Blackboard Communication in Prolog

Koenraad O.M. De Bosschere*
Electronics Laboratory
State University of Ghent, Belgium
kdb@lem.rug.ac.be

Abstract

This paper describes a formal model for parallel programming languages, based on macroscopical coarse grained parallelism and shared blackboard communication. Examples of such languages are Multi-Prolog [3, 4], and Shared Prolog [1]. Both languages support a blackboard which acts as a common communication medium between processes. The blackboard communication is explicit, by means of dedicated communication primitives. The parallelism exploited is not based on the properties of the logic (and-, or-parallelism), but on the visible parallelism in the application. A program consists of a number of communicating sequential (Prolog) processes [2].

1 Introduction

A formal model is created in two steps. In a first step, the semantics of Horn Clause logic is generalized to the semantics of *Communicating Horn Clause logic*, i.e., Horn clauses extended with two communication primitives, one to send information to and one to receive information from a blackboard.

The declarative semantics is a generalization of the model-theoretic semantics of Horn clauses. An associated operational semantics is defined and proved to be sound and complete. This shows that the theory of Communicating Horn Clauses, although it is not side-effect free (blackboard communication is clearly a side-effect), is not in conflict with the logic programming paradigm.

In a second step, a number of Communicating Horn Clause processes are put around the *blackboard* and can start cooperating to solve a goal. In this model, we look at processes as sets of traces, because the complete blackboard input/output behavior of a process can be described by means of the set of traces it can generate. A trace now becomes a blackboard transforming function.

Declaratively, a goal is true iff a trace t_1 associated with the goal, and a trace t_2 generated by the other processes, can be combined in such a way that one of the resulting traces can be executed successfully on the current blackboard. This is an example of a *may-semantics* [6]. This means that there must be 'a way to do it', but not every way must be successful, although it might.

*Research Assistant with the Belgian National Fund for Scientific Research.

An operational semantics that covers the declarative semantics is also defined. However, at this point, due to the may-semantics, two selection functions must be introduced: one to decide the order in which processes are going to communicate (process selection function to solve the inter-process non-determinism), and one to decide which process trace is going to be used next (trace selection function to solve the intra-process non-determinism).

2 Related work

This work is related to both Distributed Logic [9] and Linda [5]. At first sight, it seems to be a weakened version of Distributed Logic. However, a closer look reveals quite the contrary. In Distributed Logic, the creation and termination of processes have a *logical* meaning. The concurrent execution of a conjunction of two goals is true iff both goals can be proved true. In contrast, the creation and termination of processes in communicating Horn Clauses do *not* have a logical meaning. They are extra-logical.

Moreover, the communication is open, rather than channel-based. That means that a message is broadcast to the blackboard, where precisely one process can pick it up and produce an answer. Messages may even stay there for a while before they are picked up. So, the communication in Communicating Horn Clauses is asynchronous, in contrast to Distributed Logic where it is synchronous.

The resemblance with Linda is, to say the least, quite strong. The blackboard is an implementation of Linda's tuple space. The communicated ground terms correspond to Linda's tuples. The use of terms is appropriate for a logic programming language, because terms are a natural way to represent information, and unification is a built-in mechanism to do the pattern matching on the tuples.

Communicating Horn Clauses could also be seen as a model for the language Shared Prolog [1]. An application in this language also consists of a number of sequential processes (called 'theories') which cooperate to execute a task by means of activation patterns. However, the communication in Shared Prolog is more structured (preactivation and postactivation parts). Our model makes fewer assumptions on the program structure, and is therefore more general.

3 Syntax and Informal Semantics

The syntax of Communicating Horn Clause logic is very similar to the syntax of traditional Horn Clause logic. The alphabet consists of connectives, punctuation symbols, functors, predicate symbols, variables, and communication modes.

So as to express parallelism and communication, the language must have at least two additional operators: one to create processes, and one to communicate between processes. At this point, we consider the creation of processes as a side effect without a direct logical meaning; for the time being, it is considered extra-logical. We introduce two communication primitives: one to send information to the the shared blackboard, and one to receive information from it.

The introduction of communication operators introduces sequentiality in the underlying logic model. That is, the truth value of a predicate is not necessarily a timeless

function. It may depend on the current state of the blackboard, which might have been affected by earlier communication operations.

The dependence on a blackboard state implies that the declarative and refutation semantics of Communicating Horn Clauses must anew be proved equivalent [7].

