Journal of Universal Computer Science, vol. 3, no. 5 (1997), 568-602
submitted: 20/12/96, accepted: 20/5/97, appeared: 28/5/97 © Springer Pub. Co.

Correctness proof of a Distributed Implementation of Prolog by means of Abstract State Machines.

Lourdes Araujo
(Dpto. Informática y Automática (Fac. Matemáticas)
Universidad Complutense de Madrid
Madrid 28040, Spain
lurdes@dia.ucm.es)

Abstract: This work provides both a specification and a proof of correctness for the system PDP (Prolog Distributed Processor) which make use of Abstract State Machines (ASMs). PDP is a recomputation-based model for parallel execution of Prolog on distributed memory. The system exploits OR_parallelism, Independent AND_parallelism as well as the combination of both. The verification process starts from the SLD trees, which define the execution of a Prolog program, and going through the parallel model, it arrives to the abstract machine designed for PDP, an extension of the WAM (Warren Abstract Machine), the most common sequential implementation of Prolog. The first step of this process consists in defining *parallel SLD subtrees*, which are a kind of partition of the SLD tree for programs whose clauses are annotated with parallelism. In a subsequent step the parallel execution approach of PDP is modeled by means of an *OR_TASK* ASM. In this ASM each task is associated with the execution of a parallel SLD subtree. The execution of the parallel SLD subtree corresponding to each task is modeled by a *NODE* submachine which is an extension of the one proposed by Börger and Rosenzweig to verify the sequential execution of Prolog. Accordingly, the verification leans on the results of this work in order to avoid the verification of the common points with the sequential execution. The new elements of the execution due to parallelism exploitation are modeled at successive steps of the verification process, finally leading to the extended WAM which implements PDP. The PDP verification proves correctness for this particular system but it can readily be adapted to prove it in other related parallel systems exploiting AND, OR or both kinds of parallelism.

1 Introduction

Abstract State machines (ASMs) (or *Gurevich ASMs*) are a useful tool to express and verify algorithms in a precise way. Originally, the idea was to provide operational semantics for programs and programming languages by improving Turing's thesis [Gur91] —according to which every algorithm can be simulated by an appropriate Turing machine. Translating a given algorithm into a Turing machine may, however, be very tedious because every step of the algorithm may require a long sequence of steps of the Turing machine. Gurevich looked for machines able to simulate algorithms in a stepwise manner —one step of the simulating machine for each step of the algorithm. ASMs are intended to be such machines. Furthermore, while Turing machines have a fixed level of abstraction

—which may be very low, an ASM may be tailored to the abstraction level of the given algorithm. If an algorithm is given as a program in some programming language then the appropriate ASM provides the operational semantics for the program on the abstraction level of the program. One may have a whole hierarchy of ASMs of various abstraction levels for the same algorithm. At every abstraction level of the verification process the system is defined by both its basic *objects* and the elementary *operations* which determine its dynamic behavior or *state*. Thus, each level is represented by means of an ASM consisting of a pair $(\mathcal{A}, \mathcal{R})$, where $\mathcal{A}$ is a set of domains with partial functions, and $\mathcal{R}$ is a finite system of *transition rules* [Gur91].

ASMs also provide a powerful and simple mechanism for *information hiding* and definition of the *precise interface* by means of *external functions*. Any function g not appearing in any update of the transition rules R is called external for R and for the ASM corresponding to R. Otherwise, the function is internal. The rules of the ASM give no information on the behavior of external functions, and they can not be modified by the rules, but they can be used in the rules to determine arguments at which an internal function is changed.

Many programming languages and systems have also been specified or verified by using ASMs. Some of them make a detailed description of a language or a system architecture: the operational semantics of Occam is described in [Gur89, Bör94a]; the semantics of the concurrent logic programming language Parlog has been represented by an ASM [Bör93]; the architecture of the Parallel Virtual Machine (PVM), a software system to manage a heterogeneous set of computers as a distributed memory system, has been defined to provide a correct understanding of the system at the C-interface level [Bör94c]. In some cases ASMs have been used not only to specify the system but also to provide a proof of correctness with respect to the specification of a language or another system: Börger and Mazzanti [Bör97] present a correctness proof for pipelining with respect to the sequential model in RISC architectures; Börger and Durdanovic present a proof of correctness of transputer code with respect to Occam [Bör96]; a graph narrowing machine is derived from the functional logic programming language BABEL [Bör94b]; using the technique of successive refinements, Börger and Rosenzweig [Bör95a] reconstructed the Warren Abstract Machine (WAM) [Warr83] (a virtual machine model which underlies most of the current Prolog implementations) from a Prolog specification ASM. Then, using WAM correctness as an starting point, some extensions of Prolog have been specified or verified; e.g. Beierle and Börger [Bei96] have provided a specification and correctness proof of a Prolog extended with abstract type constraints. This extension is focused on the representation and unification of terms, where the type of the constraints has to be considered.

The aim of this work is to use ASMs to provide a specification and a proof of correctness for the system PDP (Prolog Distributed Processor) [Ara94, Ara97]. To achieve this we again take WAM's proof of correctness for granted. The proof given by Börger and Rosenzweig [Bör95a] to verify the WAM consists of a number of refinement steps leading from an ASM, $(\mathcal{A}, \mathcal{R})$, closer to the Prolog model, to an ASM, $(\mathcal{B}, \mathcal{S})$, closer to the execution system. Both ASMs are related by a *proof map* $\mathcal{F}$ mapping states B of $(\mathcal{B}, \mathcal{S})$ on states $\mathcal{F}(B)$ of $(\mathcal{A}, \mathcal{R})$ and rule sequences R of $\mathcal{R}$ on rule sequences $\mathcal{F}(R)$ of $\mathcal{S}$. Such a proof map is considered to establish *correctness* of $(\mathcal{B}, \mathcal{S})$ with respect to $(\mathcal{A}, \mathcal{R})$ if $\mathcal{F}$ preserves initiality, success and failure of states. Furthermore, the proof map is considered to establish *completeness* of $(\mathcal{B}, \mathcal{S})$ with respect to $(\mathcal{A}, \mathcal{R})$ if every terminating computation in $(\mathcal{A}, \mathcal{R})$ is image under $\mathcal{F}$ of a terminating computation in $(\mathcal{B}, \mathcal{S})$, since in that case every successful (failing) abstract computation may be viewed as implemented by a successful (failing) concrete computation. Since the notion of operational equivalence —reached if both correctness and completeness are proved— is symmetric, the way $\mathcal{F}$ goes is unimportant and thus it may be taken to map $(\mathcal{A}, \mathcal{R})$ on $(\mathcal{B}, \mathcal{S})$.

Parallelism is one of the most successful techniques in the search of efficient execution systems of Prolog. In spite that most of these parallel models have been implemented, what allows to check them, only a formal verification of the system can guarantee the general correctness. This is specially important on parallel execution systems, in which the execution of a program changes from run to run, depending on the available processors and the fluidity of communications.

PDP is a recomputation-based model for parallel execution of Prolog on distributed memory. The system exploits OR_parallelism, Independent AND_parallelism, and the combination of both.

The extension of Prolog to be analyzed in the present work is mainly concerned with the treatment of the structure of predicates and clauses where parallelism appears. The PDP verification process starts from the SLD trees, and going through the parallel model it arrives to the abstract machine designed for it —the WAM. An OR_parallel execution of a Prolog program can reach a number of solutions larger that the sequential execution. The search space is explored depth-first, left to right in the sequential execution, so that infinite branches cause that solutions on their right be never reached. Therefore, it is not possible to establish correctness of the parallel execution algorithm with respect to any model of the sequential one, and the starting point of the verification process has to be the SLD tree. Thus, the first step of this process consists in defining *parallel SLD subtrees*, which are a kind of partition of the SLD tree for programs whose clauses are annotated with parallelism.

In a subsequent step the parallel execution approach of PDP is modeled by means of an *OR_TASK* ASM. In this step every task represents the execution of a parallel SLD subtree, and the model is proved to be correct wrt the previous one. Completeness must be restricted to the special case in which each branch of the SLD tree corresponds to a different parallel SLD subtree, i.e. to the cases in which every program clause is annotated with parallelism. The execution of the parallel SLD subtree corresponding to a given task is modeled by a *NODE* submachine which extends the one proposed by Börger and Rosenzweig to verify the sequential execution of Prolog. Therefore, the verification heavily leans on the results of this work in order to avoid the verification of the common points with the sequential execution. Accordingly, in spite that we have tried to make this article self-contained, by including definitions and terminology borrowed from Börger and Rozensweig's work, reading their article first will certainly help to understand this one.

The new elements of the execution due to parallelism exploitation are modeled at successive steps of the verification process, finally leading to the extension of the WAM used to implement PDP. The system is still modeled by the *OR_TASK* ASM at every verification step, but further ASMs must be included in order to model the new elements introduced by parallelism in the execution of goals inside each task.

The exploitation of AND_parallelism leads to introduce AND_tasks. PDP is still modeled by OR_tasks, but now they can execute some of their subgoals in parallel by creating AND_tasks. The results of the AND_tasks execution must be synchronized before continuing the OR_task execution. This creates new difficulties which have to be dealt with.

At this point the exploitation of combined parallelism is modeled. This leads to a new distinction among the kind of tasks, primary or secondary, depending on the kind of their parent task.

External functions are very useful to represent the environment in which an ASM is defined, and they are used in the verification of PDP to represent the scheduling and communications.

The PDP verification not only provides correctness results for this particular system but it can be readily adapted to other related parallel systems. In this way, since the PDP approach to exploit AND_parallelism is an extension for distributed memory systems of the RAP model [Her86], the verification of this model is also obtained as a particular case. In the same way the verification of the OR_parallel exploitation by recomputation may be easily adapted to systems which use a different method for the exploitation of OR_parallelism, such as the copying in MUSE [Ali90].

The rest of the paper proceeds as follows: section 2 presents an overview of PDP; section 3 defines the parallel SLD subtrees from which the verification starts from; section 4 describes the Prolog tree task, the first step in the verification process; section 5 introduces AND_tasks to model AND_parallelism; section 6 describes the way in which combined parallelism is modeled; section 7 models the communication level of the system; section 8 introduces stacks in the tasks to approach PDP's final architecture; section 9 deals with predicate structure, while section 10 deals with clause structure. Finally, after outlining how the model is completed in section 11, conclusions are drawn in section 12.

