CONCURRENT PROLOG IN A MULTI-PROCESS ENVIRONMENT

Rosanna K.S. Lee
Department of Computer Science
University of Waterloo
Waterloo, Ontario, Canada
N2L 3G1

Research Report CS-84-46
November 1984

CONCURRENT PROLOG IN A MULTI-PROCESS ENVIRONMENT†

Rosanna K. S. Lee

Logic Programming and Artificial Intelligence Group
Department of Computer Science
University of Waterloo
Waterloo, Ontario, Canada N2L 3G1

ABSTRACT

Concurrent Prolog is Shapiro's definition of a language which is a simple yet powerful transformation of Prolog that incorporates concurrency, communication, synchronization and indeterminacy. A computation model which uses processes communicating via message-passing was defined and a prototypical interpreter was developed on *Waterloo Port*‡ — a multi-process programming environment that supports concurrent activities. The model defines the process structure, communication paths, binding environment, and synchronization procedures. The interpreter is a partial implementation of the model. Some measurements were made to evaluate the performance of the interpreter.

†This report was originally submitted as a Master's thesis to the Department of Computer Science, Faculty of Mathematics, at the University of Waterloo by the author.

‡ Port and Waterloo Port are trademarks of the University of Waterloo.

Acknowledgements

This thesis would not have been possible without the guidance and encouragement of my supervisor, Prof. Randy Goebel.

I would like to thank Patrick Chan for many useful discussions during the preparation of this thesis and development of the interpreter.

I would like to thank R. Vasudevan for several valuable suggestions regarding the performance aspects of the interpreter.

I would like to thank Prof. Maarten van Emden for providing some helpful comments on this theisis.

I would also like to thank the members of the Software Portability Group for developing and supporting such a productive research environment.

I am grateful to the Department of Computer Science at the University of Waterloo, and the Natural Sciences and Engineering Research Council of Canada for financial support.

Table of Contents

List of Figures

1

Introduction

1.1 Motivation

A most appealing characteristic of logic programs is their inherent admissibility of parallel execution. Substantial effort has been expended exploring this potential. One of these efforts is Shapiro's definition of the language *Concurrent Prolog* — a simple yet powerful transformation of Prolog that incorporates "concurrency, communication, synchronization and indeterminacy."†

A subset of Concurrent Prolog has been implemented in Prolog [Shapiro 83]. Although it is simple and elegant, the interpreter suffers the same limitations as Prolog. This implies speed hindrance (due to the extra level of interpretation) and unattainable "true" parallelism. Moreover, the interpreter's role as a simulator for developing and debugging Concurrent Prolog programs is limited as it uses much of Prolog's deterministic control strategies, instead of strategies that can properly support Concurrent Prolog's indeterminacy and parallelism. A program that executes correctly on the simulator may fail to do so on a truly concurrent interpreter because the program depended on, for example, the sequential selection of clauses.

This thesis examines the issues surrounding the design and implementation of a concurrent Prolog interpreter, *Port‡ Prolog*, in a message-based multi-process environment. By constructing the interpreter outside a Prolog environment, the difficulty of implementation is drastically increased. For example, instead of having Prolog's binding environment handle shared variables, the binding mechanism of variables (shared or otherwise) must be restructured. However, an independent interpreter can improve execution speed by eliminating the intermediate Prolog system. Furthermore, an interpreter that makes use of the distributed facilities of a multi-process environment has the potential of realizing true parallelism. Without multiple processes which work cooperatively in accomplishing a task (the solving of a Concurrent Prolog goal, in this case), we would have to resort to a sequential approach which, at best, can only simulate the parallelism offered by Concurrent Prolog.

1.2 Objectives

This research investigates the feasibility of employing message-based processes to build a Concurrent Prolog interpreter. By capitalizing on an environment which supports processes, concurrency and networking, we wish to implement an interpreter which is truly parallel.

Another objective is to succinctly define and validate the workability of Shapiro's informal description of a distributed Concurrent Prolog machine. The design of Port Prolog is based on this description, rather than his sequential implementation which depends on a centralized processor. The distributed machine proposal may incorporate any number of processors.

† [Shapiro 83], page 9.
‡ Port and Waterloo Port are trademarks of the University of Waterloo.

1.3 Organization of Thesis

Chapter 2 studies the relationship between logic programming and parallelism, focusing on the problems of parallelism, and examples of how proposed systems have addressed these problems. We assume that the reader has at least a working knowledge of Prolog [Kowalski 82, Clocksin & Mellish 81].

Chapter 3 focuses on the parallelism aspect of several logic programming languages. We first concentrate on language features defined for handling parallelism. We then present a survey of execution models of parallel logic programming system to demonstrate how such systems are designed.

Chapter 4 describes Port Prolog's model of computation. The chapter begins with a description of the communication primitives which are used to define the Port Prolog interpreter. We then articulate the distributed Concurrent Prolog model by proposing a more precise process structure and by defining the communication paths, variable bindings and synchronization procedures.

Chapter 5 describes the syntax and structure of Port Prolog programs and the interpreter's user interface. We first provide a summary of the Waterloo Port implementation environment.

Chapter 6 describes the implementation of Port Prolog. We start by giving an overview of the process structure of the interpreter. This is followed by discussions of the major components of the interpreter.

Chapter 7 presents some performance measurements of the prototype. We then compared these measurements with those from a sequential Prolog interpreter on Port.

Finally, a summary of the work is presented and some suggestions for future research are discussed in Chapter 8.

2

Parallelism in Logic Programs

"Statements" in logic programs, unlike those in procedural languages such as Pascal, have no explicit ordering to them. The separation of control from the program's logic admits flexibility to how logic programs can be executed. It is this inherent flexibility of execution which permits parallelism in logic programs.

Conery and Kibler classified four types of parallelism in logic program execution: *AND*-parallelism, *OR*-parallelism, *stream* parallelism, and *search* parallelism [Conery & Kibler 81]. In this chapter, we explain the difficulties of realizing them in practical implementations.

2.1 AND-parallelism

Solving literals in a clause simultaneously is referred to as *AND-parallelism*. One difficulty of implementing AND-parallelism arises from the sharing of variables. For example, the goal

$$father(\ Child, Father\ )\ \&\ mother(\ Child, Mother\ )$$

requires that processes executing the two literals coordinate and agree upon the binding of *Child*. *father*(*robert*, *joseph*) and *mother*(*dan*, *mary*) are inconsistent answers because *Child* cannot be both *robert* and *dan* at the same time. Consistency is only one aspect of the problem introduced by shared variables; timing may also be a factor in some situations. For example, the order in which *father* binds *Child* and *mother* binds *Child* may be important in some circumstances. We discuss this timing problem in more detail in Sections 2.1.1 and 2.1.2 in this chapter. The concurrent execution of literals that share variables is called *dependent* AND-parallelism whereas the concurrent execution of literals in a clause which *do not* share variables is called *independent* AND-parallelism.

2.1.1 Execution Order

Another difficulty in realizing AND-parallelism is determining the order in which to execute literals in a clause. Sequential Prolog always chooses literals from left to right. However, the "left-to-right" rule is inappropriate as literals are executed simultaneously in a parallel interpreter. When variables are shared, the instantiation of these variables often depends on the evaluation order of the literals that share them. At the syntactic level, the order is immaterial; however, synchronization, either in the form of execution order or special dynamic control operations, is necessary for the correct evaluation of a clause of literals. For example, evaluation of the goal

$$get(\ X\ )\ \&\ put(\ X\ )$$

is meaningful only when $get(\ X\ )$ be executed before $put(\ X\ )$. This requires either an execution order that ensures this order, or some form of special control that only allows $put(\ X\ )$ to be executed when X is instantiated.

Predicates with side-effects also require specific execution orders. This is discussed in Section 2.5 in this chapter. Most existing designs extend the logic programming language with control operations that relegate the task of controlling execution to the user. The control operations in all of the surveyed implementations are specified with special symbols embedded in a clause. We shall see some examples of such extensions in the next chapter, when we survey some parallel logic programming languages.

2.1.2 Handling Multiple Solutions

Another implementation problem of AND-parallelism is the possibility of multiple solutions for a set of goals. For example, for each literal, there may exist several clauses which can be successfully executed to yield results. Some of these results may be later rejected if they are inconsistent with other results. A sequential interpreter *backtracks* to examine the results one at a time. There are several approaches that a parallel interpreter may take to solve this problem.

One solution is to gather *all* consistent sets of bindings at each stage of execution. Bindings to a group of literals are obtained by performing a massive *JOIN* operation on the bindings found for each literal [Date 77]. For example, consider the problem of obtaining values for X from the literals

$$fruit(\ X\ )\ \&\ red(\ X\ )$$

given the following facts:

fruit(*apple*)	*red*(*car*)
fruit(*orange*)	*red*(*apple*)
fruit(*cherry*)	*red*(*cherry*)
fruit(*lemon*)	*red*(*radish*)
fruit(*pear*).	

A JOIN on the values of X obtained by solving *fruit*(X) and *red*(X) separately results in $X = apple$ and $X = cherry$. Although simple to conceptualize and implement, this approach has some serious disadvantages. Because solving each literal involves finding all possible answers, much effort can be wasted computing instances which cannot be joined with those generated by other literals. Furthermore, the JOIN operation is expensive in terms of time and space.

Another approach is similar to the backtracking solution of sequential interpreters: generate answers for a literal one at a time and *retry* only upon request. At each stage of execution, a group of literals has at most one set of bindings. For the above example, *fruit*(X) would have the binding $X = apple$ and *red*(X) would have the binding $X = car$. If the union of two groups of literals introduces inconsistencies among bindings, action must be taken to produce a set of bindings agreed upon by both groups. This action usually involves finding a new solution to one (or more) of the literals in either (or both) group(s). In the above example, the union of *fruit*(*apple*) and *red*(*car*) is inconsistent as X cannot be both *apple* and *car* at the same time. Hence, we must either discard *apple* and try to solve *fruit*(*car*) or discard *car* and try to solve *red*(*apple*). Several levels of execution may have to be repeated before a consistent set of bindings is found. Another drawback to this approach is the horrendous bookkeeping that is required. State information about each literal in the proof tree is necessary in order for the retry procedure to work correctly. For example, the interpreter needs to keep track of which clauses have not been tried yet, so that ones that failed will not be re-tried. Also, the strategy for selecting literals to retry is non-trivial. Of course, if the system is willing to sacrifice parallelism, it can use backtracking to simplify the bookkeeping and selection process: simply solve literals using the left-to-right rule and retry the most recent literal that caused the binding inconsistency. Although this method does not promote parallelism, it is often adopted by systems (e.g. PRISM [Kasif et al. 83]) that support only independent AND-parallelism but still wish to correctly execute dependent literals using existing facilities.

