)`

where Name identifies a customer, and Amount is a positive integer. To process a query, the bank_server needs to know the current balance. Since the balance will change after the query evaluation, the account tuple is consumed from the blackboard, together with the query message.

Let us see a pattern more closely:

```
Amount≤Balance
% Read_guard
{
query(Tty,Name,withdraw(Amount)),
account(Name,Balance)
}
% In_guard
|
NewBalance is Balance-Amount
% Body
{account(Name,NewBalance)}.
% Out_Set
```

If the blackboard contains, among the others, the facts

```
query(1,mark,withdraw(100)),
account(mark,1200)
```

the Read_guard is satisfied, the fact are consumed, and the pattern fires. The result of the evaluation of Postactivation is that the Out_set

```
account(mark,1100)
```

is written on the blackboard.

The program for the tty theory is shown in Figure 2.

```
tty(Tty):-
   not ready(tty(Tty)),
   not query(Tty,_,_)
   |         % (re)initialization
   {ready(tty(Tty)}.
♦
   {ready(tty(Tty)}
   |         % query to the system
   read(Name),read(Query),
   check(Query)
   {query(Tty, Name, Query)}.
♦
   account(Name,Balance)
   {query(Tty, Name, ask_balance)}
   |         % answer from the system
   write(Balance)
   {}.
with   % Prolog program
check(withdraw(Amount)).
check(deposit(Amount)).
check(new(Amount)).
check(ask-balance).
```

Figure 2 *Tty Theory*

A `tty` theory has an identification number (a local variable whose scope comprises all the patterns), and three patterns. The first pattern is for initialization, the second one for querying the system after receiving a command, and the third one for printing the balance after a user made the corresponding request.

This agent is a typical example of SP shell interacting with a user. The first pattern can fire only if the blackboard contains neither the atom `ready(tty(Tty))` nor the atom `query(Tty,_,_)`, i.e. it is used to enable the theory. The second pattern fires if the theory is enabled: input is asked to the user (a valid command). The third pattern fires if there is a pending "`ask_balance`" request. Note that the first pattern is mutually exclusive with the other two. Negative Read_guards (pattern 1) are evaluated on the blackboard under the Closed World Assumption.

3. DESIGNING THE SP MACHINE

SP has been used to specify and prototype some distributed AI applications [Ciancarini 88]. However, an interesting test for Shared Prolog was the specification of its own metainterpreter. Meta(circular) interpreters are good tests for programming languages, since they allow both to refine the language design and to compare the merits of different implementations.

For a parallel language, it is obviously interesting to explore parallel metainterpreters, both to test the language expressiveness and to exploit parallelism for building an efficient implementation on a multiprocessor architecture. In fact, the specification of one metainterpreter was used as guideline for a real distributed implementation. The actual prototype implementation of the distributed interpreter is described in Section 4.

3.1 Theory-grained Metainterpreter

There are many possible parallel metainterpreters for SP. Let us start showing the metainterpreter in which theories compute in parallel.

We must define a data structure to represent programs. A pattern is a tuple:

```
pattern(Read,In,Body,Out,T)
```

where T is the name of the theory the pattern belongs to. Each theory is represented by a set of `pattern` predicates.

To write a parallel meta-interpreter for SP, we need to specify the blackboard and the theories that execute the object program. The theory-grained parallel meta-interpreter keeps in the initial blackboard the facts of the object blackboard and all the patterns of the theories.

```
par_metaBB:-
  {P1,... , Pn,         % object bb facts
  pattern(Read11,In11,Body11,Out11,T1),
  ...,
  pattern(Read1m,In1m,Body1m,Out1m,T1),
  ...,
  pattern(Readk1,Ink1,Bodyk1,Outk1,Tk),
  ...,
  pattern(Readkn,Inkn,Bodykn,Outkn,Tk),
  }.
```

Object theories are handled by the following SP (meta) theory, whose parameters are the theory name T and the knowledge base KB:

```
par_Control_Theory(theory(T,KB)):-
  pattern(Read,In,Body,Out,T), Read
```

```
    |
    Body
    {Out}.
 with KB.
```

If the starting goal specifies a final condition, a special theory is devoted to check the related event on the blackboard, and to force termination for all the agents, including itself.