2 Overview of PDP

PDP [Ara94, Ara97] is a multisequential system supporting both independent-AND and OR_parallelism, as well as its combination. Parallelism is supposed to be annotated in the source program. Because the system is devoted to distributed memory architectures, the execution model has been designed in such a way that there are no variables in a worker defined in terms of variables that belong to another worker, thus reducing communication overhead. Independent AND_parallelism is exploited by following a *fork-join* approach, which is an extension for distributed memory systems of the one followed in the RAP model [Her86]. Each goal in a *parallel call*, set of independent goals annotated to be executed in parallel, is executed in a different processor, if available. Thus, the results of the execution of each goal are collected in the parent processor (the one finding the parallel call).

OR_parallelism is exploited by following a recomputation approach [Ara93]; a processor environment is reconstructed by recomputing the initial goal without backtracking (see Figure 1), following the socalled *success_path*, i.e. the sequence of clauses which have succeeded, obtained from the parent processor (the one finding the parallel clause). Recomputation allows the exploitation of OR_under_AND parallelism in a very natural way. The PDP approach to exploit

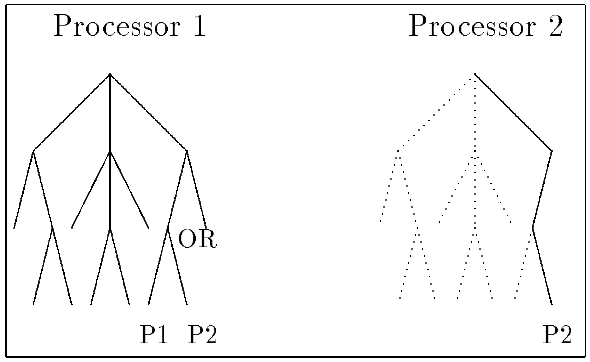

Figure 1: Recomputation model. Processor 1 has had to explore the search tree on the left of P1 to arrive at this point, while processor 2 has only computed the shorter path (success path) until P2, which is annotated to be executed in parallel. This path to P2 is provided by processor 1.

OR_under_AND parallelism [Ara94] is designed to create, in an automatic and decentralized way, an independent computation for each solution.

OR_parallelism appearing under AND_parallelism is exploited by recursively splitting off the execution of OR_parallel clauses into independent branches of computation (each with a pure AND_parallel call, as illustrated in Fig. 2), thus taking advantage of the recomputation technique. This approach has two main advantages: first, it avoids storing solutions because no parallel call is waiting for the completion of the whole set of solutions, and second, the splitting off algorithm can resolve any OR_under_AND parallel call in an automatic, decentralized way, i.e. no processor has to perform the splitting, but anyone which gets idle selects, by applying the algorithm, its corresponding solution.

The results of the implementation of this model [Ara97] have proved that OR_parallelism exploitation provides a linear speedup for high granularity programs. For some programs presenting both kinds of parallelism PDP achieves a greater speedup than the product of the speedups achieved by exploiting each kind of parallelism separately. The reason is that the exchange of messages required in the exploitation of AND_parallelism is avoided in PDP when OR_under_AND parallelism is exploited.

2.1 Task based execution

The computation of a goal in PDP is called a *task*. Two types of tasks are distinguished: *OR_tasks* and *AND_tasks*. An OR_task computes solutions to the initial goal by exploring a portion of the search tree. An AND_task computes a solution to a goal which belongs to a parallel call. A task (*parent task*) exploits AND_parallelism and OR_parallelism by creating new AND_tasks and OR_tasks respectively. In this way, the parallel execution of a program defines a *task tree*. The model supports combined parallelism in a very natural way. As a result, the execution of the search tree is automatically distributed among tasks by means

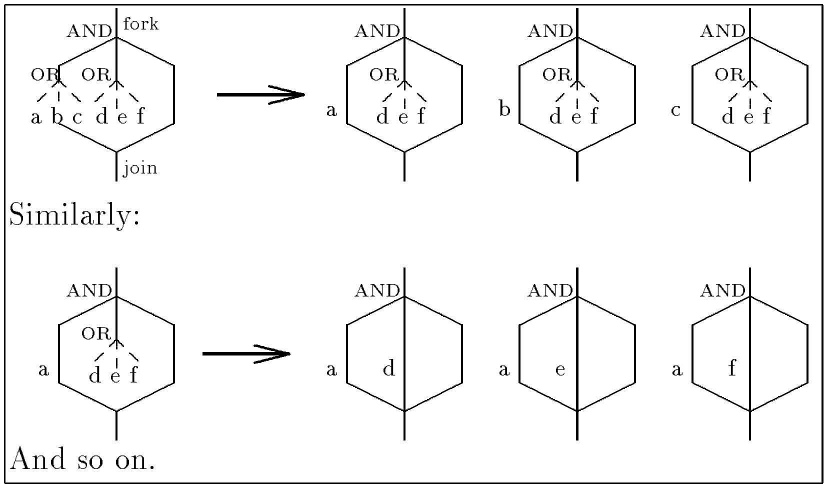

Figure 2: Example to illustrate the combined parallelism algorithm. OR_parallelism in the first branch of a parallel call leads to independent computations of the parallel call, each of them with a different solution for the parallel branch. OR_parallelism in the remaining branches of the parallel call is exploited similarly.

of a combination rule, so that no specific task is in charge of the distribution. The model is outlined as follows:

- The program execution begins as an OR_task (the root of the task tree), which performs a sequential computation until a parallel call or a parallel procedure is reached.
- The execution of a *parallel call* is carried out by the creation of an *AND_task* for each independent goal. These AND_tasks receive from its parent task a goal and the computed answer substitution restricted to the variables of the goal. Each AND_task computes its goal, returns the *local solution* to the parent task and finishes. The parent task waits for the answer to each independent goal and it is in charge of synchronizing the reception of those answers.
- The execution of a *parallel procedure*, i.e. a procedure with OR_parallelism, is carried out by the creation of a new *OR_task*. This receives the *success_path* of the predecessor OR_task and recomputes the initial goal following this path. After the recomputation, the execution of the next solution is computed sequentially.
- If both kinds of parallelism appear combined, parallelism is still exploited by creating the corresponding tasks. If AND_parallelism appears under OR_parallelism, the execution is performed as in the case of pure AND_parallelism, since the exploitation of OR_parallelism produces the same environment as a sequential execution.
- If OR_parallelism appears under AND_parallelism, the OR_tasks arising from an AND_task have to re-execute the parallel call in order to find new solu-

tions to it. If this were done blindly, the result would be the simple *repetition of solutions*. To avoid this, it has been introduced a *combination rule* which decides which branch is explored to solve each independent goal. The *ancestor goal* of an OR_task arising from an AND_task is defined as the goal executed by this AND_task. The key point of the combination rule is to fix the solution of the goals on the left of the ancestor goal and to combine them with *every* solution of the remaining goals.

The PDP approach to exploit combined parallelism – when it appears in the form OR_under_AND – is based upon the fact that the recomputation allows the AND_tasks to exploit OR_parallelism by creating OR_tasks. If an AND_task finds OR_parallelism, it creates a new OR_task to deal with the parallel clauses, and transfers to it the success_path leading to the parallel call. Notice that the AND_task has received this information only for this purpose. The new OR_task applies the combination rule in order to decide which solution is to be explored. The PDP approach to exploit OR_under_AND parallelism leads to a distinction between different types of OR and AND_tasks. The type of a task depends on both the type of its parent task and the ancestor goal position. The types of tasks are:

- **Primary OR_task:**
  This is created by an OR_task to exploit OR_parallelism. When the recomputation of the received success_path is completed, the execution proceeds in the normal way.
- **Secondary OR_task:**
  This is created by an AND_task to exploit OR_parallelism. When the recomputation of the success_path leading to the parallel call is completed, a new combination of solutions is created.
- **Primary AND_task:**
  This is created to exploit AND_parallelism by an AND_task, a primary OR_task, or a secondary OR_task, provided the latter does not correspond to a goal on the left of the ancestor goal. Primary AND_tasks exploit OR_parallelism appearing during the execution.
- **Secondary AND_task:**
  This is created to exploit AND_parallelism by a secondary OR_task corresponding to a goal on the left of the ancestor goal. According to the combination rule, this task must ignore any OR_parallelism appearing during the execution.

The example of Figure 3 illustrates the way in which the different tasks are created and the way in which the combination rule works to achieve every solution without repetitions. The parallel execution of the program appearing in Figure 3a) is represented in Figure 3b). The query of the program presents AND_parallelism ($a\&b$) and thus the goals a and b can be computed at the same time. Furthermore, the procedures for a and b present OR_parallelism (*) and thus their clauses can be explored simultaneously so as to find different solutions. The query is always executed by a primary OR_task, a kind of task able to reach a final solution and to exploit all parallelism found. This OR_task finds out the goal annotated with AND_parallelism (&) and creates primary AND_tasks to execute each subgoal. The first solutions ($a1$ and $b1$) reached by these tasks are returned to their parent task. However, the AND_tasks find out the annotation of OR_parallelism (*) and thus create new OR_tasks to execute the query again with other clauses. These new OR_tasks are secondary since they have been created by AND_tasks and have to follow a particular path to avoid arriving to solutions already explored. The new OR_tasks have the goal a and b as ancestor goal respectively. Accordingly, the OR_task which computes $(a_2, b_1), c$ exploits AND_parallelism by means of primary AND_tasks, because there is not goals on the left of the ancestor one, while the OR_task which computes $(a_1, b_2), c$ creates a secondary AND_task, which does not exploit OR_parallelism, to solve the goal a which is on the left of the ancestor goal b. The computation of the solution $(a2, b2), c$ follows the same scheme.

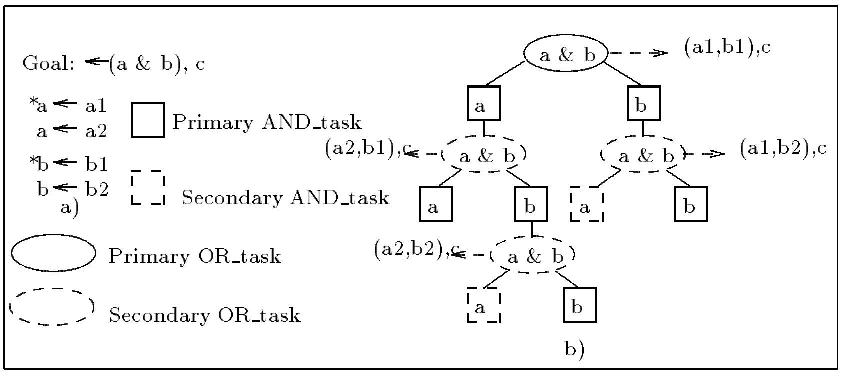

Figure 3: PDP execution scheme of a goal annotated with AND_parallelism (&) whose clauses are annotated with OR_parallemism (*).