A third method for handling the possibility of multiple answers is to eradicate the problem at its source. The system, with the help of user specified control operations, always eliminates all but one solution at each stage of execution so that if that solution proves to be inconsistent, the literal has no other answers and fails [Shapiro 83, Clark & Gregory 81]. This is analogous to the use of the *cut* operation in sequential Prolog, except in this case, the *cut* is applied to every clause. See the next section for a more detailed look at the implications of the *cut* operation.

2.2 OR-parallelism

With *OR-parallelism*, clauses with the same predicate name and arity are invoked concurrently. For example, the procedure to append two lists,

append([], X, X).

append([A | B], C, [A | D]) ← *append*(B, C, D).

would invoke two processes, one for each clause. Because of the potential exponential fanout of the search tree, OR-parallelism typically exhausts system resources very rapidly. Also, if unrestrained, the system wastes time following execution branches which obviously lead to failure nodes.

Some implementations have incorporated OR-parallelism into their interpreters, though only a few have considered controlling OR-parallel activities. Ciepielewski and Haridi have reduced this controlling problem to that of "(1) controlling the traversal of the search tree and (2) pruning some branches of the search tree"† with (1) being handled by an appropriate scheduling algorithm and (2) being solved by adopting a richer control language. A popular solution to (2) uses ideas borrowed from the guard concept of Hoare's Communicating Sequential Processes notation [Hoare 78]. This solution is discussed later in the chapter on **Parallel Logic Programming Languages**.

Methods for controlling OR-parallelism are "clause selection" strategies. In Prolog, clauses with the same predicate name and arity are sequentially selected (as they are ordered in the program) until one satisfies the goal. Based on this control policy, some non-logical operations have been defined in Prolog to help prune the search space and to make Prolog more "usable." These non-logical operations introduce problems when we migrate towards a parallel selection mechanism. The problems discussed in the following subsections have been presented in the literature, most notably in [Shapiro 83] and [Haridi & Ciepielewski 83].

2.2.1 Cut

The *cut* operation, which stops the interpreter from choosing alternatives that have already been tried (i.e. backtracking), is more than simply an efficiency mechanism. In many Prolog programs, it affects the program's logic. Eliminating the *cut* operation from some programs could render them "incorrect" — that is, the functionality of the program is altered. *Cut* is not applicable in a parallel logic programming language because all choices for a literal are tried simultaneously. There is, however, an analogous construct in some parallel logic languages which serves to prune the proof tree and to prevent backtracking. This construct is the *guard* and is discussed in the next chapter.

2.2.2 Negation-as-failure

An example of the importance of *cut* and the selection order of clauses in Prolog is its role in the definition of the meta-level function *not*(X).

not(X) ← X, *cut*, *fail*.

not(X).

To prove the negation of a goal, say X, we first try to prove X; if X succeeds, then *not*(X) fails. Negation is only a specific example of the dependence of certain styles of Prolog programming on the selection order of clauses. Typically, a "catch-all" clause is placed as the last clause in a group of clauses with the same name and arity to ensure that the clause invocation always succeeds (or always fails). In parallel implementations, special primitives for negation or mechanisms for controlling the clause selection process (i.e. OR-parallelism) are ways of achieving the "negation-as-failure" effect. Examples of these mechanisms are given in the next chapter.

† [Ciepielewski & Haridi 83], page 536.

2.3 Stream Parallelism

Using a list or structured data as it is being generated is referred to as *stream parallelism* or *eager evaluation*. For example, if a list is shared by several literals, these literals can begin working on the list's elements as soon as they become available. A good application of stream parallelism is given by Conery and Kibler: "one might begin testing for membership in a list while the list was being constructed."‡ To make use of this feature, the interpreter must support AND-parallelism or co-routining; otherwise, stream parallelism is useless because all literals are solved in their entirety before other literals have a chance to execute.

The implementation of stream parallelism depends heavily on how the binding of variables is done and how concurrently executing literals are synchronized. The user may control stream parallelism by specifying, with the use of special notations, how variables can be instantiated. For example, by *annotating* variables, the user can fix the literal containing the variable to be either the producer or consumer of values for the variable. Annotation of variables is explained in more detail in the next chapter when we describe the language features of some parallel logic programming languages.

2.4 Search Parallelism

A potential bottleneck of program execution is the task of searching for clauses. *Search parallelism* refers to the ability to concurrently search the database of clauses. This can mean searching for clauses with the same name or those with different names. The availability of search parallelism is especially useful for interpreters that support OR-parallelism. The time gap between the invocation of the first clause found in the database and the last can be greatly reduced, thus speeding up the start-up time of OR-parallelism.

Search parallelism may be achieved by dividing the database into separately manageable components and initiating searches simultaneously on each component. We consider this problem to be in the realm of database theory. Its contribution to logic programming language implementation is efficiency.

2.5 Side-effects

Non-logical built-in system predicates such as *read, write, add_axiom,* and *delete_axiom* introduce problems for parallel logic programs because they cause permanent side-effects. The parallel logic programming systems which we have surveyed (proposed or implemented) do not support *add_axiom* and *delete_axiom*. Race problems due to the asynchrony of concurrently executing goals using these built-in predicates is the major concern. For example, the timing of an *add_axiom* is vital to the correct execution of the program because of its global effects. The same is true for *read, write* and *delete_axiom* predicates.

Consider a Prolog program that toggles a bit. If the bit is on, the procedure turns it off, and vice versa. Suppose the following are in the database:

toggle_bit ← *bit*(1), *delete_axiom*(*bit*(1)), *add_axiom*(*bit*(0)).

toggle_bit ← *bit*(0), *delete_axiom*(*bit*(0)), *add_axiom*(*bit*(1)).

bit(1).

OR and AND parallelism are the control strategies. Suppose the goal *toggle_bit* is evaluated. Because of AND-parallelism, the three literals in each clause are evaluated concurrently; and because of OR-parallelism, the two clauses are evaluated simultaneously also. The first clause may fail (even though it should succeed) if *delete_axiom*(*bit*(1)) is executed before *bit*(1). The second clause may succeed (even though it should fail) if *add_axiom*(*bit*(0)) of the first clause is evaluated before *bit*(0). Also, there is the possibility of both clauses succeeding in which case *bit*(1) and *bit*(0) will both be in the database — a situation unexpected, unintended and probably unwanted by the programmer. The programmer must be aware of these problems when he is working with a parallel logic programming system.

‡ [Conery & Kibler 81], page 166.

2.6 Summary

We have described four types of parallelism in logic programs and with each type, identified implementation considerations. Parallelism makes irrelevant the control of execution; however, a certain amount of control is desirable and sometimes necessary. Hence, most implementations extend the definition of the language to include control primitives.

The problem of shared variables introduces many fundamental problems in the realization of AND-parallelism. This, as we shall see, makes binding environments very difficult to define in parallel logic programming implementations. Although OR-parallelism is more straight-forward to implement, special operations for controlling it are still necessary for practical use. Stream parallelism can potentially increase the degree of AND-parallelism in the execution of logic programs by making available elements of shared structured data before all elements of that data are instantiated. A useful feature for logic programming systems is the use of search parallelism to reduce the search time for clauses in a database. Predicates which produce side-effects must be used prudently because through these predicates, one branch of execution can violate the integrity of other branches.

In the following chapter, we survey some logic programming languages which incorporate these types of parallelism and examine how they deal with the associated problems.

3

Parallel Logic Programming Languages

In this chapter, we focus on the parallelism aspect of several logic programming languages. We first concentrate on their language features which are defined particularly for handling parallelism. We then present a survey of execution models of parallel logic programming systems to illustrate their designs.

3.1 Language Features

Parallel logic programming languages are usually modeled after Prolog. These parallel languages adopt Prolog-like syntax and semantics for defining programs but employ different control strategies. A parallel logic language can be defined without altering any of Prolog's syntax; by replacing the left-to-right depth-first execution order of Prolog by one that uses *AND*-parallelism and *OR*-parallelism, we have a parallel logic programming language. This conceptually simple extension is advantageous because it eliminates the user's burden of understanding how the syntax affects the control and, consequently, the result of his program. However, such extensions are usually not done without some seemingly innocent but fundamental alterations. For example, by introducing a notation to prune the search tree (such as a "guard"), the completeness of the interpreter may be affected — some solutions which existed had the notation not been installed may be never found.

To solve the control problems described in the previous chapter, the most general solution has been to devise control operations which give the programmer control over the order in which literals are executed. In this section, we give a brief overview of some parallel logic programming languages, concentrating mainly on their syntactic and semantic aspects rather than their computation models.

3.1.1 PRISM

PRISM is a "Parallel Inference System for Problem Solving" being developed at the University of Maryland [Kasif et al. 83, Eisinger et al. 82]. It is constructed on the ZMOB multiprocessor machine (also built at the University of Maryland) and makes use of the many processors to achieve real parallelism. It supports independent AND-parallelism, OR-parallelism, and search parallelism. The PRISM language provides the user with notations for controlling the execution order of literals (AND-parallelism) as well as the selection order of clauses (OR-parallelism).

There are two notations for controlling the execution of literals. Literals enclosed by parenthesis (...) are executed left to right whereas literals enclosed by square brackets [...] are executed in parallel. For example, the clause definition

$$a \leftarrow (b,\ c,\ [d,\ (e,\ f)]).$$

requires that b be executed before c and c before $[d, (e, f)]$; d and (e, f) can be executed in parallel with e being solved before f. These two notations can be omitted in which case a default strategy is used; "left-to-right" and asynchronous are the two choices of default available.