This metainterpreter suggests a specific architecture for the SP programming environment. We need as many processors as theories in the initial goal. The `par_Control_Theory` is replicated on each processor. Read_guards are interpreted as read-only accesses to the blackboard; In_guards are interpreted as messages from the blackboard to the theory; Out_sets are interpreted as messages from a theory to the blackboard. If there are as many processors as theories in the object system, the behavior is perfectly emulated. Notably, if the evaluation of a body does not terminate, the processor in charge of such a body loops. The blackboard can be either centralized or distributed: this metainterpreter does not make any such assumption.

3.2 Pattern-grained Metainterpreter

A more fine-grained parallel metainterpreter is obtained if there are as many processors as patterns. In this case the metablackboard is slightly different from the preceding one:

```
par_metaBB:-
  {P1,... , Pn,         % object bb facts
  pattern(Read11,In11,Body11,Out11,T1),
  ...,
  pattern(Read1m,In1m,Body1m,Out1m,T1),
  th(T1,KB),            % theory semaphore
  ...,
  pattern(Readk1,Ink1,Bodyk1,Outk1,Tk),
  ...,
  pattern(Readkn,Inkn,Bodykn,Outkn,Tk),
  th(Tk,KB),            % theory semaphore
  }.
```

The control theory replicated on each processor (one for each pattern) is the following:

```
par_Control_Pattern
  (pattern(Read,In,Body,Out,T)):-
  Read
  {In, th(T,KB)}
  |
  Body
  {Out, th(T,KB)}.
with KB.
```

When the evaluation of a pattern succeeds, the theory semaphore `th(T)` is retracted from the blackboard to prevent the activation of multiple instances of the same theory.

4. IMPLEMENTATION OF SP

The metainterpreters shown above specify a different granularity of parallelism for theories, without specifying blackboard control. This is a common situation for metainterpreters: some features are reified, i.e. completely specified and programmed, while other features are absorbed, i.e. directly assumed from the underlying machine, leaving room to many possible refinements for a given metainterpreter. Blackboard control, for instance, could be centralized or distributed.

We implemented the first prototype interpreter for SP as a Prolog (sequential) simulator following the guideline of a sequential metainterpreter [Brogi 89]. Afterwards we have implemented a parallel interpreter based on the theory-grained metainterpreter shown above. The architecture of the current system is a distributed network of extended Prolog interpreters running under Unix and communicating via Internet sockets.

The user view of the SP system is a Prolog shell that accepts a number of commands. Blackboard control is centralized. We were able to perform some compile-time optimizations. We are currently studying a distributed implementation for the blackboard.

In the following subsections we describe the system view and (shortly) the user view of the current implementation.

4.1 Operating Environment: System View

Standard Prolog implementations do not allow to start multiple, concurrent threads of execution within a given knowledge base. Therefore, we need the ability to activate concurrent threads of execution and, furthermore, a mechanism to exchange information between concurrent threads. These features are achieved as follows:

- The standard set of Prolog primitives was extended with new predicates supplying interprocess message passing primitives. True Prolog terms (including variables) can be exchanged as messages.
- A Prolog process was allowed, by an ad hoc mechanism, to activate a process corresponding to a Prolog interpreter executing a different program.

We have adopted *sockets* as the low level interprocess communication mechanism. We have built a mailbox server for interprocess and interprocessor communication.

An "open" version of the Prolog interpreter was used as implementation kernel. With "open" we mean a Prolog interpreter with a procedural attachment to a standard system language (e.g. the C programming language), and to the underlying operating system (e.g. Unix). This feature is now common in commercial Prolog systems.

4.1.1 SP Runtime System

The blackboard and each theory are separately evaluated by a group of Prolog language processors running control programs (the resulting processes are called *agents*). The agents of a SP system communicate via a kernel server, using asynchronous `send/3` and `receive/3` primitives, and a synchronous `sync_receive/3`.

An initial goal starts a "compilation phase" that creates the control programs for both the blackboard process and the theory processes.