There are two communication modes: ! to write to, and ? to read from the blackboard. Communication (blackboard) goals are built from the communication modes and terms. The expressions `!john` and `?mary` are examples of communication goals. The goal `!john` will write the term `john` to the blackboard, while `?mary` will search for a term `mary` in the blackboard, and remove it as soon as such a term is found. Operationally, the requesting process will block until an answer is produced. A communication goal will generate communication events, or simply *events.*

At this point, it is necessary to introduce some notation and auxiliary concepts.

Definition 3.1 A *formula* is an atom, a blackboard goal, or a conjunction of formulas.

Definition 3.2 A *blackboard communication goal* is of the form $?\tau_1$, or $!\tau_2$, where τ_1 and τ_2 are ground terms.

At run time, the $!\tau$ goal can only put ground terms onto the blackboard. This is a consequence of the fact that processes of Communicating Horn Clauses are considered as independent agents. The communication of non-ground terms would be in conflict with this principle because a non-ground term actually creates a direct connection between two processes. Hence, streams as they exist in other parallel logic programming languages do not exist in Multi-Prolog.

Definition 3.3 The *basic clauses* of Communicating Horn Clauses are:

$\mathtt{f} \leftarrow$
$\mathtt{f} \leftarrow \mathtt{g}_1 \ , \ \mathtt{g}_2$
$\mathtt{f} \leftarrow \mathtt{e}$

where $\mathtt{e}$ is a blackboard communication goal ($!\tau$ or $?\tau$), $\mathtt{f}$ an atom, $\mathtt{g}_1$ and $\mathtt{g}_2$ are formulas.

To ease our task in proving theorems, we use the following lemma.

Lemma 3.4 (structural induction) *Every set of clauses can be rewritten to a form using only basic clauses while preserving unsatisfiability.*

4 Traces generated by a Horn Clause process

The semantics of Horn Clause logic only depends on the set of clauses, not on the selection rule used during resolution. However, the introduction of communication events in Horn Clause logic creates the notion of time, which affects the semantics. In our model, time is formalized by means of a *trace of communication events.* Since a trace is an ordered collection, it imposes an ordering on the clauses generating it, and also on the selection function used to implement resolution. The selection function we shall assume in the sequel is left to right. The most important consequence of this restriction is that Communicating Horn Clause logic is semantically not equivalent with traditional Horn clause logic. It resembles more the semantics of Prolog.

Although the loss of commutativity, which is one of its consequences, may seem a severe restriction, this is not the case. It only shows that the model is more closely related to Prolog, which evaluates left to right, and in which the conjunction does not commute either. Our model can be used to study the behavior of Prolog more carefully than is possible with Horn Clause logic.

The external behavior of a set of Communicating Horn Clauses can be characterized by the set of traces it can generate. For example, the trace of a process that first writes the term a onto the blackboard, and afterwards reads b from the blackboard is <!a,?b>. This trace completely describes the external behavior of the process. We shall now formalize this notion.

Definition 4.1 The set of *communication events* H_E on a set H_U is given by $H_E = \{!,?\} \times H_U$

Definition 4.2 A *trace* t is a finite sequence of communication events. The set of all traces $(H_E)^*$ is denoted by H_T [2, 10].

For example, for the Herbrand universe $H_U = \{a,b,c\}$, the set of Herbrand events H_E is given by {!a,?a,!b,?b,!c,?c}. An example of a trace on H_E is given by <!a,?b,!b>. The order of events in a trace is significant, and must not be changed.

In the sections that follow, we study the behavior of one single sequential communicating process. Such a process interacts with a trace by means of its communication goals. We first discuss the modified concepts of interpretation and model. Afterwards we discuss the formal semantics.

5 Interpretations and Models for Communicating Horn Clauses

In traditional Horn Clause logic, an interpretation for a set of clauses can be described as a subset of the Herbrand base containing the ground atoms that are taken to be true [8]. Unfortunately, in Communicating Horn Clauses, such a subset of the Herbrand base is no longer sufficient to make a communication goal true. The truth value of a goal depends on the state of the blackboard, that is, the cumulative effect of all previous communication operations. Therefore, an interpretation must also depend on that blackboard. We shall use traces for this purpose. Atoms are now considered true in the presence of some trace, and false in the presence of others. A Herbrand interpretation I is therefore a subset of $H_T \times H_B$.