The specification for controlling OR-parallelism is "static" — the clause selection order is given at the time the clauses are defined and not during execution. Clauses with the same name and arity are numbered in accordance with their selection order. Lower numbered clauses have higher priority than those with higher enumerations. For example, the definitions

1 $a \leftarrow b.$

1 $a \leftarrow c.$

2 $a \leftarrow d.$

*3 $a \leftarrow e.$

permit the first two clauses to be executed simultaneously. The third clause may be tried along with the first two or after the first two. The asterisk on the fourth clause forces the first three clauses to be tried before the fourth is attempted. The "negation-as-failure" programming technique may easily be simulated by forcing the interpreter to adopt a sequential clause selection scheme.

3.1.2 IC-Prolog

IC-Prolog is Imperial College's version of Prolog, which, among other features, permits more control over how a program can be executed. It offers much more sophisticated methods for controlling parallelism than PRISM. IC-Prolog programs resemble Prolog programs with the exception of some control notations. There are two annotations used with variables: the *input* "?" and *output* "^" variable annotations. The input annotation (e.g. $X?$) specifies that the annotated variable (X) must be bound before execution is carried any further. The output annotation (e.g. $X\hat{}$) permits clause invocation only when the variable being annotated (X) is uninstantiated. Note that clause selection is suspended until these annotations are satisfied (or failed); no other clause (with the same name and arity) may be tried until the annotations are resolved.

Literals in a goal can be joined by either sequential "&" or concurrent "//" conjunction symbols. For the goal,

$$A \ \& \ B \ // \ C,$$

"A & B" and *"C"* are executed concurrently with B being started only when A has been successfully completed. This conjunction notation is functionally equivalent to that used in PRISM, and is less cumbersome.

A clause in IC-Prolog has the following syntax:

$$P \leftarrow G : A_1 \ \& \ \cdots \ \& \ A_m.$$

The *guard* primitive ":" makes the execution of the literal G indivisible with the unification of the head P — the bindings in P are not passed back until after G has been successfully completed. The guard does not prevent backtracking of P or G; if P or G fails, their alternatives can still be tried.

Programs may execute in different "modes" according to their use of the control directives. Some examples of execution modes are unsynchronized parallel execution, parallelism with direct communication, and data-triggered co-routining. In unsynchronized parallel execution, the "&" for conjunction is replaced by "//". Basically, literals are evaluated asynchronously; if a literal binds some shared variables, then other literals are signaled to check the bindings for consistency. In parallelism with direct communication, only one literal is allowed to generate bindings for a specified shared variable — the one that annotated the variable using "^". This creates a *producer-consumer* relationship, wherein each time the producer updates the shared variable, consumers are reactivated in order to record the new binding. This is more controlled than unsynchronized parallelism; in the unsynchronized case, *any* literal can perform the binding. Data triggered co-routining is different from the direct communication case in that there is no "forking."† When shared variables are bound, producers are suspended until the consumers are resuspended. Syntactically, goals in parallelism with direct communication are conjoined using "//" whereas "&" is used for data-triggered co-routining.

3.1.3 The Relational Language for Parallel Programming

Most notations in IC-Prolog are present in the Relational Language; however, the semantics of the Relational Language are significantly different enough to warrant its own discussion.

Clauses in a program have the following format:

† In parallelism with direct communication, the producer keeps on executing even after it has bound a variable; it need not wait until the consumer is resuspended.

$$P \leftarrow G_1 \,\&\, \cdots \,\&\, G_k \mid S_1 \,//\, \cdots \,//\, S_n.$$

Like IC-Prolog, "&" denotes sequential execution and "//" denotes concurrent execution. The *clause bar* "|" is similar to the guard ":" of IC-Prolog. It makes the execution of the *guard sequence* $(G_1 \,\&\, \cdots \,\&\, G_k)$ and the unification of the head P indivisible. However, unlike IC-Prolog, it also prevents backtracking once evaluation has reached the clause bar (like the *cut* in Prolog). In other words, the first clause to successfully execute its guard sequence becomes the *candidate* clause and all other alternatives are eliminated. In IC-Prolog, clause selection is sequential. The clause bar can be omitted if no guard sequence is desired, in which case unification of P acts as the guard. Each S_i that makes up the *goal sequence* $(S_1 \,//\, \cdots \,//\, S_n)$ consists of sequentially ordered literals $(S_i = A_1 \,\&A_2 \,\&\, \cdots \,\&\, A_m)$. All S_i's are started simultaneously (after the guard sequence has been successfully executed) and are synchronized with the use of *modes* and annotated variables. The variable annotations ("?" and "^") are identical with those in IC-Prolog. Mode declarations, however, are more general than variable annotations. For example, if we know that *append* always takes two lists as input to its first two arguments and return a third list as output, we can use the mode declaration

mode append(?, ?, ^)

without specifically annotating the variables in the head of the *append* clauses. In this way, *append* is not invoked until it has the first two arguments instantiated and the last one free.

Clark and Gregory point out two major differences between IC-Prolog and the Relational Language [Clark & Gregory 81]. The first is that, due to the semantics of the guard and the fact that every clause must have a guard, there is no backtracking in the Relational Language. Only *shallow backtracking*, wherein clauses whose guard sequence fail are discarded, is present. The other major difference is that the OR-parallelism in the Relational Language requires all clauses with the same head predicate and arity to be tried simultaneously. In IC-Prolog, clause selection is sequential.

3.1.4 Concurrent Prolog

Concurrent Prolog is very similar to the Relational Language. A few tidy constructs are borrowed from the Relational Language, making Concurrent Prolog syntactically and semantically simple and yet expressively powerful. It accommodates AND-parallelism, OR-parallelism, and stream parallelism.

A clause in Concurrent Prolog has the following syntax:

$$P \leftarrow G_1, \cdots, G_k \mid S_1, \cdots, S_n.$$

The *guard sequence* $(G_1, \cdots, G_k)$ is defined much like the guard sequence of the Relational Language except the G_i's need not be executed sequentially. In fact, all literals within a sequence (either the goal or guard sequence) are executed concurrently with the order of execution determined dynamically by annotated variables and special built-in system predicates. There is no mode declaration. A clause that successfully completes the guard sequence eliminates all other alternatives; it is known as the *candidate* clause. Another major difference between the Relational Language and Concurrent Prolog is that Concurrent Prolog has only one annotation for its variables: the *read-only* "?" symbol, contrasting with the Relational Language's input "?" and output "^" annotations.

The programming style of Concurrent Prolog is to group each set of clauses into a *perpetual process* — that is, the clauses in a group are recursive so that the group acts as a "process" which takes requests from other "processes"

3.2 Execution Models

This section surveys the execution models underlying some parallel logic programming systems. Included are University of Maryland's PRISM system, Bowen's model based on the ABC Algorithm, Conery and Kibler's AND/OR model, and Shapiro's distributed Concurrent Prolog. All these models rely on the concept of *processes*. Though brief, these descriptions should be sufficiently detailed to outline the

general algorithms.

Bowen's model and the distributed Concurrent Prolog description are proposals. An interpreter which simulates the AND/OR model has been written in Prolog [Conery & Kibler 82]. It supports only OR-parallelism; AND-parallelism is not exploited by the interpreter. PRISM has been implemented and supports independent AND-parallelism and OR-parallelism [Kasif et al. 83].

3.2.1 PRISM

There are three components in the PRISM model [Eisinger et al. 82]: *problem solver* (PS), *extensional database* (EDB — assertions or facts), *intensional database* (IDB — clauses or rules).

Solver Component

The PS controls the search space of a goal. It selects a literal from the goal literals to be expanded and sends it to be unified with clause heads in the database components (EDB and IDB). The EDB/IDB returns all necessary information for generating all successors of the literal. While waiting for the EDB/IDB results, the PS expands other clauses (search parallelism). Built-in predicates are evaluated by the PS. The goal selection strategy depends on the user's control specifications.†

Processor Allocation

All processors are viewed as individuals in a pool of processors. At system initialization time, this pool is divided into three groups: PS machines (PSMs), IDB and EDB machines (collectively know as the database machines or DBMs). It is natural to use a processor for each node in the search tree. The PSMs are selected from the pool of PSMs, or DBMs if no PSMs are available. A new PSM is started every time a branching of the search tree takes place and there is a PSM available. The branching of the tree is done at the OR and independent AND branches so that no interprocessor communication is necessary until the results are available.

Database Components

The database is separated into EDB and IDB because the EDB contains only ground clauses (no variables or calls) and hence does not require an "occurs check" in its unification algorithm. The basic functions of the EDB/IDB are to store clauses and to perform unification.

For databases that are small enough to fit into the memory of a single machine, Eisinger et al. propose duplicating the database over all DBMs to promote search parallelism and to avoid communication bottlenecks. The degree of parallelism is dependent upon the number of DBMs available. For large databases, all clauses defined for one predicate are grouped and whole groups are placed on one machine. Groups may be duplicated over several DBMs. A group that cannot fit into one machine employs a master/slave tree relationship, with one DBMs controlling others which contain clauses in the same group.

When a PSM sends a request to the DBMs, the request is passed along until a DBM agrees to handle that request. For small databases, the first idle DBM processes the request. For large databases with non-tree structures (i.e. groups are small enough to fit into one DBM), the first idle DBM containing the matching clauses processes the request. For large databases with tree structures, if a slave picks up the request, then the master is informed and the slave awaits instructions from the master. If the master picks up the request, then it allocates a slave to process the request.

3.2.2 A Model based on the ABC Algorithm

A model similar to the PRISM approach of using identical processes (PSMs) to represent nodes in the search tree is Bowen's model of a concurrent Prolog interpreter based on the ABC algorithm [van Emden 82]. We begin with a description of the *ABC algorithm*.

† See the description of the PRISM language in the previous section Section 3.1.