2.2 PDP Architecture

In order to reduce the communications overhead, PDP has been designed with a hierarchic scheduling policy. PDP is composed of a set of clusters, each of them consisting of a *scheduler* and a set of *workers*. Schedulers are responsible for the distribution of pending work among idle workers. Each worker operates on its own private memory and interprocessor communication is performed only by the passing of messages. A worker executes a task of any type until it is finished, then executes a new one, and so on.

Every worker implements an extension of the WAM [Warr83] consisting in the addition of new data structures and instructions related to parallelism. The *success stack* and the *success pointer* (SP) have been added to record the *success_path*. Other data structures have been introduced with the purpose of synchronizing the execution of a parallel call. The *Cross Product Environment* (CPE) associated with each parallel call has been introduced in order to perform the combination of solutions according to the combination rule. Prolog programs are compiled to PDP instructions. These consist of WAM instructions along with instructions to manage each kind of parallelism. A complete description of the PDP data structures and instructions can be found in [Ara94, Ara97]. The detailed refinement steps arriving to the PDP architecture are not included in this work but can be found elsewhere [Ara96].

3 Parallel SLD subtrees

From now on this work will deal with logic programs annotated with parallelism. These annotations may be added to clauses (OR_parallelism) or to goals (AND_parallelism).

Def 3.1 *A* labeled program clause *is a pair consisting of a clause and the constant* parallel*:*

$$((A \leftarrow B_1, \cdots, B_n), parallel)$$

Def 3.2 *A* labeled logic program *is a finite set of program clause and labeled program clauses.*

The SLD tree may be partitioned into a number of subtrees, all of them with the query as root. The idea of these subtrees is that they correspond to an OR_parallel computation.

Def 3.3 *Let P be a labeled program with an order established among their clauses, and let G be a goal and R a computation rule. Then a* parallel SLD subtree *for $P \cup \{G\}$ via R is defined as follows:*

1. *Each node of the tree is a goal.*
2. *The root node is G.*
3. *Let $\leftarrow A_1, \cdots, A_m, \cdots, A_k$ $(K \geq 1)$ be a node in the tree and let A_m be the atom selected by R. Then this node has a descendent for (the variant under renaming of) each clause in the subset of clauses $A \leftarrow B_1, \cdots, B_q$ of P such that A_m and A are unifiables for each one of them, they are in sequence, the first of them is either the first of the predicate or the following to a labeled program clause, and the last of them is either a labeled program clause (the only in the subset) or the last of the predicate.*
4. *Nodes which are the empty clause have no descendent.*

Figure 4 shows an example for the following labeled program:

$$((p \leftarrow p_1), parallel)$$
$$(p \leftarrow p_2)$$
$$((p \leftarrow p_3), parallel)$$
$$(p \leftarrow p_4)$$
$$(p \leftarrow p_5)$$

$$(p_1 \leftarrow q_1)$$
$$(p_1 \leftarrow q_2)$$
$$\cdots$$

Thus, the clauses of a predicate can be seen as a set of subsequences, where subsequences can be executed in parallel:

$$[[p \leftarrow p_1], [p \leftarrow p_2, p \leftarrow p_3], [p \leftarrow p_4, p \leftarrow p_5]]$$

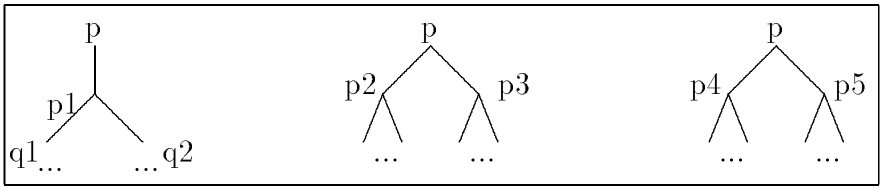

Figure 4: SLD subtrees. The set of clauses until one marked parallel (including it) are assigned to the same SLD subtree.

The relation of parallel SLD subtrees and the SLD tree is easily established:

Theorem 3.1 *Each success branch of a parallel subtree corresponds to a success branch of the SLD tree, and each failure and infinite branch of a parallel subtree corresponds respectively to a failure and infinite branch of the SLD tree.*

Proof By induction over the number of parallel clauses.

4 Prolog tree task

As the initial ASM with respect to which the correctness and completeness of the PDP implementation of Prolog will be proved, let us choose an *OR_task* ASM. The idea is to define a universe OR_TASK whose elements correspond to a depth-first, left to right execution of a parallel SLD subtree. The Prolog tree representation proposed in [Bör95a], extended to deal with parallelism, has been adopted as a submachine which models the execution of the goals inside each task.

The computation inside each OR_task is modeled by a NODE submachine, an adaptation of the one proposed by Börger and Rosenzweig [Bör95a] for the whole SLD tree:

$$(NODE; root, currnode; father)$$

with the function

$$fnode : OR\_TASK \rightarrow NODE$$

The function $fnode$ provides the $currnode$ of the task.

In order to make the paper selfcontained, the ASM proposed as initial level in [Bör95a] is outlined here. For a complete explanation of the rules of the node machine see the work by Börger and Rosenzweig [Bör95a]. Let us call *act* (activator) the selected literal of a node n. For n they are created as many sons as unlabeled alternatives there are to solve *act*. Each son has associated a *candidate clause* of the program. The sons of n are attached to it as the list $cands(n)$, reflecting in this way the order of the clauses. When a labeled alternative is reached a new task is created.

When a node n gets first visited ($Call$ mode) new nodes are created for each $cands(n)$ and the mode changes to $Select$. In this mode the first unifying son from $cands(n)$ gets visited ($currnode$), again in $Call$ mode. If in $Select$ mode there are no $cands(n)$ left, then control returns to the father of $currnode$ (backtracking). The $father$ function then may be seen as representing the structure of Prolog's backtracking behavior. The mentioned switching modes are represented by a distinguished element $mode \in \{Call, Select\}$.

Now, in order to complete the description of this level we have to complete the signature and to define the transition rules. We assume the universes of Prolog literals, goals, terms and clauses of [Bör95a]:

$$LIT, GOAL = LIT^*, TERM, CLAUSE$$

The computation state of a node is associated by functions on universe NODE.

The representation of the cut (!) operator at this level follows the one of [Bör95a]. The goals waiting for execution in a state are not represented linearly, but as subsequences in which clause bodies are decorated with *cutpoints* to which they are to return in case of backtracking. Thus, we have the following representation:

$$DECGOAL = GOAL \times NODE$$
$$decglseq : NODE \rightarrow DECGOAL^*$$

Assuming a $SUBST$ universe of $substitutions$ we have the following functions

$$s : NODE \rightarrow SUBST$$
$$unify : TERM \times TERM \rightarrow SUBST \cup \{nil\}$$
$$subres : DECGOAL^* \times SUBST \rightarrow DECGOAL^*$$

where s represents the current substitution in a state, $unify$ associates to two terms either their unifying substitution or the indication that there is none, and $subres$ yields the result of applying the given substitution to all goals in the sequence. It is also assumed a substitution concatenating function $\circ$.

As in [Bör95a], the renaming of variables is represented by the function:

$$rename : TERM \times N \rightarrow TERM$$

which renames the variables of the term with the given index. The current renaming index is given by the 0-ary function vi.

Furthermore, we will use all the usual list operations adopting standard notation. hd and bdy are also used to select head and body of clauses, respectively.

As in [Bör95a], it is still used an abstract universe CODE of clause occurrences, with the functions *cll(n)* being the candidate clause occurrence of a candidate son n of a computation state, and *clause(p)* being the clause pointed to by p. It is also assumed a *procdef* function to yield the list of candidate clause occurrences for the given literal in the given problem.

$$clause : CODE \rightarrow CLAUSE$$
$$cll : NODE \rightarrow CODE$$
$$procdef : LIT \times PROGRAM \rightarrow CODE^*$$

Components of a decorated goal sequence are accessed as:

$$\begin{aligned}
&goal \equiv fst(fst(decglseq))\\
&cutpt \equiv snd(fst(decglseq))\\
&act \equiv fst(goal)\\
&cont \equiv [< rest(goal), cutpt > \mid tail(decglseq)]
\end{aligned}$$

where *act* stands for the selected literal, and *cont* for continuation.

Now, the OR_TASK ASM is defined. Parallel subtrees are represented by a set, *OR_TASK*, with a distinguished element, *initial_task*. Each element of *OR_TASK* has the information for the state of computation of the subtree it represents. This information consists of the pending resolvent, the substitution computed so far, and the sequence of alternative resolvents that have not been tried yet.

The introduction of parallelism requires extending the signature of the NODE submachine. Parallel annotations in the Prolog program are considered by including a universe and a function:

$$\begin{aligned}
&MARKCLAUSE = CLAUSE \times \{seq, or\}\\
&orparallel : MARKCLAUSE \rightarrow BOOL
\end{aligned}$$

Let us also introduce a universe

$$SUCCESS\_PATH = (MARKCLAUSE \times N)^*$$

representing the sequence of followed successful clauses along with the representation of the renaming index, what allows to ensure the same renaming in parallel computations.

The structure of the task tree is given by the function:

$$success\_path : OR\_TASK \rightarrow SUCCESS\_PATH$$

which associates the sequence of clauses leading to the current resolvent of a task. If the last clause of a success_path is labeled with parallelism, then the end of the parallel subtree corresponding to the task has been reached and the alternative clauses are explored in a new task which is created at this point.

Then, the task ASM is defined:

$$(OR\_TASK; initial\_task; success\_path)$$

This is a dynamic ASM and the elements of OR_TASK are created dynamically by the computation out the initial_task, consisting of the query goals and an empty success_path.

The STATUS universe is also introduced with the function:

$$status : OR\_TASK \rightarrow \{working, recomputing, reporting, finished\}$$

to distinguish among different modes of computation. The *recomputation* status corresponds to a task which is reconstructing the state of its parent task out of its success_path. The remaining status have the obvious meaning.