A Herbrand interpretation I can be structured *atom-wise*. For every element a of I, let I[a] be the set of traces that make a true, i.e., $I[a] = \{t \in H_T : \quad (t,a) \in I\}$.

Definition 5.1 Let a be a ground atom, and let t a trace. Then, the expression $t \models_I a$ means that, under the interpretation I and in the presence of trace t, the formula a is true, i.e., $(t,a) \in I$.

In the sequel we shall always use the notation $t \models_I f$ because of its convenience. The set-interpretation will only be used in proofs.

Notice that, to make a true, the trace t must be completely consumed. If the atom a is true by using a strict prefix of t, $t \models_I a$ is not true. The expression $<> \models_I a$ means that

the truth of the atom a only depends on the program, not on the trace. It is trivially the case that $\langle\rangle \models_I \mathtt{true}$.

It is useful to extend the definition of $\models$ to non-atomic expressions in the following way:

1. by $(t \models_I g,h)$ we mean $(t_1 \models_I g \wedge t_2 \models_I h \wedge t = t_1 + t_2)$, for g and h ground formulas;
2. by $(t \models_I a \leftarrow f)$ we mean $(t \models_I f) \rightarrow (t \models_I a)$, for a and f ground formulas;
3. by $(t \models_I \leftarrow f)$ we mean $\neg\, (t \models_I f)$, for f a ground formula;
4. by $(t \models_I e)$ with e a ground communication event, we mean that t unifies with $\langle e \rangle$.

In case the formulas are not ground, the above expressions must hold for all ground instances of the formulas.

So far, we have only considered isolated traces. However, a process is modeled by a *set* of traces. This set of traces describes all possible blackboard communication operations a process can ever perform. It is useful to extend the definition of $\models$ to a set of traces T: $(T \models_I f)$ iff $\exists t \in T\colon (t \models_I f)$.

Let us now turn to the concept of a *model* of a set of clauses S. We say that an interpretation I is a model iff every clause of S is true under I. To make clear distinction between a classical model (subset of the Herbrand base), and our model, based on traces and the Herbrand base, we shall call our model a *generalized model.*

6 Declarative Semantics

The *declarative semantics* of a set of clauses S is the least generalized Herbrand model of S.

In this section, we use a fixpoint argument to characterize the least general model of a set of clauses S. As stated above, an interpretation for a set of clauses S is a subset of $H_T \times H_B$. Let $\mathcal{I}$ be the set of all interpretations for S. With S we associate an operator $\mathcal{S}\colon \mathcal{I} \mapsto \mathcal{I}$ (the least consequences operator). Given an interpretation I, the set $\mathcal{S}(I)$ is constructed as follows:

1. let $a \leftarrow$ be a ground instance of a fact $a' \leftarrow$ of S. Then, it follows that $(\langle\rangle, a) \in \mathcal{S}(I)$;
2. let $a \leftarrow g_1, g_2$ be a ground instance of a clause $a' \leftarrow g_1', g_2'$ of S. If $t_1 \models_I g_1$ and $t_2 \models_I g_2$, then $(t_1 + t_2, a) \in \mathcal{S}(I)$;
3. if $t \models_I e$ (that is, $t = \langle e \rangle$), and S contains a clause $a' \leftarrow e'$ of which $a \leftarrow e$ is a ground instance, then $(t, a) \in \mathcal{S}(I)$.

The above conditions imply that $I \subseteq \mathcal{S}(I)$; obviously, $\mathcal{S}(I)$ is also an interpretation. More importantly, the construction of $\mathcal{S}(I)$ suggests a way to construct a generalized model for S, as indicated by the following theorem.

Theorem 6.1 *An interpretation* I *is a generalized model for a set of clauses* S *iff* I *is a fixpoint of* $\mathcal{S}$.

Example 6.2 The program S

```
p(a)←
q←p(X),!X
r←?X, X=a
X=X←
```

is rewritten as a set of basic clauses S′

```
p(a)←
q←p(X), x′(X)
r←x″(X), X=a
X=X←
x′(Y)←!Y
x″(Z)←?Z
```

We choose the empty set $\emptyset$ as initial interpretation.

$I_0 = \emptyset$
$I_1 = \{(<>,p(a)), (<!a>,x'(a)), (<?a>,x''(a)), (<>,a=a)\}$
$I_2 = \{(<>,p(a)), (<!a>,x'(a)), (<?a>,x''(a)), (<>,a=a),$
$(<!a>,q), (<?a>,r)\}$
$I_3 = I_2 = M$
...