Figure 3-1: ABC algorithm

```
        cn is the current node.

        cn ← root
A:      if P( cn ) then
                halt with success
        else
                initialize son() for cn
                goto B

B:      if son( cn, x ) then
                { x is next son of cn }
                cn ← x
                goto A
        else
                { all sons of cn have been tried }
                goto C

C:      if father( cn, x ) then
                { x is the father of cn }
                cn ← x
                goto B
        else
                { cn is the root }
                halt with failure
```

The ABC Algorithm

The ABC algorithm is a simple algorithm for interpreting Prolog programs. It is "derived from a mathematical description of the SLD theorem prover."‡ It is a depth-first, left-to-right tree traversal algorithm for finding leaf nodes with the given property P. In the case of Prolog, P is a subgoal.

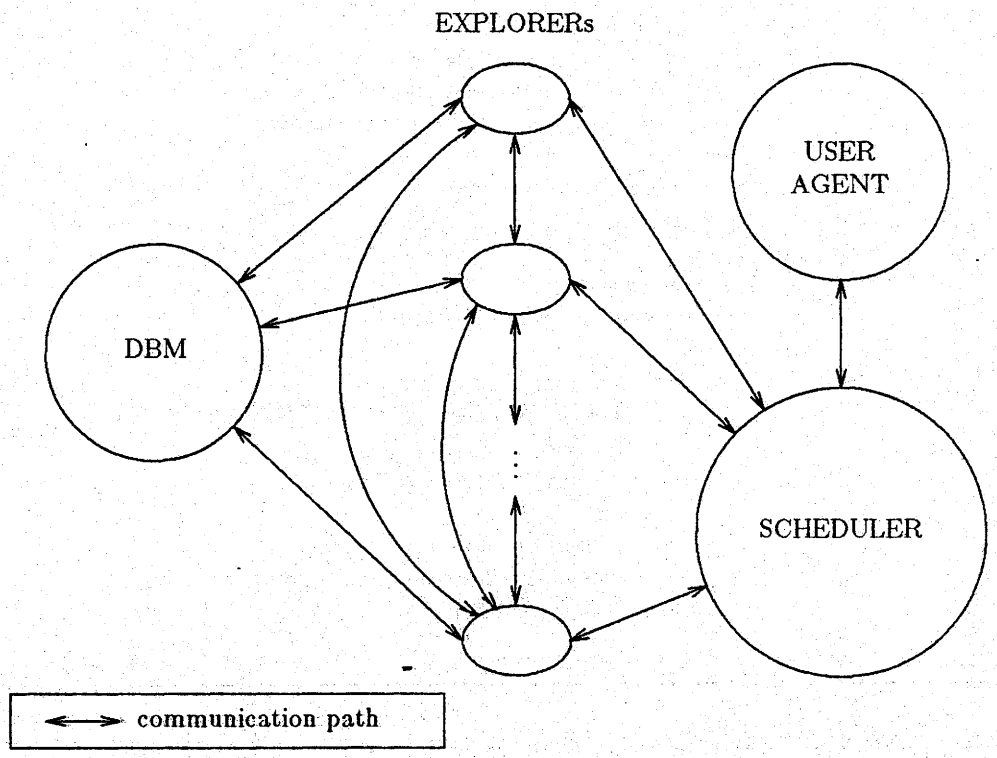
There are two main functions used: *son*(x, y) and *father*(x, y). Given a node x, suppose x has n sons; *son*(x, y) returns *TRUE* the first n times and successively binds y with one of the n sons. *FALSE* is returned after n calls. This function can be viewed as a generator of sons. Given a node x, *father*(x, y) returns *FALSE* if x is the root; otherwise, it returns *TRUE* and binds y to the father of x. The basic ABC algorithm is given in Figure 3-1 [van Emden 82]. This algorithm can be considered as Prolog's theorem prover — it can be applied to a Prolog program tree with the initial goal as the root.

A Concurrent Interpreter

Bowen describes the design of a multiprocess interpreter for Prolog based on the ABC algorithm [Bowen 82]. This design uses no shared memory and processes communicate via message-passing. The interpreter allows both AND and OR parallelism. Figure 3-2 gives the process structure of the system [Bowen 82]. The USER agent process interacts with the human user to "generate" the problems. SCHEDULER allocates EXPLORERs to problems and coordinates the EXPLORERs and USER AGENT. The database component, DBM, manages the database of Prolog clauses while the EXPLORERS traverse

‡ [van Emden 82], page 56.

Figure 3-2: Process structure of a concurrent Prolog interpreter



the program tree, looking for a node with property P.

The concurrent "ABC" algorithm, given in Figure 3-3, is carried out by each EXPLORER. If the EXPLORER finds that its node satisfies property P, it then informs its parent and awaits further instructions. Otherwise, the EXPLORER tries to expand its node by generating all its sons. If there is no son, then the node fails and the EXPLORER informs its parent of its failure. The EXPLORER asks the SCHEDULER for an EXPLORER process to solve each son node. If no EXPLORER is available for a son, then the EXPLORER recurses and acts as the son to traverse that son's search tree. This design is flexible in that it does not distinguish between a *process* and a *processor*. An EXPLORER can be either a process or a processor.

An EXPLORER can receive three types of message: *stop, failure,* and *success*.

- **stop.** *stop* messages are sent from parent to child processes. When a process receives a *stop* message, it forwards the message to its children and then halts.
- **failure.** *failure* messages are sent from a child process to its parent to indicate that it has failed to find a solution.
- **success.** A child process sends its parent a *success* message to inform it that it has found a solution; when the message has been sent, the child then awaits instruction from the parent to either find another solution or stop.

To handle AND-parallelism, the interpreter depends on the explicit notations of the language (like those in IC-Prolog) to tell it how the different AND branches are to interact. Literals which are conjuncts are passed separately to child processes to be evaluated. When a child indicates that it has succeeded, the parent must check the validity of the solution according to the results of other children.

Figure 3-3: Concurrent algorithm

```
A:  if P( cn ) then
        inform parent of success
        wait for parent's instructions
        { parent can request solution and
          instruct cn to find more solutions }
    else
        generate sons of cn
        goto B

B:  if no sons then
        inform parent of failure
    else
        for each son:
            request a process from SCHEDULER
            if process available then
                pass generated son to process
            else
                explore the most recently
                generated son as a new problem
                (i.e. recurse).
```

The database machine (DBM in the figure) is not clearly defined. If it consists of only one machine as depicted in the figure, then it may turn out to be the bottleneck of the system. If it consists of several machines, then search parallelism is possible (as in PRISM).

3.2.3 AND/OR Model

Like the previous models, the AND/OR model of Conery and Kibler is also based on the concept of representing the search tree of a logic program as processes, except that the processes in this model are heterogeneous [Conery & Kibler 81]. The search tree of Prolog is an *AND/OR tree* with alternating levels of *AND-nodes* and *OR-nodes*†. An AND-node contains a conjunction of literals — the literals in a Prolog clause; an OR-node contains a literal and all the alternative clauses with the same head predicate name and arity as the literal.

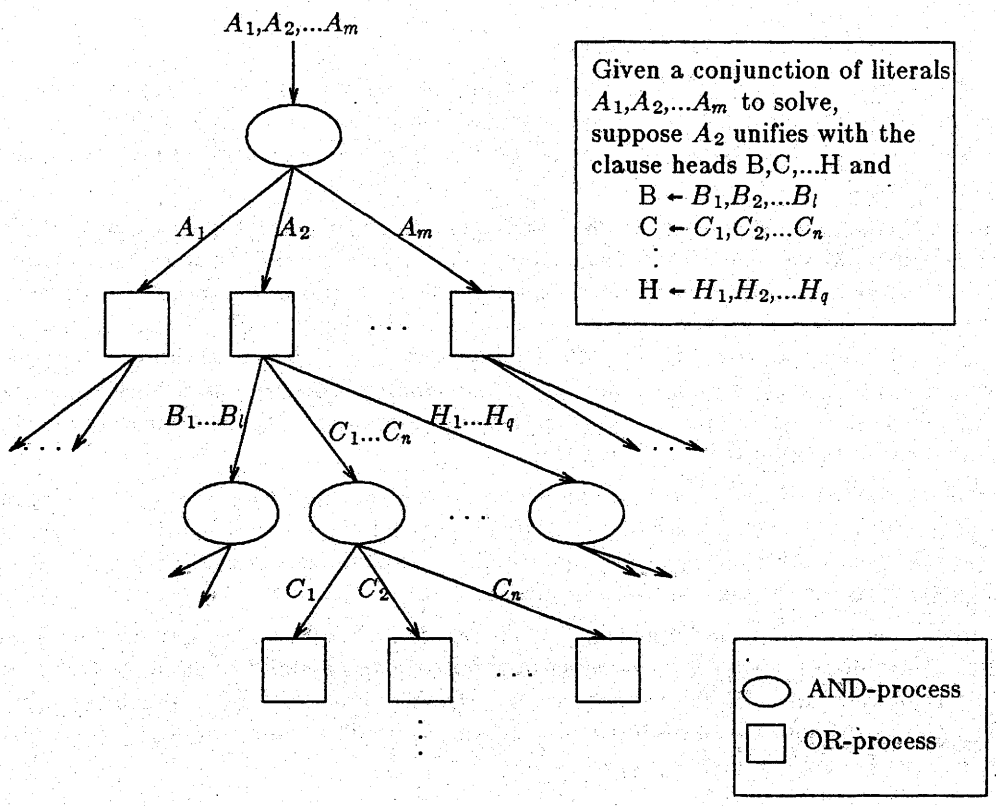
This model represents more intuitively the concepts of AND and OR parallelism than any other model surveyed. The other two models discussed so far merge the components for controlling AND and OR parallelism into one process; hence, it is difficult to identify the algorithms used to handle each type of parallelism. In this model, the AND-process handles the AND-parallelism and the OR-process handles the OR-parallelism. The distributed Concurrent Prolog model, as we shall see in the next subsection, is a version of this AND/OR model.

AND-Processes

Given a conjunction of literals, the AND-process spawns an OR-process for each literal and then awaits the results of the OR-processes.‡ Incompatible bindings result in "backtracking" — OR-processes

† See [Nilsson 79], pp 99-112 for a definition of AND/OR trees.
‡ The actual implementation uses the "left-to-right" evaluation scheme. An OR-process is spawned to solve a literal, then when it succeeds, another OR-process is spawned for the next literal, and so on.