The parallel treatment of *cut* in PDP consists in restricting the parallelism exploitation to goals outside the scope of a cut. This naive approach provides better performance than other approaches when they are applied to distributed memory systems. On this kind of systems the overhead due to the communications required in more complex approaches may be greater than the speedup achieved by the parallelism exploitation. The chosen approach leads to introduce a boolean function *scopecut* used in order to know whether goal is inside the scope of a cut in a clause, that is, whether its execution may be pruned by cut in the clause.

$$scopecut : DECGOAL^* \rightarrow BOOLEAN$$

Once the signature has been established, let us introduce the set of transition rules. In the initial OR_task ASM the *success_path* is empty and the *currnode* of the task is the one having *nil* as root which is the father of *currnode*; the latter has a single element list $[< query, root >]$ as decorated goal sequence, and empty substitution; the mode is *Call*; *db* is the given program and the list *cands* is not yet initialized. The parameter *currnode* is usually abbreviated in the rules $(father \equiv father(currnode)$, $cands \equiv cands(currnode)$, $s \equiv s(currnode)$ and $decglseq \equiv decglseq(currnode)$.

The *query success rule* is modified with respect to the one in [Bör95a] in order to view Prolog as returning all solutions. This rule triggers backtracking

if *all_done*
then *backtrack*

where *all_done* represents $decglseq = [\,]$.
Rules *goal, true, fail* and *cut* of the Prolog tree ASM by Börger and Rosenzweig are maintained in the present representation:

if $goal = [\,]$
then $decglseq := rest(decglseq)$

if $act = true$
then *succeed*

if $act = fail$
then *backtrack*

And the **cut rule** is as follows

if $act = !$
then $father := cutpt$
succeed

where *success* represents $decglseq := cont$.

Rules *call, selection* and *query success* of the NODE machine by Börger and Rosenzweig [Bör95a] have to be modified to introduce the new data representation. While in the machine for the WAM model the *Call* rule creates as many sons of *currnode* as there are candidate clauses in the procedure definition of its *activator*, the machine for the PDP model creates either *tasks* or *nodes* depending on the appearance of parallelism.

```
if status(t) = working
then if  is_user_defined(act) & mode = Call
        then
        let n = length(procdef(act, db))
        seq i = 1,··· , n
            cl := nth(procdef(act, db), i)
            if orparallel(clause(cl)) & not scopecut(decglseq(act))
            then
                extend OR_TASK by task_i with
                 success_path(task_i) := append(success_path(t),< cl, vi >)
                 status(task_i) := recomputation
                endextend
            else
                extend NODE by temp_i with
                 father(temp_i) := fnode(t)
                 cll(temp_i) := cl
                 success_path(temp_i) :=
                        append(success_path(fnode(t)),< cl, vi >)
                 cands := append(cands, [temp_i])
                endextend
        mode := Select
```

with the abbreviations $currnode \equiv node(t)$ and $act \equiv act(t)$.

According to this rule, a new task appears for each parallel clause, while clauses not annotated with parallelism are put in the *cands* list. The rule states that only if act is outside the scope of a cut the parallelism is exploited. In this way it is avoided to create tasks to compute branches that may be pruned by a cut.

The selection rule, which attempts to select a candidate resolvent state, is maintained as in the WAM mode since the selection of nodes follows the sequential model inside each task. It has the following form:

```
if status(t) = working
then if  is_user_defined(act) & mode = Select
        thenif cands = [ ] then backtrack
        else
        let clause = rename(clause(cll(fst(cands))),vi)
        let unify = unify(act,hd(clause))
        if unify = nil
        then cands := rest(cands)
        else  currnode := fst(cands)
              decglseq(fst(cands)):=
                 subres([⟨bdy(clause),father⟩|cont],unify)
              s(fst(cands)) := s ∘ unify
              cands := rest(cands)
              mode := Call
              vi := vi + 1
```

As in the node machine of the WAM model *vi* is a technicality representing the renaming index for the variables. Now, backtrack is the following abbreviation

$$
\begin{aligned}
backtrack \equiv\ & \textbf{if}\ father = root \\
& \textbf{then}\ status(t) := finished \\
& \textbf{else}\ currnode := father \\
& \qquad mode := Select
\end{aligned}
$$

New rules have to be introduced to take into account the different values of status.

The termination occurs when the status of every task is *finished*. In order to represent termination a boolean element *stop* is used.

$$
\begin{aligned}
& \textbf{if}\ \forall\ t\ status(t) = finished \\
& \textbf{then}\ stop = true
\end{aligned}
$$

A new rule is introduced to express the process when the status is *recomputation*. The rule says that the selected clause is the one indicated be the success_path until all their components have been considered. *sp* represents a pointer to the point of the success_path currently being considered.

$$
\begin{aligned}
& \textbf{if}\ \ status(t) = recomputation \\
& \textbf{then}\ \ \textbf{if}\ success\_path(act,sp) = nil\ /*\ success\_path\ finished\ */ \\
& \textbf{then}\ status := working \\
& \textbf{else}\ \ \textbf{let}\ vi = snd(hd(success\_path(act,sp))) \\
& \qquad \textbf{let}\ clause = rename(fst(hd(success\_path(act,sp))),\ vi) \\
& \qquad \textbf{let}\ unify = unify(act,hd(clause)) \\
& \qquad decglseq(currnode) := \\
& \qquad\quad subres([\langle bdy(clause),father\rangle|cont],unify) \\
& \qquad s(currnode) := s \circ unify \\
& \qquad sp := sp + 1
\end{aligned}
$$

I want to remark in this rule that the renaming index *vi* is included in the success path. This ensures that the recomputation process will produce the same substitutions with the same renaming as in the parent task until the point in which the computation is split.

In order to clarify the process, let us consider the example in Figure 3. Its execution process (Figure 5) at this level can be sketched as follows: At the beginning there is only the *initial task* with a single node *root*. This node has as decorated list of goals only one element which is the initial query decorated with the cutpoint *root*. AND_parallelism (&) is ignored at this point and, after setting *act* to *a*, the *continuation* list is assigned the remaining goals in the query. Then the function *procdef* provides the sequence of clauses matching *act*. According to the *Call* rule, depending on the appearance of parallelism, new nodes or tasks are created for each element of the procedure for *act*. In this case the clauses are marked as parallel, and new tasks $T1$ and $T2$ are created. Each of them is provided with a success path which leads them to compute different solutions. Let us consider for instance the process in the first of the new tasks $T1$, which appears in Figure 6. The new task $T1$ has to compute also the initial query. However, its starting status is *recomputation* and thus it follows the given success path until finishing it. During the recomputation process parallelism is not exploited (otherwise solutions would be repeated). When the recomputation

Initial task:

- status = *working*
  success_path = [ ]
  Nodes:
  root
  - $mode$ = Call
    $decglseq = [< [a\&b, c], root >]$
    act = a
    $cont = [< [b, c], root >]$
    $procdef(act,db) = \{a \leftarrow a1, a \leftarrow a2\}$
    New tasks:
    * T1: success_path = $[< a \leftarrow a1, 0 >]$, status = recomputation
    * T2: success_path = $[< a \leftarrow a2, 0 >]$, status = recomputation

    $cand$ = [ ]

    $mode$ = Select
    backtrack
    $stop$ = true

Figure 5: Scheme of the initial execution process.

T1:

- status = *recomputation*
  success_path = $[< a \leftarrow a1, 0 >]$
  vi = 0
  clause = $\{a \leftarrow a_1\}$
  status = *working*
  Nodes:
  root
  - $mode$ = Call
    $decglseq = [< [a1], root >, < [b, c], root >]$
    act = a1
    $cont = [< [b, c], root >]$
    $procdef(act,db) = \{a_1.\}$
    New Nodes: N1(a1)
    $cands$ = [N1]

    $mode$ = Select
    a1 is solved

    $mode$ = Call
    $decglseq = [< [b, c], root >]$
    act = b
    $cont = [< [c], root >]$
    $vi = 0$
    $procdef(act,db) = \{b \leftarrow b_1, b \leftarrow b_2\}$
    New tasks:
    * T3: success_path = $[< a \leftarrow a1, 0 >, < b \leftarrow b1, 1 >]$, status = recomputation
    * T4: success_path = $[< a \leftarrow a1, 0 >, < b \leftarrow b2, 1 >]$, status = recomputation

    $cand$ = [ ]

Figure 6: Scheme of the execution process of a task initialized by recomputation

finishes, the mode of the only node in the task changes to *Call* and the next goal $a1$ in the first element of *decglseq* is considered. After solving it the first goal in the first element of *decglseq* is assigned to *act*. When solving it OR_parallelism appears again and new tasks $T3$ and $T4$ are created. For instance, the success path of $T3$ indicates that when solving the goal a the first clause in the procedure has to be chosen, and the same for the goal b. The process for the remaining tasks follows the same scheme.

The node classification proposed in [Bör95a] can be applied to the nodes of a task. According to this classification a node is

- **visited** if it has already been the value of *currnode*;
- **active** if it is *currnode* or it is on the path from *currnode* to *root*;
- **abandoned** if it has suffered backtracking;
- **candidate** if it belongs to the *cands* list of an active node.

By introducing some modifications to the Börger and Rosenzweig results at this point, it can be established correctness of the task model.

Lemma 4.1 *Given a pure Prolog program and a query, every visited node of a task corresponds to a node of the parallel SLD subtree with the same success_path and the same substitution as the visited node.*

Proof Börger and Rosenzweig [Bör95a] have proved that given a pure Prolog program and a query, every visited node of the Prolog tree corresponds to a node of the SLD-tree with the same substitution. It has been showed by induction over the time of the first visit (number of rule executions preceding). Induction step follows from the Select rule together with the definition of SLD-tree and candidate clause. Since the *Select* rule of [Bör95a] has been maintained in the task tree representation and since theorem 3.1 establishes a one-to-one correspondence between the branches in the SLD tree and those of the parallel subtrees, the same result is fulfilled. Furthermore, considering the *Call* rule, in which the *success path* is constructed as the sequence of clauses chosen to solve the current goal together with the renaming index, and the *Recomputation* rule, in which a given success path is followed, it is also true that the success path of the task is the path of the corresponding node in the SLD tree.

Lemma 4.2 *Given a pure Prolog program and a query, every abandoned node of a task corresponds to a failure node of the parallel SLD subtree with the same success_path as the abandoned node.*

Proof Börger and Rosenzweig [Bör95a] have proved that given a pure Prolog program and a query, every abandoned node of the Prolog tree corresponds to a failed node of the SLD-tree. It has been showed by induction over the abandonment time. Because of theorem 3.1 this result is extended to parallel SLD subtrees.