The restriction to the Herbrand base of the initial set of clauses gives

$M = \{(<>,p(a)), (<!a>,q), (<?a>,r), (<>,a=a)\}$

7 Refutation Semantics

The refutation semantics is a formal model for the operation of a derivation procedure. The set $S \cup \{\leftarrow f\}$ is unsatisfiable iff $\leftarrow f$ is false in all generalized models of S, and therefore in the minimal generalized model. The fact that $\leftarrow f$ is false in the minimal generalized model of S is equivalent to the fact that there is at least one instance $\leftarrow f_0$ of $\leftarrow f$ which is false in the minimal generalized model of S. That means that there is at least one f_0 which derives true for the set S. The derivation procedure is purely syntactical.

We start with a formalization of the derivation procedure, which is modeled as a rewriting system. Let S be a set of basic clauses and b be an atom. The rewriting rules for the basic clauses listed below must be understood as follows: assume that the atom b is unified with the head of the clause f, using the unifier θ. Then, the atom b is replaced by the right hand side of the clause (or simply omitted), and the substitution θ is applied to the remaining formula. For the blackboard goals, an event will be consumed as well. In summary, we have:

clause	rewriting rule
$f\leftarrow$	$b,g:<>,\theta \rightarrow_s g\theta$
$f\leftarrow g_1,g_2$	$b,g:<>,\theta \rightarrow_s (g_1,g_2,g)\theta$
$f\leftarrow e'$	$b,g:<e>,\theta \rightarrow_s g\theta$

For the clause f←e', the substitution θ consists of two parts: θ_1, resulting from the unification of b with f, and θ_2, resulting from the unification of e' with e. A goal consisting of one atom b can be rewritten by replacing the goal by b,true.

Definition 7.1 We say that in the set of clauses S, g derives h directly with trace t and substitution θ, denoted as, $g{:}t,\theta \rightarrow_S h$, if, by using the trace t, the substitution θ, and one of the rewriting rules mentioned above, g can be rewritten into the formula h.

Definition 7.2 We say that, in a set of clauses S, the formula g *derives* h with trace t and substitution θ, $g{:}t,\theta \Rightarrow_S h$, if there exists a finite sequence of *direct derivations*

$$g{=}g_0{:}t_1,\theta_1 \rightarrow_S g_1{:}t_2,\theta_2 \rightarrow_S \ldots \rightarrow_S g_{n-1}{:}t_n,\theta_n \rightarrow_S g_n{=}\ h$$

such that $t = t_1{+}\ldots{+}t_n$ and $\theta = \theta_1 \ldots \theta_n$. If g is equal to true, the derivation is said to be successful. The rule that a number of direct derivations can be combined into one derivation is known as the *transitivity rule*. The concept of direct derivation allows us to give a precise definition of refutation semantics of a formula f.

Definition 7.3 The *refutation semantics* of a set of clauses S is the set of all ground pairs (t,a) of traces and atoms for which there exists a successful derivation $a{:}t,\{\} \Rightarrow_S true$.

Example 7.4 For the program consisting of the basic clauses S' of example 7.2 it is straightforward to produce a successful derivation for q

$$\begin{aligned} q = q,true{:}<>,\{\} &\rightarrow_{S'} p(X),x'(X),true{:}<>,\{X/a\} \\ &\rightarrow_{S'} (x'(X),true)\{X/a\} = x'(a),true{:}<!a>,\{Y/a\} \\ &\rightarrow_{S'} true \end{aligned}$$

or, as one derivation $q{:}<!a>,\{X/a,Y/a\} \Rightarrow_{S'} true$. Since q is a ground atom, the derivation can be rewritten as $q{:}<!a>,\{\} \Rightarrow_{S'} true$.

Similar derivations can be produced for the other atoms of the Herbrand base.

$r{:}<?a>,\{\} \Rightarrow_{S'} true$
$a{=}a{:}<>,\{\} \Rightarrow_{S'} true$
$p(a){:}<>\{\} \Rightarrow_{S'} true$

The refutation semantics is given by {(<>,p(a)), (<!a>,q), (<?a>,r), (<>,a=a)}.