Figure 3-4: AND/OR process tree



that had previously replied with *success* must *redo* their task and try to come up with a different answer. The failure of any OR-process or unresolvable binding inconsistencies will cause the AND-process to "fail"; in which case the AND-process sends a *fail* message to its parent OR-process. If all the descendant OR-processes reply with consistent bindings, the AND-process sends a *success* along with the bindings (if any) to its parent and awaits further instruction. If the bindings sent to the parent process prove unsatisfactory, then the parent sends a *redo* to this AND-process, in which case the OR-processes are made to work again (i.e. the AND-process sends *redo* to its OR-processes).

OR-Processes

Given a literal to solve, an OR-process spawns an AND-process for every clause whose head unifies with the literal; the bodies of these clauses (with the unification substitutions of the head applied) are sent to the AND-processes. The OR-process then goes into *waiting* mode. In waiting mode, an OR-process must return an answer to its parent as soon as it gets a successful result from one of its descendants. When that happens, the OR-process goes into *gathering* mode: it simply collects the results from its descendants and waits for the parent to issue a *redo*. A *redo* causes the OR-process to re-enter waiting mode. Any descendant AND-process that sends the OR-process a *fail* message is recorded as having no more answers. When all descendants have sent *fail* messages, all possible solutions for that literal has been exhausted. Under such circumstances, if the parent of the OR-process requires an answer (the OR-process is in waiting mode), then the OR-process fails. It can also fail if there is no unifiable clause for the literal that it is trying to solve. Upon failure, the OR-process sends a *fail* message to its parent and destroys itself.

Stream parallelism in this model is achieved by slightly modifying the functionality of the OR-process. When an OR-process receives a *success* message from a descendant, it records the answer and *immediately* issues a *redo* to the descendant, forcing it to look for another answer even though one is not needed yet. This can be used, for example, to successively process items from an input stream in a "pipelined" fashion.

3.2.4 Distributed Concurrent Prolog

In his paper, "A Subset of Concurrent Prolog and its Interpreter," Shapiro gives a sketch of a distributed Concurrent Prolog machine. This sketch strongly resembles the AND/OR model. Instead of using AND-processes and OR-processes, this model uses three types of process: *conjunction-process, goal-process* and *clause-process*.

Given a conjunction of literals, the conjunction-process spawns a goal-process for every literal and terminates when all the goal-processes terminate. A goal-process is initialized with a literal; it then spawns clause-processes for every possibly unifiable clause of the given literal (i.e. all clauses with the same head predicate name and arity). Each clause-process is given the literal and a clause which it tries to unify. If the unification is successful, then it spawns a conjunction process to try to solve the guard sequence.† Successful termination of the descendant conjunction process results in the successful termination of the clause-process. When one descendant clause-process of the goal-process terminates successfully, the goal-process in turn terminates successfully.

From the above description, we can see that the conjunction process is very similar to the AND-process and that the functionalities of the goal-process and clause-process are contained in the OR-process. The goal-process and clause-process are separated so that unification can be attempted simultaneously (by each clause-process) at the cost of more processes.

Some issues are not well-defined in this description. For example, how is the "goal sequence" to be executed? What does it mean for a process to terminate successfully? What does it mean for a process to terminate *unsuccessfully*? What happens to the variables that have been instantiated by a process? More generally, how are bindings represented and communicated? These questions are addressed in the next chapter by Port Prolog's model of computation.

† See Section 3.1 in this chapter for a description of the Concurrent Prolog language.

4

Model of Computation

This chapter presents the computational model of Port Prolog. We first elaborate on the notion of processes and communication between processes that are used in the remainder of this thesis. We then describe the Port Prolog model which is a formalization of the distributed Concurrent Prolog model. We propose a more precise process structure and also define the communication paths, variable bindings, and synchronization procedures. The handling of system predicates is also discussed.

4.1 The Port Process Abstraction

In the discussion of parallel systems, it is often useful to describe them in terms of *processes*. A *process* is the execution of a self-contained piece of code and its associated data structures. It shares no memory with other processes, regardless of their kinship. The abstraction that we are interested in is a message-based system in which data is shared between processes only via *message-passing* [Cheriton 82, Cheriton et al. 79]. Processes are identified and addressed by a *process identifier*, or *id* for short. Every process has a unique id.

Process management is performed dynamically via two primitives: *create* and *destroy*. Any process can *create* any other process: the creator is called the *parent* process and the newly created process the *child* process. There is no special relationship between parents and children. Process destruction, which is performed upon process termination or by a *destroy* primitive, involves both the abortion of execution and dissolution of all data structures belonging to the process. A process can destroy any process, including itself.

4.1.1 Communication Primitives

There are three basic message-passing primitives: *send, receive,* and *reply*. These primitives are synchronous — that is, they provide synchronization between communicating processes.

Figure 4-1: Process communication primitives

```
send( process_id, request, reply_msg )

receive( process_id, request )
process_id = receive_any( request )

reply( process_id, reply_msg )
```

The process that issues a *send* is suspended until the destination process, identified by *process_id*, receives *request* and replies with *reply_msg*. The destination process may execute any number of instructions before it replies.

There are two types of *receives*: a specific and a general receive. A process issuing the specific

receive is blocked until the process identified by *process_id* decides to *send* to it. When the *send* does occur, the receiver is free to execute. When the receiver subsequently does a *reply*, the sender process (specified by *process_id*) which is awaiting a reply is unblocked.

The semantics for the general receive, *receive_any*, is the same as that for the specific *receive* except the process doing a *receive_any* is suspended until *any* process sends to it. This primitive returns the process identifier *(process_id)* of the sending process so that the receiver knows who to reply to.

4.1.2 Process Structure

Process structure refers to the processes in an application and how they interact. Many issues are involved in the design of an application's process structure; some considerations are process load and functionality, concurrency, memory usage, and communication cost.

The appropriate assignment of duties to processes is vital to avoid overloading processes with too many or unrelated capabilities. For example, a heavily loaded process that manages files and several windows should probably be partitioned into two processes, one for handling files and the other for windows.

An application can improve its response by enlisting the aid of several processes to complete a task. For example, suppose an application needs to update some files and its window; the order of updates to the files and updates to the window is unimportant. These two devices can be updated "simultaneously" by using two processes, one for each device. As one process is updating the files and awaiting the file system to perform the actual updates, the window process can update the screen, and vice versa.

Processes can be used to minimize the overall memory usage of a program. For example, an activity that uses a large buffer during initialization and has no need for the buffer in subsequent computation can save memory by creating a worker process to do the initialization and manage the buffer. When the buffer is no longer needed, the worker is destroyed and its resources freed; hence, a permanent buffer is avoided.

Communication costs and context switching are considerations that must be addressed in the design of a process structure. The synchronous nature of the message-passing primitives requires context switching between communicating processes. Because message-passing is significantly more costly time-wise than a local procedure call, its usage should be minimized.† This suggests an autonomous approach in which processes are very independent and communicate minimally with other processes.

4.2 Overview

Port Prolog is an interpreter developed according to Shapiro's definition of the Concurrent Prolog language. Chapter 3 contains an overview of the language features of Concurrent Prolog. We first review some definitions that are important for the understanding of the model.

- **clause components.** A Concurrent Prolog clause has the following format:

$$P \leftarrow L_1, L_2, \ldots, L_m \mid L_{m+1}, L_{m+2}, \ldots, L_n .$$

P is the *head* of the clause. $L_i, (1 \leq i \leq n)$, is a *literal*. A conjunction of literals is called a *literal-list*. The literal-list $L_1, L_2, \ldots, L_m$ is referred to as the *guard sequence*. If there is no literal in the guard sequence, then the guard sequence is *empty* and the "|" may be omitted. $L_{m+1}, L_{m+2}, \ldots, L_n$ is called the *goal sequence*. The goal sequence may also be empty.

- **candidate clause.** A clause whose head unifies with a literal and whose guard sequence executes successfully is eligible to become that literal's *candidate clause*. The candidate clause is the clause whose guard sequence is *first* evaluated successfully.

- **read-only variable.** Evaluation of a literal containing *read-only* variables is suspended until the variables are instantiated. A variable is read-only if it is has a "?" as a suffix. For example, $X?$ is a

† In Port, a null function call requires approximately 0.02 ms whereas the minimum cost of a send/receive/reply cycle is 2.54ms [Vasudevan 84].

read-only variable.

4.2.1 Process Structure

Port Prolog's execution model is based on Shapiro's informal description of a distributed Concurrent Prolog machine [Shapiro 83]. This model uses three types of process: *conjunction-process, goal-process*, and *clause-process*. The purposes of these processes are similar to those in distributed Concurrent Prolog proposal but their means of achieving their objectives are different and more concise. The following presents a brief overview of the interactions between the three types of process, which are described in more detail in later sections.

Given a conjunction of literals $L_1, L_2, \cdots L_m$ to solve, the conjunction-process creates a goal-process for every L_i, $(1 \leq i \leq m)$. Each goal-process is responsible for finding a solution to its L_i; it does so by creating a clause-process for every clause which has the same predicate name and arity as L_i. The clause-processes then attempts to unify L_i with the clause heads. The successful clause-processes proceed to solve the guard sequence of their clause by spawning a conjunction-process to evaluate the literals in their guard sequence. The conjunction-process solves the sequence by the method described above — that is, spawning goal-processes for each literal in the sequence. When it succeeds, it notifies its parent clause-process. The first clause-process to inform its parent goal-process of its success becomes the candidate clause-process and all other clause-processes are destroyed by the goal-process. The goal-process then destroys itself. The candidate clause-process proceeds to solve the goal sequence of its clause by spawning a conjunction-process and giving it the goal sequence. When the goal sequence is successfully evaluated, the candidate clause-process notifies its grandparent conjunction-process and destroys itself. The grandparent conjunction-process succeeds if it receives a "success" message from every L_i's candidate clause-process.