The previous lemmas allow stating the following:

Task theorem. *Given a pure Prolog program and a query,*
(i) If a task reaches n times states with all_done, then the corresponding parallel SLD subtree has at least the same number of successful branches with the same

substitutions.
(ii) If a task fails without reaching any state with all_done, the parallel SLD subtree (finitely) fails.

This result together with the relation between the SLD tree and the parallel SLD subtrees establishes correctness of the task model wrt SLD resolution.

Counterexamples to *completeness* could be easily found. However, it can be established in a particular case:
Restricted Completeness theorem. *If each branch of the SLD tree corresponds to a different parallel SLD subtree, i.e. when every program clause is annotated with parallelism, then every successful node of a parallel SLD subtree corresponds to a successful task and every failure node of a parallel SLD subtree corresponds to a failure task.*

Proof If every program clause is annotated with parallelism, according to the *Call* rule a different task is created for each branch of the SLD-tree. Thus, the existence of infinite branches in the SLD-tree does not affect the tasks whether they are exploring successful or failure branches.

5 Introducing AND_tasks

Now the exploitation of AND_parallelism is going to be considered. The program may also present AND_parallelism annotations. That is, a set of goals in the body of a clause can be annotated to be executed simultaneously before continuing the execution of the remaining resolvent. Let us define a parallel call as a set of goals annotated to be executed in parallel:

Def 5.1 *A* parallel call *is a pair consisting of a set of consecutive goals (which may be executed in parallel) and the constant* parallel*:*

$$((G_1, \cdots, G_n), parallel)$$

The following clauses represent examples of parallel calls:

$p \rightarrow a, b, ((c, d), parallel), d.$
$q \rightarrow ((e, f), parallel).$

The ASM represents these annotation by means of a new universe and function:

$$PAR\_CALL = GOAL^*$$
$$parcall : GOAL \rightarrow PAR\_GOAL$$

where *parcall* gives the parallel call where the goal is.

The AND_parallelism is going to be modeled by introducing a new universe AND_TASK with functions:

$$goal : AND\_TASK \rightarrow GOAL$$
$$s : AND\_TASK \rightarrow SUBST$$
$$status : AND\_TASK \rightarrow STATUS$$
$$father\_task : AND\_TASK \rightarrow TASK$$

where TASK is a new superuniverse with both kinds of task, with a function to determine the kind of a task and the specification of the type of task of initial_task

$$TASK \supseteq OR\_TASK, AND\_TASK$$
$$type\_task : TASK \rightarrow \{or, and\}$$
$$initial\_task \in OR\_TASK$$

However, AND_TASKs are not elements of the ASM which models the PDP system. AND_tasks do not find solution to the initial query but to subgoals in a parallel call. Thus, the ASM of the PDP system is still that of the OR_tasks, but now these OR_tasks have the capability of performing the unification of a goal (*act*) by creating an AND_task, and thus in parallel. Nevertheless, the superset TASK has been defined in order to unify the rules, since they are similar for both kinds of tasks, and the differences are distinguished by consulting the *type_task* value.

The execution inside an AND_task is now modeled by the ASM SUBNODE.

$$(SUBNODE; assigned\_goal, currnode; father)$$

The only difference with the NODE ASM which models the execution inside an OR_task is that the *root* node is substituted by the parallel node assigned to the AND_task. Thus, the rules of this ASM are those of the NODE machine and in the following refinements only those are presented.

At this point it is necessary to represent the transmission of data among tasks. The time at which a transmission occurs depends on the status of the concerning tasks. The STATUS universe is extended

status: TASK $\rightarrow \{working, reporting, waiting, sleeping, finished,$
$recomputing\}$

As an informal explanation of these status let us consider the description of the evolution of a typical AND_task. It begins in *working* status when its father task finds AND_parallelism. When the computation finishes the task turns to *reporting* status. If there are cands in the task, it changes to sleeping after reporting. Otherwise, it changes to *finished* status.

In this refinement step, the exploitation of OR_parallelism within an AND_task is not considered. Thus, the call rule for an AND_task becomes similar to the one of the sequential case. The exploitation of OR_under_AND parallelism will be considered in the next section.

The **call_and** rule (specific for AND_tasks) is as follows

if *type(t) = and*
  & *status(t) = working*
then if
is_user_defined(act)
& *mode = Call*
then
let *n = length(procdef(act, db))*
extend *NODE* **by** $temp_1, \cdots, temp_n$ **with**
  $father(temp_i) := currnode$
  $cll(temp_i) := nth(procdef(act,db),i)$
  $success\_path(temp_i) := success\_path(father)$
  $cands := [temp_1, \cdots, temp_n]$
endextend
mode := Select

The call rule (for tasks of type *or*) is maintained except for an initial query to check that the type of the task is *or*.

The query success rule is also specific for an AND_task, arising the **success_and** rule

if *type(t) = and* & *all_done*
then *status = reporting*

When AND_parallelism is exploited the execution of the resolvent after the parallel call can not be done until the execution of every goal in the parallel call has been completed. In order to perform this synchronization a new element has been introduced, the *waiting_list*. This is a list of parallel call representations, each element including the number of goals and the status of the task executing each goal in the parallel call .

The transition rules of the task ASM are those of the OR_task ASM extended with the following:

```
if status(t) = working
thenif  is_user_defined(act)
        & mode = Select
   thenif act ∈ goal(snd(hd(waiting_list(t))))
   then status := waiting
   elseif cands = [ ]
      then backtrack
      elseif length(parcall(act)) > 1
      then
         let n = length(parcall(act))
         extend TASK by task_1, ..., task_n with
         father(task_i) := t
         success_path(task_i) := [ ]
         decglseq(task_i) = act + i
         type(task_i) := and
         status(task_i) := working
         waiting_list(t) := [⟨n, (task_1, ..., task_n)⟩|waiting_list(t)]
         endextend
         fst(fst(decglseq)) := fst(fst(decglseq)) - parcall(act)
      else
         let clause = rename(clause(cll(fst(cands))),vi)
         let unify = unify(act,hd(clause))
         if unify = nil
         then cands := rest(cands)
         else
         currnode := fst(cands)
         decglseq(fst(cands)):=
            subres([⟨bdy(clause),father⟩|cont],unify)
         s(fst(cands)) := s ∘ unify
         cands := rest(cands)
         mode := Call
         vi := vi + 1
```

This rule extends the universe TASK with new AND_tasks which are in charge of solving the goals of a parallel call (while OR_tasks solve the initial goal). The rule establishes that if *act* belongs to a parallel call being executed, and therefore it is in the *waiting_list*, the status changes to *waiting*. In other case *act* is executed. It is checked if *act* belongs to a parallel call ($length(parcall(act)) > 1$). If so new AND_tasks are created, the representation of the parallel call is added to the *waiting_list* and the goals in the parallel call are erased from the resolvent. In other case *act* is solved as in the previous *Select* rule.

The next *waiting rule* establishes that a task waiting for the answer of a set of AND_tasks becomes *working* when every of the AND_tasks is reporting. A reporting AND_task changes to status *sleeping* if there are candidate clauses for its goal and its exploration may be requested in case of backtracking. The task changes to *finished* status otherwise.

if *status(t) = waiting*
then **if** $\forall\, t_i \in snd(hd(waiting\_list(t)))$
        $status(t_i) = reporting$
then *status(t) := working*
        let *n := fst(hd(waiting_list(t)))*
        seq $i = 1,\cdots,n$
            $s(t) := s(t) \circ s(t_i)$
            if $cands(t_i) = [\,]$
            then $status(t_i) := finished$
            else $status(t_i) := sleeping$
        endseq

The backtrack operation is also changed to take into account the new situations. If *currnode* belongs to a parallel call and any task collaborating in the computation is *sleeping* (what means with pending candidate clauses), then its status is changed to *working* in order to explore a new candidate.

backtrack $\equiv$ **if** *father = root*
    then *status(t) := finished*
    else *currnode := father*
        if $length(parcall(currnode)) > 1$
        then $task_i := take\_task(waiting\_list(t))$
            if $status(task_i) = sleeping$
            then $status(task_i) := working$
                status(t) := waiting
            else *backtrack*
        else
        mode := Select

where *take_task* takes the first task in the waiting list which is sleeping.

Let us consider again the example in Figure 3, but now let us assume that the first clause for a contains a parallel call:

$$a \leftarrow a1\&d$$

The execution of the parallel call $a1\&d$ is sketched in Figure 7.

T1:

- status = *recomputation*
  success_path = $[< a \leftarrow a1, 0 >]$
  vi = 0
  clause = $\{a \leftarrow a_1\}$
  status = *working*
  Nodes:
  root
  - *mode* = Call
    $decglseq = [< [a1\&d], root >, < [b, c], root >]$
    $act = \mathrm{a1}$
    $cont = [< [d], root >< [b, c], root >]$
    procdef(act,db) = $\{a_1.\}$
    New Nodes: N1(a1)
    cands = [N1]

    mode = Select
    parcall(a1) = {a1, d}
    New tasks:
    * $T_{and}1$: to solve a1
    * $T_{and}2$: to solve d

    $Waiting\_list = [< 2, [T_{and}1(working), T_{and}2(working)] >]$
    $decglseq = [< [b, c], root >]$
    status = *waiting*
    After $T_{and}1$ and $T_{and}2$ are finished
    $Waiting\_list = [< 2, [T_{and}1(finished), T_{and}2((sleeping)] >]$
    status = *working*

    mode = Select
    $\cdots$ and the process continues

Figure 7: Scheme of the execution process when AND_parallelism is exploited

When the goal $a1$ is going to be solved in *Call* mode, it is found out that the goal belongs to a parallel call. Therefore new AND_tasks, which are included in the *Waiting_list*, are created to solve each goal in the parallel call. These goals disappear of *decglseq* and the status changes to *waiting*. When the execution of the AND_task finishes the *Waiting_list* is updated indicating the status of the task (*sleeping* means that there are pending alternatives to solve d), and the status is *working* again.