8 Soundness and Completeness

To prove that the declarative semantics and the refutation semantics are equivalent, we must show that the refutation semantics is *sound* and *complete*. The soundness property means that *no wrong* results are derived by the refutation algorithm, while the completeness property indicates that *all* solutions are generated. The proofs are not given here.

Theorem 8.1 (Soundness) *If* $f{:}t,\theta \Rightarrow_S true$ *for a set of clauses* S, *then* $S \cup \{\leftarrow f\}$ *is unsatisfiable, i.e.,* $t \models_M f_0$ *for at least one ground instance* f_0 *of* f.

Theorem 8.2 (Completeness) *If* $S \cup \{\leftarrow f\}$ *is unsatisfiable, and if* $t \models_M f_0$, *for some trace* t, *and some instance* f_0 *of* f, *then* $f{:}t,\theta \Rightarrow_S true$, *where* θ *is a substitution such that* f_0 *is an instance of* $f\theta$.

9 Communicating Horn Clauses connected to a Blackboard

Now, we change our point of view. We no longer consider the 'internal' semantics of a process, and turn our attention to the 'external' semantics, i.e., the input/output behavior of a process. This behavior is described by a set of traces. It describes the 'reaction' of a process on an input from the blackboard. The set {<?a,!c>,<?b,!c>} tells us that the event !c will be generated after the event ?a or the event ?b has occurred. In practice, a number of processes will execute simultaneously. We shall describe the behavior of a number of concurrently running processes in the same way as one single process, i.e., by means of a set of traces.

Example 9.1 Consider the program

```
p1←?a,!b
p2←?b,!c.
```

We create two processes by spawning p_1 and p_2. The set of traces from p_1 and p_2 is given by {<?a,!b>} and {<?b,!c>}, respectively. As both processes are running independently, their events may occur in any order, as long as the event order per process remains the same. The set of traces generated by the concurrent operation of p_1 and p_2 is given by the *interleaving* $\oplus$ of the traces for p_1 and p_2:

```
{<?a,!b>}⊕{<?b,!c>} =
      {<?a,!b,?b,!c>,
      <?a,?b,!b,!c>,
      <?a,?b,!c,!b>,
      <?b,?a,!b,!c>,
      <?b,?a,!c,!b>,
      <?b,!c,?a,!b>}
```

It should be clear that the resulting set of traces effectively contains all allowable orderings of the individual communication events. Hence, the newly generated set of traces describes the combined behavior. That means that the concurrent operation of two processes can be treated as one single (virtual) process. This statement has an important consequence. Instead of studying the interaction of one process with a set of processes, we can restrict ourselves to the communication between *two* processes: the process at hand and the 'virtual process' constituted by all the other processes in the system (these processes constitute the environment).

Definition 9.2 Let $P_1,\ldots,P_n$ be n processes running in parallel. The *environment* E created by the joint operation of these processes is given by the interleaving of their sets of traces, $E = P_1 \oplus \ldots \oplus P_n$, augmented with all the prefixes of these traces.

The reason why the prefix closure is taken is related to the fact that traces, generated by the environment do not have to be consumed entirely.

Example 9.3 The environment E of the sample program

```
p1 ←?a,!b
p2 ←?b,!c
```

with two processes p_1 and p_2 is given by

```
{<?a,!b,?b,!c>,<?a,!b,?b>,<?a,!b>,<?a>,<>,
<?a,?b,!b,!c>,<?a,?b,!b>,<?a,?b>,
<?a,?b,!c,!b>,<?a,?b,!c>,
<?b,?a,!b,!c>,<?b,?a,!b>,<?b,?a>,<?b>,
<?b,?a,!c,!b>,<?b,?a,!c>,
<?b,!c,?a,!b>,<?b,!c,?a>,<?b,!c>}
```

It contains all traces or prefixes of traces which could ever be generated by the processes when running independently. The set E is called the *prefix closure* of {<?a,!b>} ⊕{<?b,!c>}.

Definition 9.4 A *Herbrand blackboard* is a bag of terms of the Herbrand universe.

Examples of Herbrand blackboards for H_U = {a,b,c} are: [a,b,a,a], [a,a], [], etc. The blackboard is a model for the common 'communication space' of all processes. We denote the set of all Herbrand blackboards as H_{BB}.