Each type of process can fail. A conjunction-process fails if *any* of its descendant goal-processes or candidate clause-processes fails. A goal-process fails if *all* of its descendant clause-processes fail or if it cannot find any clause with the same predicate name and arity as the L_i that it is supposed to solve. A clause-process fails if it cannot unify its L_i with the given clause or if its child conjunction-process fails. Upon failure, a process notifies its parent process and then destroys itself.

4.2.2 Communication

Port Prolog's execution model is based on independent processes which do not share memory. This attribute contributes greatly to the distributive nature of the model; however, it also requires the definition of an explicit communication scheme. All messages passed between processes can be classified as either *control* messages or *binding* messages. Most of the messages are of the latter kind, either to obtain or to make available variable instantiations. The rest of the messages are control messages that convey the status of the sender.

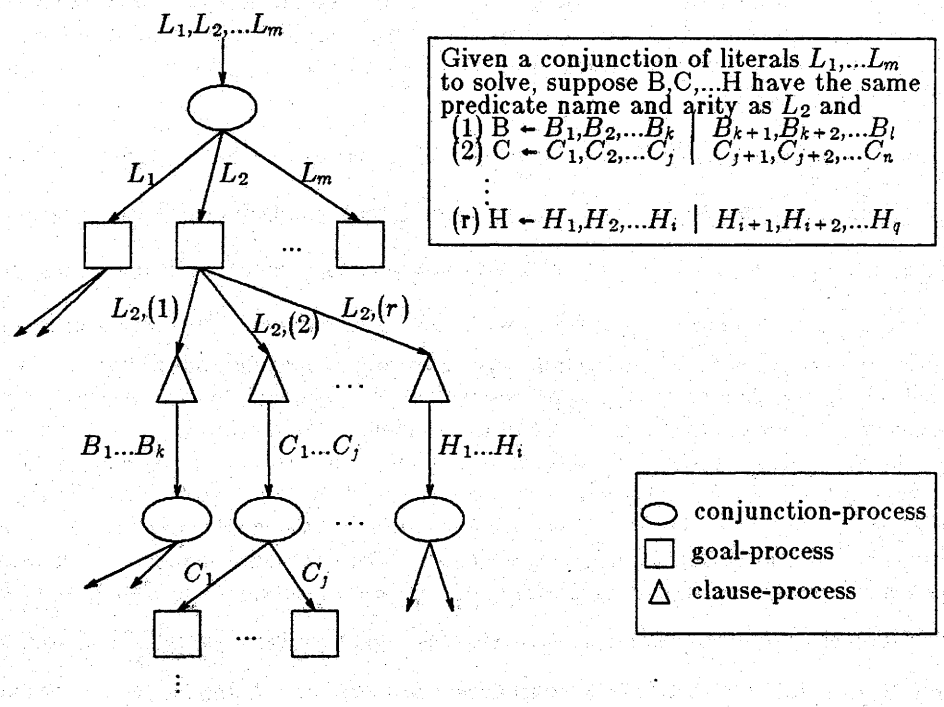
4.3 Conjunction-Processes

The purpose of a conjunction-process is to oversee the proof of a given literal-list. To this end, the conjunction-process creates a goal-process for every literal in the literal-list and passes the literals to their corresponding goal-processes. When the conjunction-process has completed these initialization duties, it enters into *service* mode. In service mode, it may receive messages from any other process, although some messages *(fail, success, commit)*, can only come from its descendants. A conjunction-process keeps track of the values of the variables in its literal-list. It is designed to always record and then pass on whatever bindings it receives immediately to its parent to make available bindings for non-read-only variables which may be shared. When all literals have succeeded, the conjunction-process returns any variable bindings not yet sent to its parent process and destroys itself.

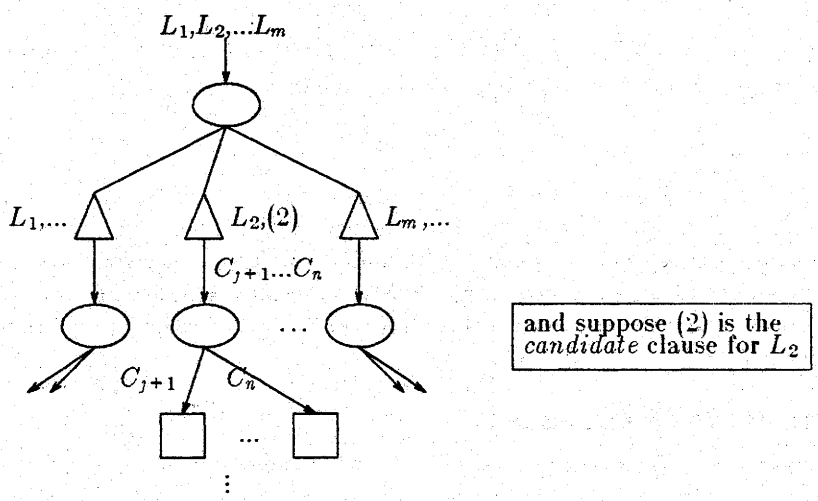
During service mode, the conjunction-process must also remember the state of execution of each literal. The possible states are: UNCOMMIT, COMMIT, SUCCEED, and FAIL. A literal is in the

Figure 4-2: Port Prolog process structure

(a) Before ANY *commits*

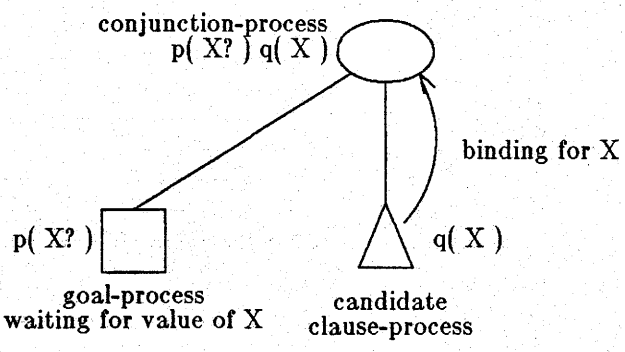


(b) After ALL L_i's have been *committed*



COMMIT state if the evaluation of clauses with the same predicate name and arity as it produces a candidate clause. A literal is UNCOMMIT if a candidate clause has yet to be found. A COMMIT literal is always associated with its candidate clause-process whereas an UNCOMMIT literal is always associated with its goal-process. Initially, all literals are UNCOMMIT. The state of a literal is SUCCEED if the candidate clause has been solved successfully. The fourth state, FAIL, is the only state which is not recorded

Figure 4-3: Synchronization of variable bindings



because once a literal in the literal-list fails, the conjunction-process itself fails and states are no longer needed.

Figure 4-4: Conjunction-process messages

(a) incoming:

commit (from goal-process)
success (from clause-process)
fail (from goal- or clause- process)
new binding
need binding
patch binding

(b) outgoing:

success (to parent process)
fail (to parent process)
new binding
need binding
patch binding

4.3.1 Incoming Messages

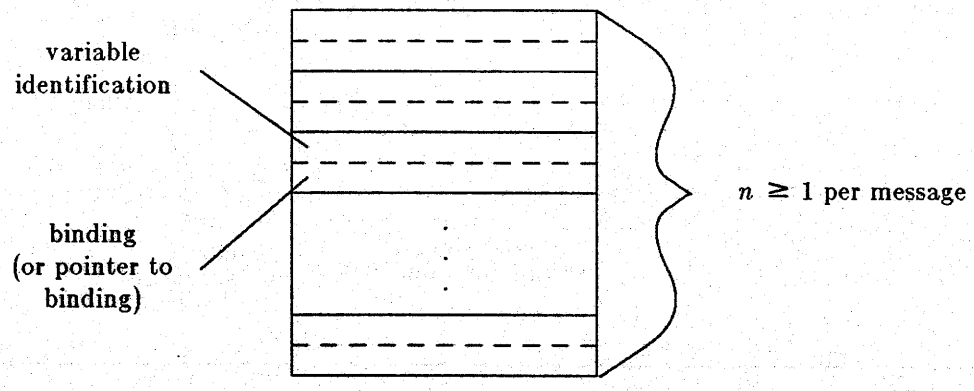
Variables may be instantiated via the *new binding* message and updated via the *patch binding* message.

- **new binding.** When a *new binding* message comes in, the conjunction-process records the binding and passes it onto any process that was waiting for it.
- **patch binding.** A *patch binding* message is used when the variable has already been instantiated but contains unbound sub-components.

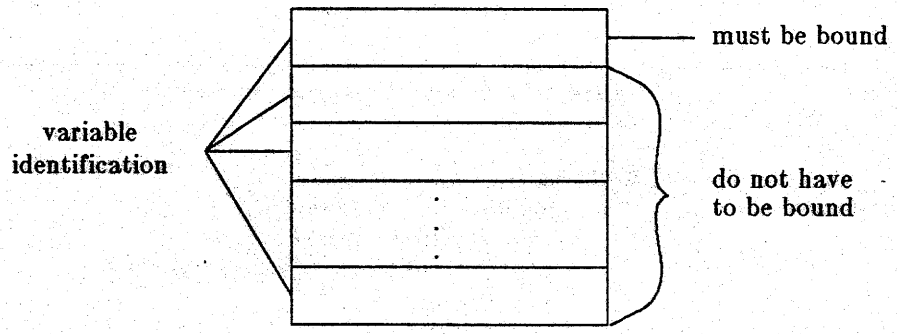
An example of the use of *new binding* and *patch binding* is the instantiation of a variable to a list. Usually, the variable is initially bound to a list using *new binding* with only the head instantiated. The tail is later bound using a *patch binding* message.

Figure 4-5: Binding message formats

(a) The *new* and *patch binding* message format



(b) The *need binding* message format



• **need binding.** Other processes may obtain binding information from the conjunction-process by sending it a *need binding* message. A *need binding* message contains two parts: the first identifies the variable (say, X) *must* be bound and the second contains a list of variables that has yet to be instantiated. If there is no binding for X, then the requester is suspended until the binding becomes available. When X is instantiated, its binding is returned along with any other available bindings for variables specified in the second part of the *need binding* message.

• **commit.** When a *commit* message arrives from a goal-process, the conjunction-process changes the state of the literal from UNCOMMIT to COMMIT and records the candidate clause-process.