The state component map is the identity between the new OR_tasks and the previous ones. In order to establish correctness of the rules of this extended OR_TASK model let us name $call_s$ and $call_p$ to the sequential and parallel parts of the call rule of the previous section, being *Select* and *Call* the complete rules of that section. Let us call $Selecta_s$ and $Selecta_p$ the sequential and parallel parts of the current select rule. Then, the rule map, which does not change for the remaining rules, is defined as follows:

$$\mathcal{F}([Selecta_p, waiting]) = [Select, Call, goal]$$
$$\mathcal{F}([call\_and, success\_and]) = [Call_s]$$

The definition of the implicate rules allows stating that $\mathcal{F}$ commutes, giving

Prop 5.1 *The model of 5 is correct and complete wrt that of 4.*

6 Modeling OR_under_AND parallelism

A fundamental point in modeling PDP is the appearance of OR_under_AND parallelism. This event may happen in an AND_task. According to the combination rule, OR_parallelism appearing in an AND_task is exploited by creating new OR_tasks with the appropriate success_path. The set of solutions to a parallel call are explored in a distributed way by creating a new task for each element of the cross product among the solutions to each goal in the parallel call. In this way no synchronization is needed and all annotated parallelism is exploited.

At is was stated in Section 2.1, the *ancestor goal* of a task created by an AND_task is the goal executed by this AND_task. Then the combination rule fixes the solution to the goals on the left of the ancestor goal and combines them with *every* solution of the remaining goals. Accordingly, the model has to distinguish between tasks able to explore parallelism or not. This leads to a new classification of the tasks. The OR_TASK and AND_TASK universes are now refined to distinguish between primary and secondary tasks depending on whether they have a particular path to follow (secondary) or they have to explore every alternative to the assigned goal (primary).

$$
\begin{aligned}
&OR\_TASK \supseteq PRI\_OT, SEC\_OT \\
&AND\_TASK \supseteq PRI\_AT, SEC\_AT \\
&TASK \supseteq PRI\_OT, SEC\_OT, PRI\_AT, SEC\_AT \\
&type\_task : TASK \rightarrow \{pri\_or, sec\_or, pri\_and, sec\_and\} \\
&initial\_task \in PRI\_OT \\
&ancestor\_goal : TASK \rightarrow GOAL \\
&cpe : TASK \rightarrow CPE
\end{aligned}
$$

where informally a secondary task do not exploit OR_parallelism and *ancestor_goal* is a partial function which maps a task to the goal belonging to a parallel call whose OR_parallelism has caused the task. The position of this goal in the parallel call determines the solution to the parallel call which corresponds to the task. A new universe is also introduced: the CPE (cross product environment), used to specify the combination of solutions to a parallel call which corresponds to a task. Each task records its CPE list (cpel).

The call rule changes according to the mechanism explained above. OR_parallelism is not exploited by *secondary AND_TASKs*, which follows a success path already explored leading to the goal to be executed. The first clause do not generated a new OR_task in order to provide a solution to its assigned goal for its parent task. It is specified in the rule that OR_tasks created by another OR_task have the same ancestor goal as its parent task, while if the type of current task is *and* the ancestor goal of the new OR_TASKs is the *act* goal. Then, the rule takes the form:

if *status(t) = working*
then **if** *is_user_defined(act)*
        & mode = Call
    then
    let $n = length(procdef(act,\ db))$
    seq $i = 1,\cdots,n$
        $cl := nth(procdef(act,\ db),\ i)$
        if $i > 1$ & *orparallel(clause(cl))* & *not scopecut(decglseq(act))*&
        $(pri\_or(t) \vee sec\_or(t) \vee pri\_and(t))$
        then
            if $(pri\_or(t) \vee sec\_or(t))$
                createor $task_i$ **with**
                $initial(t, cl, ancestor\_goal(t), task_i)$
                endcreateor
            else
                createor $task_i$ **with**
                $initial(t, cl, act, task_i)$
                endcreateor
        else
            extend *NODE* **by** $temp_i$ **with**
            $father(temp_i) := fnode(t)$
            $cll(temp_i) := cl$
            $success\_path(temp_i) :=$
                    $append(success\_path(fnode(t)), < cl, vi >)$
            endextend
    $mode := Select$

with the mnemonic abbreviations:

createor
    $initial(t, cl, act, task_i)$
endcreateor
$\equiv$ **extend** *TASK* **by** $task_i$ **with**
    $status(task_i) := recomputation$
    $father(task_i) := t$
    $success\_path(task_i) :=$
        $append(success\_path(t), < cl, vi >)$
    $cpel(task_i) := cpel(t)$
    if $pri\_or(t)$ **then** $pri\_or(t) := true$
    else $sec\_or(t) := true$
        $ancestor\_goal(task_i) := act$
        $cpe := last(cpel(task_i))$
        $cpe(act){+}{+}$
endextend

That is, the new OR_task begins recomputing the success_path consisting of the success_path of its parent_task followed by the parallel clause and if the new task is secondary the *cpe* is updated to lead to a new combination of solutions to the parallel call.

The waiting rule is also modified to take the *cpe* into account:

```
if status(t) = waiting
then if ∀ t_i ∈ snd(hd(waiting_list(t)))
          status(t_i) := reporting
then status(t) := working
     let n := fst(hd(waiting_list(t)))
     seq i = 1,···,n
         s(t) := s(t) ∘ s(t_i)
         sp(t) := append(sp,success_path(t_i))
         cpe(t_i) := end(success_path(t))
         if cands(t_i) = [ ]
         then status(t_i) := finished
         else status(t_i) := sleeping
     endseq
```

The working selection rule takes the following form:

```
if status(t) = working
thenif is_user_defined(act)
         & mode = Select
   thenif act ∈ snd(hd(waiting_list(t)))
   then status := waiting
   elseif cands = [ ] then backtrack
      elseif length(parcall(act)) > 1
      then
         let n = length(parcall(act))
         createand task_1, ···, task_n with
            initial(t,act,task_1, ···, task_n)
         endcreateand
         initial(cpe)
         waiting_list(t) := [⟨n, (task_1, ···, task_n)⟩|waiting_list(t)]
         fst(fst(decglseq)) := fst(fst(decglseq)) - parcall(act)
      else
      let clause = rename(clause(cll(fst(cands))),vi)
      let unify = unify(act,hd(clause))
      if unify = nil then cands := rest(cands)
      else
      currnode := fst(cands)
      decglseq(fst(cands)):=
         subres([⟨bdy(clause),father⟩|cont],unify)
      s(fst(cands)) := s ∘ unify
      cands := rest(cands)
      mode := Call, vi := vi + 1
```

where

createand $\equiv$ **extend** $TASK$ **by** $task_1, \cdots, task_n$ **with**
initial(t,act,task$_1, \cdots, task_n$*)*
endcreateand

$status(task_i) := working$
$father(task_i) := t$
$success\_path(task_i) := [\,]$
$decglseq(task_i) = act + i$
$cpel(task_i) := cpel(t)$
if $(pri\_or(t) \vee pri\_and(t) \vee$
$(sec\_or(t)\ \&\ i \geq ancestor\_goal(t))$
then $pri\_and(t) := true$
else $sec\_and(t) := true$
endextend

initial(cpe) $\equiv$ **extend** CPE **by** cpe **with**
seq $i = 1, \cdots, n$
$cpe[i] := nil$
endextend

Let us consider again the example of Figure 3, extended as in Figure 7 (i.e. substituting the first clause for a by $a \leftarrow a1\&d$). Let us assume an AND_task T created to solve b, being b the *ancestor goal* itself. Figure 8 sketches the process. T is a primary AND_task, so it has the capability of creating new OR_tasks

T:

- status = *working*
  type = pri_and
  success_path = $[< a \leftarrow a1, 0 >< [d], 0 >]$
  vi = 0
  clause = $\{b \leftarrow b_1\}$
  Nodes:
  root
  - *mode* = Call
    $decglseq = [< [b, c], root >]$
    act = b
    $cont = [< [b], root >]$
    $procdef(act,db) = \{b \leftarrow b_1, b \leftarrow b_2\}$
    $ancestor\_goal$ = b
    New Node: N' (to solve with $b \leftarrow b_1$).
    New Task: T'
    type = sec_or
    $success\_path = [< a \leftarrow a1, 0 >< d., N_d >, < b \leftarrow b_2, N_b >]$
    status = recomputation
    $\cdots$ and the process continues

Figure 8: Scheme of the execution of combined parallelism

if OR_parallelism is found. T solves b using the first clause in the predicate and finds OR_parallelism. Since b is not on the left of the *ancestor goal* the second clause for b will be explored by a new OR_task T', which is given its corresponding and updated *success path*.

At this point it can be described a mapping $\mathcal{F}$ between the current OR_task ASM and the one of section 5. The state component of the proof map $\mathcal{F}$ is the identity (primary and secondary or_tasks can be identified with a general or_task with a particular set of alternative clauses to explore). In order to prove it let us distinguish between different actions formulated within a rule. Then, in the Working Select (WSelect) rule let us call $WSelect_w$ to the change to waiting status, $WSelect_b$ to the case of backtrack, $WSelect_p$ to the parallel_case, and $WSelect_s$ to the last case (sequential). In $WSelect_p$, let us distinguish between $WSelect_{pp}$ and $WSelect_{ps}$ corresponding to the parts creating a primary and a secondary AND_task respectively. In the working call (WCall) rule, let $WCall_p$ and $WCall_s$ denote the parallel and sequential parts respectively. And in the Select rule introduced in the previous section (*and* tasks) let us call $Select_b$ to the backtrack part and $Select_s$ and $Select_p$ to the sequential and parallel parts respectively.

Then the rule map is homonymous except for

$$
\begin{aligned}
\mathcal{F}([WSelect_{pp}, WSelect_w, Waiting]) &= [Select_p, Select_w, Waiting, Select_b] \\
\mathcal{F}([WSelect_{ps}, WSelect_w, Waiting]) &= [Select_p, Select_w, Waiting] \\
\mathcal{F}([WCall_p, Recomp]) &= [Call\_and, Select_b] \\
\mathcal{F}([WCall_s]) &= [Call_s] \\
\mathcal{F}([WCall_p, WSelect_{ps}]) &= [success\_and] \\
\mathcal{F}([WCall_p, WSelect_{pp}]) &= [success\_and, Select_b]
\end{aligned}
$$

Commutativity of $\mathcal{F}$ with rule comes from these correspondences yielding

Prop 6.1 *The task model of 6 is correct and complete wrt that of section 5.*

7 Introducing Workers and Communications

At this point it is introduced in the model a correspondence among tasks and processors. In order to do this, a new universe *WORKER* is defined.

External functions provide a flexible and open framework to represent the environment in which an ASM is intended to work. Since there is a number of possible scheduling policies, and the choice of one of them does not affect the correctness of the system, but only its performance, the notion of external function is used to model the scheduler of the system. The function scheduler has as arguments the state of the workers of the system as well as pending tasks. A worker may be *idle*, *busy* (working) or *offering* (with pending task). Applying a fix algorithm (exchange of work among closer workers, the oldest work, etc) the scheduler decides which offering worker if going to give a task to which idle worker. This is the output of the scheduling function.