Blackboard control is centralized in the blackboard agent. Its knowledge base consists of:

- The blackboard name, represented by the fact `bb(Name)`.
- The blackboard initial content, directly represented as a set of facts.
- The list of the names of all the theories: `theories(List)`.
- The set of patterns of the theories, represented as follows:

```
pattern
(Theory,Nofpattern,Read,In,Body,Out).
```

- The state of a theory: the fact that a *theory is active on a pattern* as:

```
active(Theory,Nofpattern,Body,Out).
```

- Control program: the clauses that define the behavior of the blackboard agent.

```
control_bb:-
  % final condition (if any)
  eval(Final_condition),
  terminate(Theories),
  halt.
control_bb:-
  % evaluation loop
  schedule(Activable_patterns),
  % choose patterns
  send_all(Activable_patterns),
  % send messages
  all_receive(Answers),
  % wait at least 1 answer
  assert_all(Answers),
  % update object bb
  control_bb.
```

The scheduling of activable patterns (given the actual blackboard) is a possible bottleneck. Currently, we are developing partial evaluation techniques that should speed up this activity.

A theory agent has an independent knowledge base. The theory agent knows its name and the name of the blackboard to which it is connected, and executes a control program. The following clauses define the control behavior of a single theory agent:

```
control_th(Theory,BBname):-
  sync_receive(Theory, BBname, Body),
  eval(Theory, Body, BBname),
  control_th(Theory,BBname).

eval(T, Body, BB):-
  call(Body), !, send(T,BB,Body).
eval(T, Body, BB):-
  send(T,BB,failure).
```

A theory agent receives a body, i.e. a Prolog goal to evaluate. If the goal evaluation terminates, it answers

- The goal modified with the resulting variable substitutions, in case of success.
- The atom `failure`, in case of failure.

4.1.2 The Kernel Architecture

In this section we present the kernel mechanisms implemented to run Shared Prolog programs on a network of Unix machines.

The implementation suggested by the theory-grained metainterpreter shown in Section 3 is based on a network of Prolog processes: each process is a Prolog program running either a blackboard manager or a theory manager program. This implementation relies upon an extended Prolog kernel able to perform the following tasks:

- To "fork" a number of different threads from within a single Prolog program.
- To make available some communication mechanisms for the communication of full Prolog terms among Prolog processes (running on the same machine or on a network of machines).

Given such features, a Shared Prolog program is implemented according to the following architecture (see Figure 3):

- A Prolog process for the blackboard control program is activated.
- For each theory, a Prolog process for the theory control program is activated.
- Communications between the theory agents and the blackboard agent are handled by the kernel. For instance, a theory can issue a request for the evaluation of a pattern as a list of Prolog facts that must be matched in the blackboard. Afterwards, the blackboard answers to the theory once more with a list of Prolog facts.

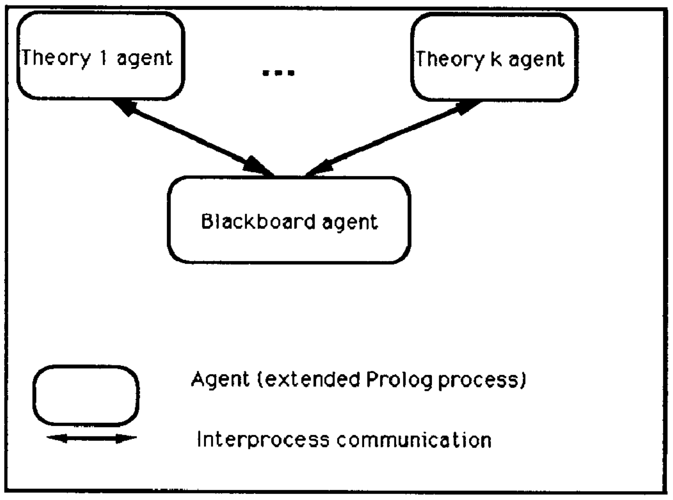

Figure 3 *Current implementation of the SP interpreter*

In order to fulfill the requirements imposed by the distributed execution of SP programs, an interprocess communication mechanism has to be supplied. In the current implementation we used the Unix BSD internet socket facility. Sockets allow fast bidirectional communications between any pair of Unix processes running on the same machine or on different machines in an Internet network.