• **success.** A *success* message (from the candidate clause-process) indicates that a literal has been evaluated successfully; the conjunction-process changes the state of the literal from COMMIT to SUCCEED.

• **fail.** A *fail* message indicates that a literal cannot be solved. It can come from either a goal-process or a candidate clause-process. When a conjunction-process receives a *fail* message, it immediately halts the evaluation of all literals in its literal-list (by destroying the processes which are doing the evaluations), informs its parent of the failure, and destroys itself.

The messages *new binding,* and *patch binding* may "fail" if the bindings they contain are inconsistent with those stored in the conjunction-process; in this case, the effect of the message which introduced the inconsistency is the same as a *fail* message. The sender is informed of the failure also. The bindings in these two messages, as shown in Figure 4-4, are stored as pairs with the first element

identifying the variable and the second element containing the binding.

4.3.2 Outgoing messages

The conjunction-process sends messages to its parent and other conjunction-processes. A *success* message (which may be accompanied by bindings) is sent to the parent process when the conjunction-process have received a *success* message for each literals in its literal-list. A *fail* message is sent to the parent when the conjunction-process receives a *fail* message. When the conjunction process sends a *new* or *patch binding* message to another process, this message fails if its binding is inconsistent with those of the receiving process; in which case, the conjunction process sends a *fail* message to its parent.

Since all variable bindings that the conjunction-process receive are *committed*, any binding which it receives may be forwarded immediately to its parent via the *new binding* or *patch binding* message. *need binding* messages could be issued periodically to poll the parent process for any newly instantiated non-read-only variables. The polling frequency depends on many factors such as the available processor power and the degree of stream parallelism desired. The implementation described in Chapter 6 does not have this feature.

4.4 Goal-Processes

The objective of a goal-process is to locate a *candidate* clause for a given literal. Given a literal to solve, a goal-process spawns a clause-process for every clause whose head has the same predicate name and arity as the literal. This is done, instead of spawning a clause-process for every clause whose head *unifies* with the literal, so that unification may be done simultaneously. For literals with read-only variables, the creation of clause-processes is suspended until all read-only variables have been instantiated in order to minimize the number of message passes which would have been necessary due to the read-only variables if the suspension is relegated to the clause-processes.

The goal-process gives the literal that it is trying to solve and a clause which may unify with the literal to each clause-process. When all clause-processes have been created, the goal-process waits for one of the clause-processes to send it a *commit* message to indicate that the particular clause-process has successfully solved its guard sequence. This clause-process becomes the *candidate* process. The goal-process then identifies the candidate process to its parent conjunction-process and destroys all other descendant clause-processes. The goal-process then destroys itself as its service is no longer needed.

If there is no clause with the same predicate name and arity as the literal or no candidate clause-process is found, the goal-process sends a *fail* message to its parent conjunction-process and destroys itself.

Figure 4-6: Goal-process messages

(a) incoming: (from clause-process)
commit
fail

(b) outgoing: (all to parent conjunction-process)
commit
fail
need binding

4.4.1 Incoming Messages

The goal-process is the simplest of the three process types and its messages reflect this simplicity. It only handles two types of message: *commit* and *fail*, both from its descendant clause-processes. During its lifetime, a goal-process receives at most one *commit* message but may receive several fail messages. Suppose a goal-process has spawned n clause-processes. Then, it receives either n *fail* messages or i (where $0 \leq i < n$) *fail* messages and one *commit* message. The reception of a *commit* message from a descendant clause-process will result in the goal-process ignoring all subsequent messages sent to it.

4.4.2 Outgoing Messages

All messages from a goal-process are directed towards its parent conjunction-process.

- **commit.** Upon the reception of a *commit* message, the goal-process issues a *commit* message of its own to its parent conjunction-process. This *commit* message contains information which identifies the candidate clause-process.

- **fail.** If and when the goal-process has received *fail* messages from every descendant clause-process, it responds by sending a *fail* message to its parent conjunction-process.

- **need binding.** When a goal-process finds uninstantiated read-only variables in the literal that it is trying to solve, it sends a *need binding* message to its parent. n such variables will result in n *need binding* messages.

4.5 Clause-Processes

A clause-process is responsible for the evaluation of the given literal using the given clause. The first thing that a clause-process does is attempt to unify the given literal with the head of the clause. If unification is successful, it is ready to attack the guard sequence of the clause; otherwise, it fails. If there is no guard sequence, (in which case unification act as the guard) the clause-process sends a *commit* message to its parent goal-process immediately; otherwise, it spawns a conjunction-process to solve the guard sequence. When the guard sequence is successfully solved, the clause-process sends a *commit* message to its parent goal-process. If it is the first of its brother processes to send such a message to its parent, then it becomes the candidate clause-process.

Immediately after becoming a candidate clause-process, the process sends the bindings obtained from unification and evaluation of the guard sequence to its grandparent conjunction-process (see Figure 4-7). It then spawns a conjunction-process to solve the goal sequence. The candidate clause-process must remain (unlike the goal-process) to maintain the mapping between the variables in the literal being solved and the candidate clause. When its child conjunction-process has terminated successfully, the clause-process sends binding messages of variable values not yet forwarded to its grandparent conjunction-process. It then sends a *success* message to its grandparent.

A clause-process fails if its child conjunction-process cannot solve the sequence given to it. Upon failure, the clause-process sends a *fail* message to its parent goal-process if the sequence is a guard sequence, or to its grandparent conjunction-process if the sequence is a goal sequence. Then, the clause-process destroys itself.

4.5.1 Incoming and Outgoing Messages

A clause-process receives all of its messages from its child conjunction-process. It receives a *success* or *fail* message depending on the evaluation of the literal-list sent to its child. A *success* message from the child conjunction-process solving a guard sequence causes the clause-process to send a *commit* message to its parent goal-process, which in turn replies with the identity of the grandparent conjunction-process. When a *need binding* message is received, the variables contained in the message are "mapped" to variables of the variables of the grandparent process and a *need binding* message is sent to the grandparent; *patch binding* and *new binding* messages are similarly handled. This mapping is described below in Section 4.6. *patch binding* and *new binding* are received only when the child conjunction-process is solving the goal sequence (not the guard sequence).

Figure 4-7: Candidate clause-process

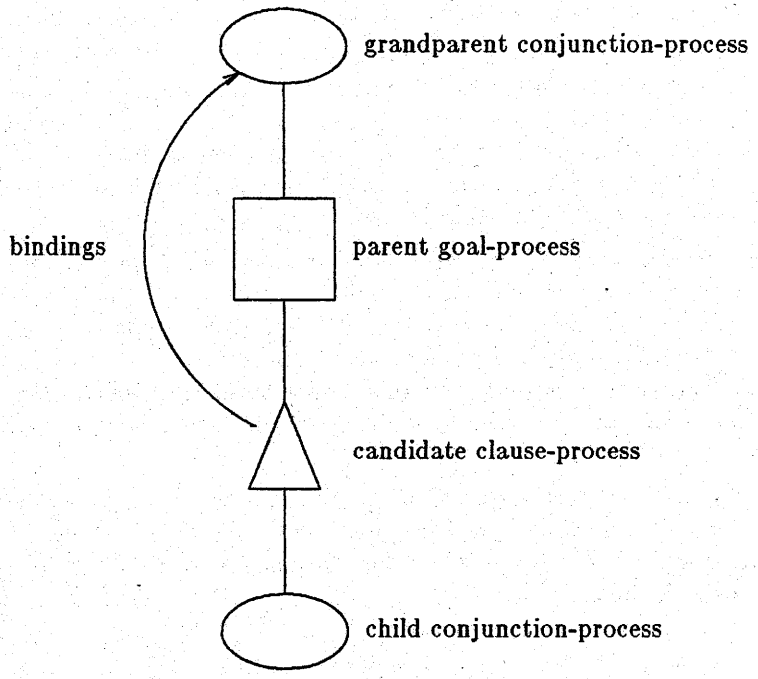


Figure 4-8: Clause-process messages

(a) incoming: (from child conjunction-process)

success
fail
new binding
need binding
patch binding

(b) outgoing:

fail
success (to grandparent conjunction-process)
commit (to parent goal-process)
new binding (to grandparent conjunction-process)
need binding (to grandparent conjunction-process)
patch binding (to grandparent conjunction-process)

4.6 Binding of Variables

The binding of variables in Port Prolog is done distributively — in other words, there is no centralized source (cf. the stacks in Sequential Prolog) which keeps track of all variables. This non-centralized approach makes the system less dependent upon tightly-coupled processor architectures, and less prone to communication bottlenecks and consistency problems due to different processes instantiating the same variables simultaneously. There are also, however, disadvantages associated with this method.

First, redundancy of bindings is an inevitable consequence. In a centralized scheme, parts of bindings can refer to other bindings easily (for example, by using pointers). In a distributed scheme, such reference is expensive in terms of storage and retrieval time — as reference must include the process that has the bindings as well as directions as to how to query the process for them. Therefore, Port Prolog will copy bindings whenever they are referenced, so that future references by the same process need not go through the costly routine of extracting them from other processes.

Another disadvantage to the distributed binding environment is the difficulty of creating and determining the binding references. Which process does a process inquire when it contains uninstantiated variables that need to be bound? The answer to this question requires not only a correct algorithm, but also one which takes communication and time costs, memory costs, and number of intermediate processes into consideration. Moreover, read-only variables must be distinguished from unannotated variables for synchronization purposes; and in order to support stream parallelism, variable bindings must be distributed promptly.

The design of Port Prolog's binding mechanism hinges on three concepts: (1) creation and distribution of a *tag* for uninstantiated variables, from which processes can derive information on how to eventually obtain bindings; (2) classification of uninstantiated variables; (3) an algorithm for retrieving bindings for uninstantiated variables.