The execution of events by a process will give rise to changes in the blackboard. The event !a will put the term a onto the blackboard, while ?a will get a from the blackboard, as soon as it is available. The blackboard will only contain the net effect of both operations. We say that the blackboard is modified by the *execution of a process trace*, and that a trace is a blackboard-transforming function. Therefore, we give a more useful definition of a trace

Definition 9.5 A *trace* $t: H_{BB} \mapsto H_{BB}$ is a blackboard-transforming function. For s a Herbrand blackboard, and τ a ground term, the function is defined as a set of rewriting rules:

$$<>s \sim s$$
$$(<!\tau>+t)s \sim t(s \cup \{\tau\})$$
$$(<?\tau>+t)(s \cup \{\tau\}) \sim ts$$

Example 9.6 The application of the trace <?a,!b,!c,?c> on the blackboard [b,a] can thus be rewritten as

```
<?a,!b,!c,?c>[b,a] ~ <!b,!c,?c>[b] ~
     <!c,?c>[b,b] ~ <?c>[b,b,c] ~ <>[b,b] ~ [b,b]
```

When the rewriting rules are used left to right, the length of the trace is always decreasing. Eventually it will be the empty trace, which can be omitted, resulting in a Herbrand blackboard. However, some trace applications cannot be rewritten as a Herbrand blackboard, such as, <!a,?b,!c>[] ~ <?b,!c>[a]. The expression <?b,!c>[a] cannot be simplified further.

The rewriting equations describe an equivalence relation between trace applications on a Herbrand blackboard. This equivalence relation introduces a partition on the set of trace applications on Herbrand blackboards.

Definition 9.7 A trace t is *fully executable* on a blackboard s if the trace application t s is equivalent to a Herbrand blackboard (the element of the partition to which t s belongs, contains also *one* Herbrand blackboard).

We call H_T the set of fully executable Herbrand traces on the empty blackboard. The property of full executability is thus equivalent to set membership of the set H_T.

Example 9.8 The set of fully executable traces H_T over the alphabet {!a,?a} is given by {<>,<!a>,<!a,?a>,<!a,!a>,<!a,!a,!a>,<!a,!a,?a>,<!a,?a,!a>,...}.

10 Declarative Semantics: Formalization

Definition 10.1 The *declarative semantics* of a number of processes connected to a blackboard is given by the subset of the Herbrand base that is true.

Let $P_1,\ldots,P_n$ be n processes connected to the blackboard, let E be the environment created by these processes, and let M' be the (global) generalized model of the program. The declarative semantics is then given by

$$\{a \in H_B : \exists t \in M'[a] \ \wedge \ \exists t' \in E \ (t \oplus t') \cap H_T \neq \emptyset\},$$

where $M'[a]$ denotes the set of traces that can make a true: $M'[a] = \{t \in H_B : (t,a) \in M'\}$. The declarative semantics equals the subset of the Herbrand base of which the elements are true by using a trace generated by the environment. In other words, there must be at least one trace, needed to make a true, and one trace, generated by the processes in the system, which give rise to at least one fully executable interleaving.

Example 10.2 Consider the program

```
t(a)←
t(b)←
p1←?x,t(x),!c
p2←?c,!d
main←!a,?x,x=d.
```

The Herbrand universe H_U is given by {a,b,c,d}. The program contains two processes p_1 and p_2 with the sets of traces {<?a,!c>,<?b,!c>} and {<?c,!d>}. The generalized model M' is given by

```
{(<>,t(a)),(<>,t(b)),(<?a,!c>,p1),(<?b,!c>,p1),
(<?c,!d>,p2),(<!a,?d>,main),(<>,a=a),(<>,b=b),
(<>,c=c),(<>,d=d)}.
```

All atoms which need the empty trace to be true are automatically part of the semantics of the program ($<> \oplus <> \cap H_T \neq \emptyset$ because $<> \in H_T$) The atoms p_1 and p_2 cannot be proved because there is no trace from the environment which can make the traces for p_1 and p_2 fully executable.

The interleaving of the trace <!a,?d> (cfr. main) and the trace <?a,!c,?c,!d> from the environment {<?a,!c>,<?b,!c>}⊕{<?c,!d>} contains precisely one fully executable trace <!a,?a,!c,?c,!d,?d>. This means that main belongs to the declarative semantics. which is given by {t(a),t(b),main,a=a,b=b,c=c,d=d}.

It is remarkable that the semantics of a program, consisting of a number of concurrently operating processes can be expressed in the same way as the semantics for a sequential program. This shows that a set of cooperating logic programs is a valid way to introduce parallelism in logic programming, and that the use of a blackboard for communication is

not in conflict with the logic programming paradigm. The semantics of both a sequential and a parallel program can be expressed by the same means.