Then, the rules are modified in order to replace the creation of a task by an annotation of the possibility of creating the corresponding task if it can be assigned to a worker.

The creation of an OR_TASK is replaced by *annotating* the corresponding parallel clause in a new list, the *pending_alt* list. This change is reflected in the *Call* rule.

The creation of an AND_TASK is replaced by *pushing* the parallel goal into a new stack, the *goal stack*. A new universe *PENDING_GOAL* is introduced to model the new actions. The Select rule reflects the change.

For the sake of brevity we do not include here the new rules, but they can be found in [Ara96].

Now, it is necessary to introduce mechanisms to deal with the pending AND and OR_tasks. Let us assume that the *scheduler* function is able to detect the change in the state of the workers and to check the goal stack and pending alternative list. With these data the *scheduler* decides the task to be assigned to a worker that has become idle. The creation of the tasks takes place by indication of the scheduler. To model this process a new universe *COMMUNICATION* is introduced as well as the function.

$$com\_state : TASK \rightarrow \{input, output, rest\}$$

By default com_state(t) is assumed to be *rest*. Communications are modeled by external functions *input* and *output*, whose behavior consists in copying the data of the structure *output_men* of a task with *com_state* output to the structure *output_men* of the task assigned to the worker *destiny*. A task with the possibility of creating new tasks does it when the its *input* function adopts the appropriate request value. A new rule [Ara96] is introduced to model the system behavior when *com_state* becomes *input*.

It is necessary to model the behavior of an AND_task, which now in case of failure has to explicitly communicate this result to its parent task. In order to specify the parallel call to which the failed goal belongs, a new data structure is introduced in the creation of the AND_task: the *parent_task_data*. Backtracking has also to take into account that if a goal to be reexecuted belongs to a parallel call and has been solved by an AND_task, the new solution has to be requested to this task, and thus a new *backtracking* rule appears (this can be found in [Ara96]).

In case there is no idle processor in the system, a mechanism is needed by means of which the pending tasks are developed by the current task itself. In the case of the OR_tasks, the pending alternatives are taken automatically in the backtracking process. Then, we only need to update the pending_alt list during the backtracking process. In the case of the AND_tasks the Waiting rule is modified to take goals from the goal stack of the task itself. In order to compute by itself any of these goals, the status universe is extended with a new value *working_inside*. The rule for this state is just as the one for *working* except for the former does not change to *waiting* status when the goal belongs to a parallel call since there are not answers to wait for from other tasks.

The task model of this section turns out to be correct and complete wrt that of 6 as it is shown in [Ara96].

8 Introducing Stacks in the Tasks

In a way similar to the step from Trees to Stack in [Bör95a], the path of active nodes in a task may be viewed as a stack, if *cands* list are represented elsewhere.

The *CODE* universe is refined to *CODEAREA*, which represents sequencing of clauses in a Prolog program, with

$$\begin{array}{l} + : CODEAREA \rightarrow CODEAREA \\ cll : NODE \rightarrow CODEAREA \\ clause : CODEAREA \rightarrow CLAUSE + \{nil\} \\ procdef : LIT \times PROGRAM \rightarrow CODEAREA \end{array}$$

procdef now yields an element of CODEAREA, i.e. a pointer, instead of a list; the old list can easily be reconstructed [Bör95a].

The information contained in *currnode* is separated from other active nodes, by recording the former in 0-ary functions. These are identified with

$$decglseq \quad s \quad cll$$

NODE is renamed to *STATE*, *root* to *bottom*, *father(currnode)* and *father* to 0-ary and unary *b* (for backtracking).

$$b \in STATE \quad b : STATE \rightarrow STATE$$

replacing the previous *NODE* submachine

$$(NODE; root, currnode; father)$$

by the *statetree ASM*

$$(STATE; bottom, b; b)$$

More formally, it will be the mapping $\mathcal{F}$ proposed by Börger and Rosenzweig [Bör95a] the one which maps stack elements to task nodes as:

$$(decglseq, s, cll, b, bottom, vi) \rightarrow (node, root, vi))$$

where the node decorations in the task are recovered from the registers and the decorations of the stack elements as follows:

$$\begin{aligned} decglseq(currnode) &= decglseq \\ s(currnode) &= s \\ cands(currnode) &= mk\_cands(node, cll) \\ father(currnode) &= F(b) \end{aligned}$$

with $F : STATE \rightarrow NODE$ an auxiliary function such that

$$\begin{aligned} decglseq(F(n)) &= decglseq(n) \\ s(F(n)) &= s(n) \\ cands(F(n)) &= mk\_cands(F(n),\ cll) \\ father(F(n)) &= F(b(n)) \\ F(bottom) &= root \end{aligned}$$

where:

$$mk\_cands(Node,\ Cll) \equiv \begin{array}{l} \textbf{if } clause(Cll) = nil \\ \textbf{then } [\,] \\ \textbf{else } [\langle Node, Cll\rangle | mk\_cands(Node, Cll++)] \end{array}$$

This F is the one adopted in [Bör95a] when the state ASM is introduced, but adding the new data structures corresponding to the success path and to the cpe list.

Assuming the representation of data proposed in [Bör95a]

$$(DATAAREA; +, -; val),$$

where

$$\begin{array}{l} +, - : DATAAREA \rightarrow DATAAREA \\ val : DATAAREA \rightarrow PO + MEMORY \end{array}$$

where PO (for Prolog Objects) is a universe supplied with functions

$$type : PO \rightarrow \{Ref, Const, List, Struct, Funct\}$$
$$ref : PO \rightarrow ATOM + DATAAREA + ATOM \times ARITY$$

where $ARITY = 0, \cdots, maxarity$ and MEMORY is a universe containing DATAAREA. It is also assumed (partial) functions

$$deref : DATAAREA \rightarrow DATAAREA$$
$$term : DATAAREA \rightarrow TERM$$

The success path will be represented on the SUCCESS_STACK, a submachine of DATAAREA

$$(SUCCESS\_STACK; sp, bsp; +, -; val)$$

to be used as a stack, with $sp, bsp \in SUCCESS\_STACK$ representing top and bottom.

For the CPE list it is introduced CPE_LIST, with *cpe* and *bcpe* representing the beginning and the end of the list.

$$(CPE\_LIST; cpe, bcpe; +, -; val)$$

The elements of the NODE ASM are recovered as follows

$$\begin{aligned} node(currtask) &= F(b(currtask)) \\ success\_path(currtask) &= F(bsp) \\ cpel(currtask) &= F(bcpe) \end{aligned}$$

These changes are reflected in a new set of rules [Ara96].

As it is stated in [Bör95a], the "stacks" maintain the node tree structure corresponding to a task: they are not discarded when they are "popped" on backtracking, but they are still there and may be used when needed. The structure of visited nodes would be completely preserved if it is possible to establish a complete correspondence with the nodes of a task of the previous section by using F. Assuming $\mathcal{F}$ on rules as homonymy, the commutativity of the rule executions is obtained with $\mathcal{F}$, giving

Prop 8.1 *The stack model of 8 is correct and complete wrt sets of Prolog nodes.*

8.1 Optimization in the creation of choicepoints

In the analysis of [Bör95a], an optimization is introduced at this point of the construction of the stacks. The aim of this optimization is not to create a choice point if the selected unifying clause fails. To do this the signature of the previous section is essentially retained and the action of the select rule is decomposed into more primitive steps, in order to reorganize them more efficiently. These steps are controlled by the 0-ary function *mode*, which now extends its values by decomposing old *Select* mode. This decomposition may be applied directly to the refinement step of the PDP system.

Pushing a choice point will be now carried out either by mode value of *Try* or by *Try_par* if parallelism appears. Attempting unification is performed by mode value of *Enter*. Reusing a choicepoint is invoked either by *Retry* mode, or by *Retry_par* mode in case of parallelism. Old *Call* mode will retain its role.

A new 0-ary function ('cutpoint register') $ct \in STATE$ is introduced. It will, in call mode, store b's old value in order to find it in *Enter* mode. (See [Bör95a, Ara96] for details).

9 Predicates structure: OR_parallelism

The code for the extended WAM which corresponds to OR_parallelism exploitation appears when the disjunctive structure of Prolog predicates is considered. As in [Bör95a], a predicate is represented as a sequence of instructions to manage choicepoints. This leads to slightly modify the signature. *cll* and *clause* are replaced with p (for 'program pointer') and *code*, assuming a special location *start*. It thus results

$$p, start \in CODEAREA$$
$$code : CODEAREA \rightarrow INSTR + CLAUSE + \{nil\} + \{code(start)\}$$

where

$$\begin{aligned} INSTR = \{ & try\_me\_else(N), retry\_me\_else(N), trust\_me(N), \\ & try(N), retry(N), trust(N) \\ & try\_par(N), retry\_par(N) | N \in CODEAREA\} \end{aligned}$$

The INSTR universe has been enlarged with respect the one of [Bör95a] at this point with the parallel instructions. Besides, it will be more enlarged in the sequel, as more WAM and specific PDP instructions are introduced. The operations of previous ASMs in *Try*,*Retry* modes will now be simulated by executing instructions, *try_me_else* or *try* in case of mode *Try*, *retry_me_else*, *retry*, *trust_me*, *trust* in case of mode *Retry*, and *try_par* and *retry_par* in cases of modes *Try_par* and *Retry_par*.

The same remarks of [Bör95a] in the corresponding stage of abstraction allow to establish that the model of 9 is correct and complete wrt that of 8 (see [Ara96] for details).

10 Clause structure: AND_parallelism

The PDP extension to the WAM for the exploitation of AND_parallelism appears when the compilation of clause structure into WAM is analyzed. Following the Börger and Rosenzweig analysis, this section deals only with simplified clause structure (using only instructions for environments and parallel call ((de-)allocation, unification and calling). Thus, terms and substitutions are considered independently and the analysis of [Bör95a] is adopted directly for them.