Thus, we extended a Prolog interpreter with some new predicates for interfacing the socket mechanism. We wrote a C program performing the required tasks, and then we linked the compiled program to the addressing space of the Prolog intepreter (a standard feature of many Prolog language processors). The new predicates allow to associatively send (receive) a message consisting of a generic Prolog term to (from) another Prolog interpreter.

The syntax of the new predicates is the following:

```
send(<snd>,<rcv>,<msg>)
```

```
receive(<rcv>,<sndV>,<msgV>)
```

where the arguments can be constants, variables, or terms. The informal semantics of these predicates is the following:

`send(myname,destname,msgvalue)`

dispatches a message (*msgvalue*) from a Prolog process (identified by *myname*) to a Prolog process identified by *destname*.

`receive(myname,mittname,MsgVar)`

unifies the term *MsgVar* with a message sent by process *mittname* to process *myname* if such a send has ever been performed or with the term "`no_msg_for_you`" if such a send has never been performed.

`receive(myname,MittnameVar,MsgVar)`

unifies the term *MittnameVar* with the name of the process that performed a call `send(_,myname,msgvalue)`, and the term *MsgVar* with *msgvalue* if such a send has been performed or with the term "`no_msg_for_you`" if such a send has never been performed.

Both `send` and `receive` primitives are asynchronous. This means that they always immediately succeed upon call. As standard Prolog system predicates, they are not backtrackable, i.e. once they have been executed, they cannot be "undone".

When a synchronous receive is needed, the following (active wait) code may be used:

```
sync_receive(Myname,Mittname,Msg):-
  receive(Myname,Mittname,Msg),
  Msg\=="no_msg_for_you",!.

sync_receive(Myname,Mittname,Msg):-
  sync_receive(Myname,Mittname,Msg).
```

The implementation of `send` and `receive` rely upon a global communication server process that accepts and dispatches communication requests. The server accepts requests issued by the C procedures that implement the Prolog predicates `send` and `receive`. The server manages a mailbox and a waiting list as follows:

1. If the server receives a communication request

   `send(mittname,destname,message)`

it simply puts the communication request in the mailbox.

2. If the server receives a communication request

   `receive(destname,Mittvar,MsgVar)`

it looks up the mailbox data structure for a message sent to the *destname* process and it answers back to the requesting process either the first message found in the mailbox (deleting the message from the data structure) or a "`no_msg_for_you`" message with *MittVar* bound to "`communication_server`".

3. If the server receives a communication request

   `receive(destname,mittname,MsgVar)`

it looks up the mailbox data structure for a message sent to the *destname* process from a *mittname* process and then proceeds as in point 2.

The (informal) server algorithm is depicted in Figure 4.

```
server()
{
  initialize server data structures;
  create the global socket server;
  accept requests on this socket;
  termination=FALSE
  while(!(termination))
    {
    read a request;
    case(request_type)of
      {
      SND: if
        the destination process is
        waiting for a message from
        this sender
        then
        send the incoming message to
        the waiting process as an ACK
        message (this unblocks the
        receiver process)
        else
        put the incoming message in
        the mailbox structure
        endif;
        send the ACK message
        to the sender process;
      RCV: if
        there is a message in the
        mailbox with the requested
        sender for the
        requesting process,
        then
        send it back as ACK message
        else
        put the requesting process
        in the wait list
        endif;
      END:   termination=TRUE;
      }
    }
  close open sockets;
  clear dynamic data structures;
}
```

Figure 4 *The server algorithm*

```
send()
{
  get parameters
  (mittname,destname,msg);
  connect a socket to the
  communication server socket;
  create a reply private socket;
  send the SND message with the proper
  parameters on the server socket;
  wait for the ACK message on the
  private reply socket;
}

receive()
{
  get parameters
  (destname,Mittname,Msg);
  connect a socket to the
  communication server socket;
  create a reply private socket;
  send the RCV message with the proper
  parameters on the server socket;
  wait for the ACK message containing
  the message on
  the private reply socket;
  bind the Msg variable;
  possibly bind the Mittname variable;
}
```

Figure 5 *The send and receive algorithms*

The algorithms of the associative send and receive predicates are depicted in Figure 5.