Here also it becomes clear why we need a prefix closure for the environment E. Consider the process p←!a,?b and the goal ←?a. After spawning the process p, the goal ←?a can be solved, although the process p will suspend on the event ?b. According to the definition, the set <!a,?b>⊕<?a> does not have a fully executable trace, but the set <!a>⊕<?a> has. Actually, as soon as our goal can be made true, we do not bother about the other processes any longer. Their state has become unimportant, so they may get blocked.

11 Operational Semantics: Formalization

Definition 11.1 The *operational semantics* of a program consisting of a number of processes, connected to a blackboard is given by the set of atoms which can be proved true by executing a finite sequence of events.

In the previous section, a blackboard was considered as a bag of terms from the Herbrand universe. The application of a trace to a blackboard yieds a blackboard if the trace is fully executable. Hence, the application of a fully executable trace on a blackboard is also a representation of a blackboard. Moreover, every Herbrand blackboard can be rewritten as a trace application on the empty Herbrand blackboard.

$[\tau_1,\ldots,\tau_n]$ is equivalent to $<!\tau_1,\ldots,!\tau_n>[]$
$t([\tau_1,\ldots,\tau_n])$ is equivalent to $(<!\tau_1,\ldots,!\tau_n>+t)[]$

This representation can be further generalized to non-fully executable traces. By doing so we can specify more general blackboards than we could up to now. We therefore give a definition of a generalized blackboard.

Definition 11.2 A *generalized blackboard* is defined as a trace application on a Herbrand blackboard or on a generalized blackboard. It belongs to the set $(H_T)^+ \times H_{BB}$.

Examples of generalized blackboards are: <><><><>[], <!a,!b,!c>[], <>[a,b,c], <?a>[a], <?b>[a], <!a><!b>[], and so on.

The operational semantics of generalized blackboards is slightly different from the semantics of Herbrand blackboards. With $t \in H_T$, $s \in H_{BB}$, $s' \in H_{BB}$, the derivation rule for a generalized blackboard is as follows,

$t_1 \ldots t_{j-1} t_j t_{j+1} \ldots t_n \; s \rightarrow\!\!\!\!\rightarrow t_1 \ldots t_{j-1} t_j' t_{j+1} \ldots t_n \; s'$ where
t_j is the selected trace $\wedge$ $t_j = <e>+t_j' \wedge <e>s \sim <>s'$

Due to the fact that more than one trace t_j may be ready to be executed, we introduce a selection function S_p. It is a formalization of the *interprocess non-determinism*. At any moment in the derivation process, it returns the process (or trace) that will generate the next event on the blackboard. It is a function $(H_T)^+ \times H_{BB} \mapsto H_T \cup \{\textbf{success}, \textbf{deadlock}\}$. The function returns **success** if the trace in front of the generalized blackboard is equal to <>, and returns **deadlock** if there is no event which can be executed on the blackboard.

Example 11.3 An example may clarify the semantics of a general blackboard (the selected trace is underlined).

<!a,?b><?a,!b><?a,!c>[] →→
<?b><?a,!b><?a,!c>[a] →→
<?b><!b><?a,!c>[] →→
<?b><><?a,!c>[b] →→
<><><?a,!c>[]

Of course, the selection function could have been different, as is shown in the next example.

<!a,?b><?a,!b><?a,!c>[] →→
<?b><?a,!b><?a,!c>[a] →→
<?b><?a,!b><!c>[] →→
<?b><?a,!b><>[c]

In this case, the computation ends with a **deadlock**. One process is waiting for b, the other for a. Neither of them will ever be generated on the blackboard.

A generalized blackboard is a formalization of the interaction of primitive processes (i.e., single trace processes) on a blackboard. However, it is not yet a formalization of real processes. In order to create a full description of such a system, we need to introduce one more generalization, which is another source of non-determinism. Indeed, the description of a general process is not a single trace, but a set of traces. Not every trace will give rise to a successful computation. The choice of a particular trace out of this set of traces, is done by the trace selection function S_T.

Definition 11.4 A general blackboard is *successfully executable* if there exists a selection function S_p such that $t_1 \ldots t_n[]$ can be rewritten as $<>t_2' \ldots t_n' b$.