Viewing *decglseq* as a stack, the previous model may be seen as a stack of stacks, which contains common structures. These stacks contain a number of common copied structures which can be (partially) avoided by sharing common pieces in a new data structure, the *environment*. In general, when a clause is considered a new environment is allocated, containing the data necessary to continue the computation once the goals of the body are solved. Following the scheme of [Bör95a], it is defined a universe ENV with functions

$$\begin{array}{l} cg : ENV \rightarrow GOAL \\ cutpt : ENV \rightarrow STATE \\ ce : ENV \rightarrow ENV \end{array}$$

where cg is the continuation goal, and ce links the environment stack (for *continuation environment*). The role of (0-ary and unary) $decglseq$ will now be taken over by

$$\begin{array}{ll} goal \in GOAL & goal : STATE \rightarrow GOAL \\ e \in ENV & e : STATE \rightarrow ENV \end{array}$$

with $goal$ being the goal component of the list decorated goal of $decglseq$, while its cutpoint and continuation are contained in e.

The AND_parallelism management requires the introduction of further structures. The *waiting_list* is also kept in form of stack. Then the PCE (Parallel Call Environment) universe is introduced, with functions:

$$\begin{array}{ll} pgoal : PCE \rightarrow GOAL & \\ pce \in PCE & pce : PCE \rightarrow PCE \end{array}$$

Then, there are stacks of choicepoints, environments and parallel call environments. They are usually represented in the WAM as *interleaved* on a single stack.

The backtracking process is complicated because of AND_parallelism exploitation. If the failed goal is inside a parallel call, it is known that the backtracking of any other goal inside the parallel call would not change the fail since they are independent. Thus, during backtracking this fact has to be checked. Furthermore, for goals belonging to a parallel call the backtracking process has to distinguish between goals executed by a different task and by the task itself. And in both cases, the process has to distinguish whether the goal has pending alternatives. All these facts are controlled by introducing objects as markers in the stack. This in turn leads to introduce new instructions and to modify the backtracking process (see [Ara96] for details).

The interleaving of the stack is modeled by means of a new superuniverse of states, environments, parallel call environments and a number of markers, with a stack-linking function and a common bottom

$$\begin{array}{l} STACK \supseteq STATE, ENV, PCE, markers \\ - : STACK \rightarrow STACK \\ bottom \in STATE \cap ENV \cap PCE \cap markers \end{array}$$

The proof map to the model of the previous section will be defined by extending the proof map of [Bör95a] at this abstraction level, turning out that the current model is correct and complete wrt to that of 9.

10.1 Compilation of clause structure

As in [Bör95a], a function to produce code for a clause is assumed. But PDP requires also instructions corresponding to the management of the parallel call:

$$\begin{aligned} compile &: CLAUSE \rightarrow INSTR^* \\ compile(H : -G_1, \cdots, G_n) &\equiv [\, allocate, unify(H), \\ & \quad call(G_1), \cdots, call(G_n), \\ & \quad allocate\_pcall(n), pcall(G_1), \cdots, pcall(G_n), wait, \\ & \quad popgoal, deallocate, proceed] \end{aligned}$$

where the universe INSTR is extended to contain the new instructions *allocate_pcall* (which creates a frame to control the execution of a parallel call), *pcall* (which prepares a goal for parallel execution), *wait* (which waits for the answer to goals executed by other tasks) and *popgoal* (which executes a local goal) for the exploitation of AND_parallelism.

The state component of the proof map $\mathcal{F}$ proposed in [Bör95a] at this level is extended here (see [Ara96] for details) yielding correctness and completeness of the model of 10.1 wrt that of 10.

11 Completing the model

The next step in the process is to take into account the representation of terms and substitutions. This leads to introduce new universes that are submachines of DATAAREA. These new universes lead to the appearance of the Heap and the Trail, as well as the *putting* and *getting* instructions. Since the remaining refinement steps consist in the direct application of the transformation of [Bör95a] to the PDP model, I refer to this work for those steps.

12 Conclusions

This work provides a specification and proof of correctness for the system PDP (Prolog Distributed Processor), as well as for the abstract machine designed for it, by means of ASMs. The proof takes advantage of the WAM proof of correctness, in spite that the starting point for the process cannot be the same because the sequential execution model is not complete wrt the parallel one. Thus, the first step of this process consists in defining *parallel SLD subtrees*, which are a kind of partition of the SLD tree for programs whose clauses are annotated with parallelism. In a second step the parallel execution approach of PDP is modeled by means of an *OR_TASK* ASM. In this machine each task is associated with the execution of a parallel SLD subtree, and the model is proved to be correct wrt the previous one. Completeness is restricted to the special case in which each branch of the SLD tree corresponds to a different parallel SLD subtree. The execution of the parallel SLD subtree corresponding to each task is modeled by a *NODE* submachine which extends the one proposed by Börger and Rosenzweig [Bör95a] to model the sequential execution of Prolog. In this way the result of this work allows to avoid the verification of common points. Several of the following steps reproduce the scheme developed in [Bör95a], though introducing at each level the objects required to manage parallelism.

Thus, the appearance of AND_parallelism leads to introduce another kind of task, the AND_task which, instead of solving the initial goal, solves a goal in the body of a clause. OR_parallelism explicitly appears when the predicate structure is considered, which results in the introduction of elements to model the success path corresponding to the execution of a new solution, as well as the OR_parallel instructions. Similarly, the explicit expression of AND_parallelism appears in the analysis of the clause structure. This process leads to the WAM extension which underlies the architecture of PDP. Communication and scheduling are modeled as external functions.

The scheme we have followed to verify PDP can also be applied to other parallel systems. Thus, the RAP model [Her86], whose extension for distributed memory systems, has been adopted to exploit AND_parallelism in PDP, can be verified as a particular case of PDP when OR_parallelism does not appear. Likewise, verification of other OR_parallel systems can make profit out of the first steps of the verification of PDP, since they share the relation between OR_parallel computations and the SLD tree. Furthermore, OR_parallel systems using independent working environments, such as MUSE [Ali90], can be verified in a way similar to PDP, by simply replacing the *recomputation* rule by a *copying* rule, since copying is the mechanism used in this system to reconstruct a working environment. Finally, systems which combine both kinds of parallelism, such as ACE [Gup93], can also take advantage of a number of steps in the verification of PDP.

Acknowledgement

I would like to thank the anonymous referees for making many useful comments and suggestions which have helped to improve the paper. I would also like to thank Jose Cuesta for many valuable comments regarding the writing of this paper and for his support. This work has been supported by the project TIC95-0433.

References

[Ali90] Ali, K. A. M., Karlsson, R.: "The Muse Approach to Or-Parallel Prolog"; Int. Journal of Parallel Programming 19, 2 (1990), 129-162.

[Apt90] Apt, C.: "Logic Programming"; Handbook of Theoretical Computer Science (J. van Leeuwen ed.), Elsevier (1990).

[Ara93] Araujo, L. Ruz, J.J.: "OR-Parallel Execution of Prolog on a Transputer-based System"; *Transputers and Occam Research: New Directions.* IOS Press (1993), 167-181.

[Ara94] Araujo, L., Ruz, J.J.: "PDP: Prolog Distributed Processor for Independent_AND/OR Parallel Execution of Prolog"; Proc. Int. Conf. of Logic Programming, MIT Press (1994), 142-156.

[Ara96] Araujo, L.: "Correctness proof of a Parallel Implementation of Prolog by means of Evolving Algebras"; Technical Report DIA 21-96, Dpto. Inform'atica y Autom'atica, Universidad Complutense de Madrid, (1996).

[Ara97] Araujo, L., Ruz, J.J.: "A Parallel Prolog System for Distributed Memory"; The Journal of Logic Programming, 33, 1 (1997), 49-79.

[Bei96] Beierle, C., Börger, E.: "Specification and correctness proof of a WAM extension with abstract type constraints"; Formal Aspects of Computing, 8, 4 (1996), 428-462.

[Bör93] Börger, E., Riccobene, E.: "A Formal Specification of Parlog"; Semantics of Programming Languages and Model Theory (M. Droste, Y. Gurevich, Eds.), Gordon and Breach (1993), 1-42.

[Bör94a] Börger, E.,Durdanovic, I., Rosenzweig, D.: "Occam: Specification and Compiler Correctness" (E.-R. Olderog, Ed.), Proc. PROCOMET'94 (IFIP Working Conference on Programming Concepts, Methods and Calculi), North-Holland (1994), 489-508.

[Bör94b] Börger, E., Lopez-Fraguas, F.J., Rodriguez-Artalejo, M.: "A model for mathematical analysis of functional logic programs and their implementations"; Proc. World Computer Congress, North-Holland (1994), 410-415.

[Bör94c] Börger, E., Glässer, U.: "A formal Specification of the PVM Architecture"; Proc. IFIP 13th World Computer Congress, Volume I, Elsevier, (1994), 402-409.

[Bör95a] Börger, E., Rosenzweig, D.: "The WAM – Definition and compiler correctness"; Logic Programming: Formal methods and Practical Applications. Beierle, C., y Plümer, L. eds. North-Holland Series in Computer Science and Artificial Intelligence (1995), 21-90.

[Bör95b] Börger, E., Salomone, R.: "CLAM specification for provably correct compilation of CLP(R) programs"; Specification and Validation Methods (Börger, E. eds.), Oxford Univ. Press (1995), 97-130.

[Bör96] Börger, E., Durdanovic, I.: "Correctness of Compiling Occam to Transputer Code"; The Computer Journal, 39, 1 (1996), 52-92.

[Bör97] Boerger, E., Mazzanti, S.: "A Practical Method for Rigorously Controllable Hardware Design"; (Bowen, J.P., Hinchey, M.G., Till, D., Eds.), ZUM'97: The Z Formal Specification Notation, Springer LNCS 1212 (1997), 151-187.

[Gup93] Gupta, G., Hermenegildo, M., Costa, V.S.: "And-Or parallel Prolog: A recomputation based approach"; New Generation Computing, 11, 3 (1993), 297-322.

[Gur88] Gurevich, Y.: "Logic and the challenge of computer science"; Currents trains in theoretical computer science, (Börger, E. eds.), Computer Science Press (1988), 1-57.

[Gur89] Gurevich, Y., Moss, L.S.: "Algebraic Operational Semantics and Occam"; CSL'89, Lecture Notes in Computer Science 440, Springer-Verlag (1990), 176-192.

[Gur91] Gurevich, Y.: "Evolving Algebras. A tutorial introduction"; Bulletin of the European Association for Theoretical Computer Science, 43, (1991).

[Her86] Hermenegildo, M.: "An abstract Machine Based Execution Model for Computer Architecture Design and Efficient Implementation of Logic Program in Parallel"; PhD thesis, U. of Texas at Austin (1986).

[Warr83] Warren, D.H.D.: "An Abstract Prolog Instruction Set"; Tech. Note 309, SRI International (1983).