The overall resulting structure of the SP distributed run time system is depicted in Figure 6.

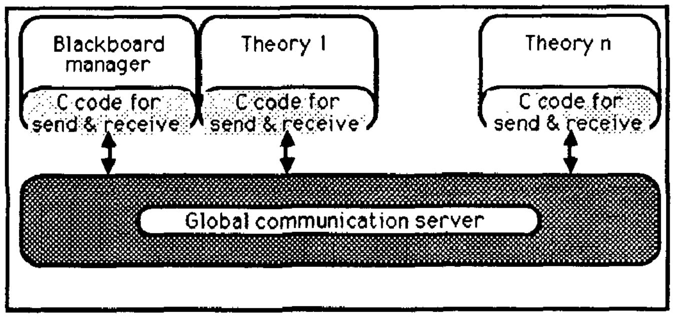
Blackboard manager
Theory I
Theory n
C code for send & receive
C code for send & receive
C code for send & receive
Global communication server

Figure 6 *SP Distributed Runtime System*

A Prolog interpreter can fork the execution of a number of different threads using a `shell` system predicate. When a thread has to be forked, the following steps are performed:

- The Prolog code for the new thread is put in a special file `prolog.ini`.
- A shell predicate

`shell(Prolog_interpreter)`

is executed, where `Prolog_interpreter` is a string denoting the name of a Prolog interpreter. This forces the activation of a new Prolog process executing the code contained in the file `prolog.ini`.

Using this simple mechanism, multiple threads can be forked at any time. For instance, new theories can be dynamically activated and connected to a running blackboard.

4.2 Operating Environment: User View

Currently, the user enters in a Prolog environment and activates the SP run time system. The user interacts with a shell through which he can do the following actions:

1) Give an initial goal that activates a blackboard process and (possibly) a set of agents.

2) Give a connection goal that connects (or disconnects) agents to a running blackboard process. The blackboard process could belong to another user. The syntax of these special goals is:

```
Theory@Blackboard
```

To connect a theory to a blackboard.

```
{Theory@Blackboard}
```

To disconnect a theory from a blackboard.

For instance, in the bank account example a new tty is added with the command

```
tty(n+1)@safe.
```

3) Trace the messages exchanged between a theory and its blackboard.

There is also a tool for browsing within a running blackboard to observe its evolution.

5. COMPARISON WITH OTHER LANGUAGES

We know of at least another blackboard logic language: Polka [Davison 87]. While its field of applications is quite similar to the SP one (i.e. distributed AI), its implementation and underlying philosophy is quite different, since it has been built as an extension of the Parlog programming environment. In this way Polka inherits all the strong points, but also all the weaknesses of the traditional parallel logic approach.

A more interesting comparison can be attempted with respect to Linda [Carriero 89a]. Linda's "coordination model" has been proposed as a framework for designing and programming open parallel systems, i.e. systems composed of a dynamic collection of asynchronous communicating agents [Carriero 89b].

It is interesting to match the Linda model with the parallel logic programming paradigm. Admittedly, a logic language (e.g. Flat Concurrent Prolog) misses the efficiency and the openendness of Linda when specifying parallel systems, with a slight gain in expressiveness. Moreover, it seems difficult to elegantly integrate different languages in FCP or GHC: these languages have problems even in accomodating Prolog!

We consider SP as the missing link between the family of Linda languages and the family of parallel logic languages. SP is closer to the Linda coordination model, while it maintains a strong logic flavour (so it should be more suited for specification tasks). In fact, the blackboard in Shared Prolog is very similar to Linda's tuple space, and with some approximations SP could be defined as a logic programming counterpart of the Linda framework. We found (and we are currently studying) that the SP approach allows a number of compilation-time optimizations, and moreover maintains a great flexibility at run-time.