Definition 11.5 A *goal expression* is a set-based general blackboard, i.e., it belongs to the set $((H_T)^+)^+ \times H_B$.

Example 11.6 Examples of goal expressions are:

{<>}[]
{<!a>,<!b>}[]
{<!a>,<!b>}{<?a>,<?b>}[c]

Goal expressions can be rewritten as a set of generalized blackboards; e.g., the goal expression

{<!a,?d>}{<?a,!c>,<?b,!c>}{<?c,!d>}[]

is equivalent to the set of generalized blackboards (i.e., single-trace processes)

{<!a,?d><?a,!c><?c,!d>[], <!a,?d><?b,!c><?c,!d>[]}.

Hence, the transition from an program with primitive single-trace processes to an program with multi-trace processes is just a matter of creating a set of single-trace processes *and* selecting one single-trace process. This way of treating the *intra-process non-determinism* is allowed because a process will generate *at most* one trace.

Definition 11.7 A goal expression is *successfully executable* if it contains a generalized blackboard that is successfully executable.

In order to check whether a goal expression is successfully executable, we need two selection functions. One to select a generalized blackboard from the set of generalized blackboards (the trace selection function S_T), and one to select the order in which processes will generate the communication events (process selection function S_P).

Definition 11.8 The operational semantics of a program with generalized model M', and consisting of n processes $P_1, \ldots, P_n$ is given by

$$\{a \in H_B : \quad M'[a]\, P_1 \ldots P_n [] \text{ is successfully executable}\}$$

The operational semantics is sound, i.e., if an atom a belongs to the operational semantics, then it also belongs to the denotational semantics.

The operational semantics is non-deterministically complete. This means that an atom that belongs to the denotational semantics, can be proved operationally, iff the proper trace selection function and process selection function is available.

12 Conclusion

The main contribution of this work is that the semantics of a set of cooperating processes connected to a blackboard can be described in precisely the same way as for a sequential program, i.e., as a subset of the Herbrand base. This means that, given a subset of a Herbrand base, one could choose either a sequential, or a parallel implementation of that semantics. It shows that the concepts of macroscopical parallelism and blackboard communication are not in conflict with the logic programming paradigm.

13 Future Work

We are now implementing Multi-Prolog, a programming language based on the model described. It would be interesting to investigate whether it is possible to automatically detect the application parallelism by inspecting the relations between procedures. It is probably possible to give the application programmer a hint how to create processes. This task is not trivial because the decomposition is not based on the syntax of the program, but on the semantics. It is related to the modularity of knowledge and locality in the program.

14 Aknowledgement

The author would like to express his gratitude towards Prof. J. Van Campenhout for the valuable discussions on this topic. He also wants to thank L. Wulteputte for the prototype implementation of this language.

References

[1] Brogi, A., and Ciancarini, P., "The Concurrent Language Shared Prolog", *ACM Transactions on Languages and Sytems*, 13(1), 1991.

[2] Brookes, S.D., Hoare, C.A.R., Roscoe, A.W., "A Theory of Communicating Sequential Processes", in *Journal of the ACM*, Vol. 31, no. 3, July 1984, pp. 560–599.

[3] De Bosschere, K., "Multi-Prolog, Another Approach for Parallelizing Prolog", in *Proceedings of Parallel Computing 89*, Leiden, NL, august 1989.

[4] De Bosschere, K., "Multi-Prolog, a Process-oriented Prolog", in *Proceedings of Software Engineering for Real-Time Systems*, Cirencester, UK, september 1989.

[5] Gelernter, D., "Generative Communication in Linda" in *ACM Transactions on Programming Languages and Systems*, 7(1), p. 80–112, jan 1985.

[6] Hennessy, M., *Algebraic Theory of Processes*, in MIT Press Series The Foundations of Computing, Cambridge, Massachusetts, 1988.

[7] Lloyd, J.W., *Foundations of Logic Programming*, Springer Verlag, Berlin, 1984.

[8] Manna, Z., *Mathematical Theory of Computation*, McGraw Hill, New York, 1974.

[9] Monteiro, L., *Distributed Logic, A theory of distributed programming in logic*, Report Computer Science Dept., University of Lisbon, Portugal, 1986.

[10] Van de Snepscheut, J.L.A., *Trace Theory and VLSI Design*, Lecture Notes in Computer Science, no. 200, Springer-Verlag, Berlin, 1985.