6. CONCLUSIONS AND FUTURE WORKS

Current research on concurrent logic programming is centered around two very different approaches.

In the first one, the focus is on parallelizing compilers, trying to get advantage from modern multiprocessor architectures in a transparent way, from the point view of the Prolog user [Hermenegildo 89].

In the second one, new logic languages are defined, like FCP, FGHC or Parlog, that are very different from Prolog and are based on fine grained parallel computing models.

Our approach has taken a different route: we have embedded standard Prolog in a distributed operating environment, aiming at having an open system of large-grained, communicating Prolog agents. Not only strings can be exchanged as messages, but also true Prolog terms (also non ground, i.e. including variables). The programmer has full control on the scheduling of processes and the granularity of communications.

An interesting topic that we are going to explore is the relationship between SP and the Linda family of languages. We think that SP can be considered the logic counterpart of the Linda approach. Moreover, we plan to develop a new implementation using a Linda based kernel for the blackboard control, that in this way will become completely distributed.

ACKNOWLEDGEMENTS

This work improved after many discussions with A.Brogi, A.Corradini, T.Castagnetti, M.Montanari. The authors were partially supported by CNR Progetto Finalizzato Informatica - Calcolo Parallelo.

REFERENCES

[Ambriola 89] V.Ambriola, P.Ciancarini, A.Corradini, M.Danelutto, "SHELL: a Shell Hierarchical Environment based on a Logic Language", TR.30-89, Dip. di Informatica, Università di Pisa, 1989.

[Brogi 89] A.Brogi, P.Ciancarini, "The Concurrent Language Shared Prolog", TR.11-89, Dip. di Informatica, Università di Pisa, 1989, pp.24

[Carriero 89a] N.Carriero, D.Gelernter, "Linda in Context", *CACM* 32:4, 1989, 444-458.

[Carriero 89b] N.Carriero, D.Gelernter, "Coordination Languages and Their Significance", DCS TR.RR716, Yale University, 1989.

[Ciancarini 88] P.Ciancarini, *Specifying and Prototyping Software Engineering Environments*, PhD Thesis (in Italian), Università di Pisa, 1988.

[Ciancarini 89] P.Ciancarini, "Blackboard Programming in Shared Prolog", in D.Gelernter, A.Nicolau, D.Padua, (eds.), Procs. 2nd Workshop on Parallel Languages and Compilers, in the series *Research Monograph in Parallel and Distributed Computing*, Pitman, 1989.

[Ciancarini 90] P.Ciancarini, "Coordination Languages for Open System Programming", Proc. IEEE Int. Conf. on Programming Languages, New Orleans, 1990.

[Clark 86] K.Clark, S.Gregory, "Parlog: Parallel Programming in Logic", *ACM Trans. on Progr. Lang. and Systems*, 8, 1986, 1-49

[Gelernter 85] D.Gelernter, "Generative Communication in Linda", *ACM Trans. on Progr. Lang. and Systems*, 7:1, 1985.

[Gelernter 89] D.Gelernter, "Multiple Tuple Space in Linda", Proc. PARLE, 1989.

[Hermenegildo 89] M.Hermenegildo, "High Performance Prolog Implementations", Tutorial, Int. Conf. on Logic Programming, Lisboa, 1989.

[Shapiro 87] E.Shapiro (ed.), *Concurrent Prolog: Collected Papers*, vol. 1 and 2, MIT Press, 1987.

[Shapiro 89] E.Shapiro, "Embedding Linda and Other Joys of Concurrent Logic Programming", TR.CS89-07, The Weizmann Institute, 1988, pp. 11.

[Takeuchi 86] A.Takeuchi, K.Furukawa, "Parallel Logic Programming Languages", Proc. 3rd Int. Logic Programming Conf., London, 1986, 242-264 (also in [Shapiro 87]).

[Tick 89] E.Tick, "Comparing Two Parallel Logic Programming Architectures", *IEEE Software*, July 1989, 71-80.

[Ueda 86] K.Ueda, "Guarded Horn Clauses", in Logic Programming '85, *LNCS 221*, Springer Verlag, 1986, 168-179 (also in [Shapiro 87]).