4.6.1 Tags

The retrieval and broadcast of a variable's value are simplified by adding to an uninstantiated variable information about how to find the value for the variable in the future. This information, which is referred to as a *tag,* is analogous to a pointer in a program; however, because references must be made between processes in different address spaces, a tag includes process identification as well as variable identification within the process. Tags provide a more direct method for accessing the variable's binding. Without this information, a process must go through its parent process which would, in turn, propagate the query to its parent (or children) and so on until a process is willing to handle the query. Tags are used for both read-only and unannotated variables. Read-only variables are distinguished from other variables for synchronization purposes. A process requiring the instantiation of a read-only variable asks the process identified by the tag and *waits* until an answer becomes available. For an unannotated unbound variable, the process identified by the tag is polled for the value of the variable.

Figure 4-9: Types of variables

```
BOUND
REFERENCE
FREE
READ_ONLY_REFERENCE
READ_ONLY_FREE
```

Variables are classified into five types as listed in Figure 4-9. There are essentially two groups: read-only and normal variables, with further distinction between their "state of uninstantiation" (FREE or REFERENCE). Once a variable is bound, its annotation, whether it existed or not, becomes meaningless; hence, no distinction is made between read-only variables that are bound and normal bound variables.

FREE means that the tag associated with the variable has not been set (i.e. the tag is empty). Processes requiring values for FREE variables must ask their parents. REFERENCE variables are those that contain tags indicating which process is likely to have values for the variables.

Tags are built by clause-processes as a result of unification and clause evaluation. They are initialized under the assumption that the descendants are more likely to bind uninstantiated variables in the future. For example, if $p(X)$ is the call and $p(Y) \leftarrow q(Y)$ is the procedure, then the unification of $p(X)$ and $p(Y)$ results in the creation of a tag for X. This tag specifies that X is a REFERENCE variable and contains the variable identification for Y and the identifier of the process solving $p(Y) \leftarrow q(Y)$. Y remains FREE.

Because of the impossibility of correctly predicting which process will be the "producer" of the binding, tags may not always be "correct." That is, they may not identify the relevant process directly, but rather indirectly. However, even when they do not identify the producer process, tags can still be useful for providing a means of notifying the process named in the tag when another process has performed the instantiation.

4.6.2 Storage of Bindings

Bindings of variables in a literal-list are stored in both conjunction-processes and clause-processes. Environment tables are built by a process to store the bindings and information pertaining to the mapping of variables in this process to those in other processes.

Bindings of variables in a literal-list are stored in the conjunction-process that is solving the literal-list. This enables descendant processes to access (via message-passing) variables shared among literals in the literal-list. For example, for the literal-list

$$father(\ Child, Father\ )\ \&\ mother(\ Child, Mother\ ),$$

a conjunction-process would keep track of the bindings for *Child, Father,* and *Mother*. The descendant processes evaluating $father(\ Child, Father\ )$ can ask this conjunction-process to instantiate or retrieve the bindings of *Child* and *Father*.

Clause-processes also store bindings. For unification, two environment tables are set up to house the bindings of variables in the clause and those in the goal literal. These environments are also used to map variables in the clause to those in the literal. This mapping is necessary for "tracking down" a variable's value. For example, if a clause-process is given

literal: $father(\ Child, Father\ )$

clause: $father(\ C, F\ ) \leftarrow male(\ F\ )\ \&\ parent(\ C, F\ ).$

The instantiation of F by the descendants of this clause-process (say, by the evaluation of $parent(\ C, F\ )$) would require a mapping between F and *Father* so that *Father* can also be bound correctly.

4.6.3 Retrieval of Bindings

Two types of process initiate retrievals of bindings: goal-processes and conjunction-processes. Retrieval of bindings for read-only variables are initiated by goal-processes. Conjunction-processes poll their parent clause-processes to update its variables so that unannotated variables may be instantiated (for stream parallelism).

A process initiates a retrieval by querying another process (using a *need binding* message); this "other" process (the *target* process) is determined using the variable's type and tag information, and is either a conjunction- or a clause- process. The target process is the parent if the variable is FREE and the process named by the tag if the variable is REFERENCE. The target process first checks if the named variable has been instantiated yet. If it has, then the value is returned; otherwise, for a read-only variable, the target process suspends the sender until the variable that the sender needs is available. The sender is not suspended for normal variables, instead, it is informed that the variable is unbound. If the target process is a clause-process, the unbound variable is mapped into a variable know to its grandparent

conjunction-process and forwarded to its grandparent. The grandparent then becomes the target process and follows the same procedure as described above.

4.7 Synchronization

Synchronization in the Concurrent Prolog language is achieved through the use of read-only variables (see Figure 4-3). Execution of a literal that contains read-only variables cannot proceed until its read-only variables become instantiated. In the Port Prolog model, read-only variables are handled exactly according to the language definition: the execution of a literal is suspended until all read-only variables (if any) are instantiated. This suspension involves the goal-process sending a *need binding* message to a conjunction-process for every unbound read-only variable. Suspension is possible due to the blocking nature of the send primitive, as assumed by our process abstraction.

4.8 System Predicates

System predicates or *built-in predicates* are literals that are evaluated by making special *system calls*. For example, *put(Char?)* is a system predicate that is evaluated by making a system call to print out the character *Char*. In this section, we describe how Port Prolog handles built-in predicates and compare its method with those from other systems.

PRISM's model evaluates system predicates in the solver process (PSM).† This can lead to very "large" PSM's if the number of system predicates is large — the code for handling all the system predicates must be duplicated for every PSM. In order to avoid this space problem, Port Prolog handles built-in predicates in processes outside the solver processes.‡

System predicate processes (SPP) handle all system predicates. Each time a built-in predicate is used, an SPP is created to solve the predicate. Using several SPP's instead of one has several advantages. First, several SPP's may be active at the same time; hence, several system predicates can be executing concurrently. Secondly, system predicates that need to be synchronized can be easily accommodated. For example, the SPP evaluating *put(Char?)* with *Char* uninstantiated can suspend by sending a *need binding* message to the producer of *Char* until *Char* becomes bound. Finally, the function of an SPP is not fixed by the model. Each SPP need not be identical to other SPP's. This flexibility leaves the decision of how to organize system predicates to assign to SPP's up to the designer of the interpreter. For example, if only a small set of system predicates is available, one might use identical SPP's, each containing all system predicates. This would simplify the procedure that decides which SPP to use. Larger libraries of system predicates may warrant a more complex decomposition involving several different types of SPP. This distribution of built-in predicates avoids the problem of duplicating large SPP's.

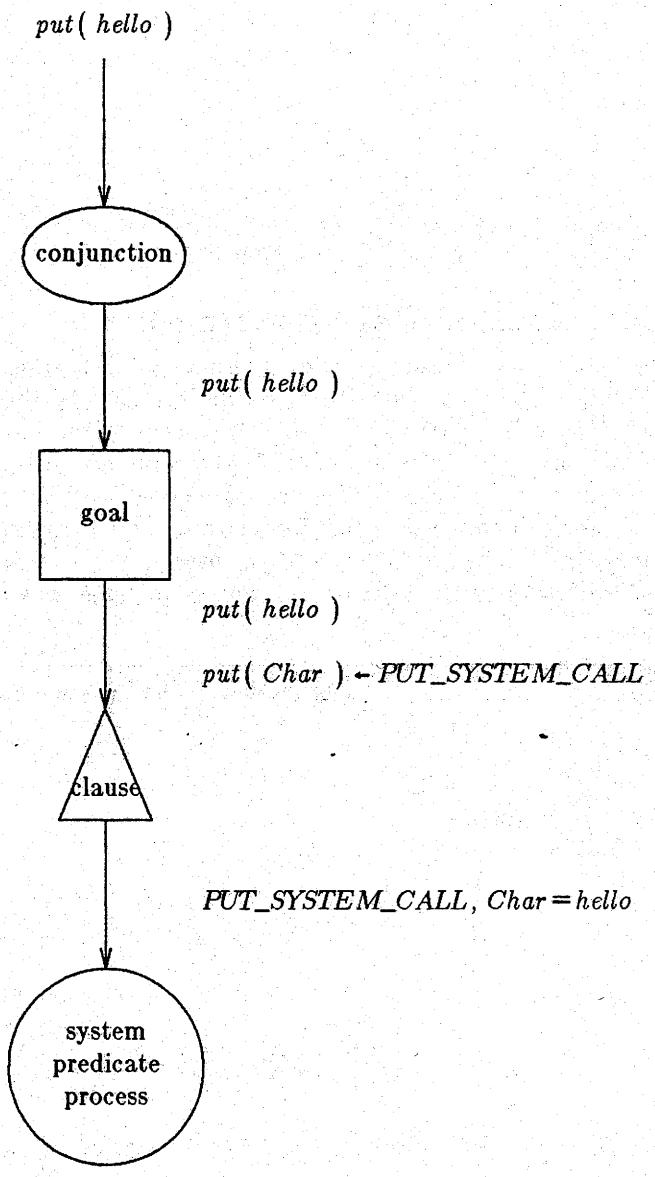
SPP's are related to the process tree of conjunction, goal, and clause processes in the following manner. A clause-process is responsible for recognizing that its given clause is a built-in predicate. This recognition is done after successful unification of the given literal and clause. The clause-process then checks whether the body of the clause is a system call. If so, it creates an SPP — the type of which is possibly determined by the type of system call — to execute the call. The SPP then asks the clause-process for variable bindings, evaluates the call, and returns any variables bound during the evaluation. The clause-process is chosen as the interface between the SPP and the solver process tree because it provides the necessary mapping of variables in the literal and the clause head.

The SPP concept fits well with Port Prolog's computation model. The clause-process uses an SPP, instead of a sub-tree of processes, to solve a literal-list. In the case of a system predicate, the literal-list happens to be a number representing a system call. Hence, system predicates in Port Prolog have no guard sequence; unification act as the guard and the system call is the goal-sequence.

Shapiro's sequential implementation of Concurrent Prolog [Shapiro 83] requires extra synchronization predicates to provide the necessary interface to system functions like input/output. For example,

† For a discussion of PRISM's PSM, see Section 3.2 in Chapter 3.
‡ By "solver processes" in Port Prolog, we mean goal-processes, conjunction-processes, and clause-processes.

Figure 4-10: System predicate process



wait(X) is a special predicate that suspends evaluation until X is instantiated. This predicate is used frequently to ensure the correct execution of predicates such as *write*(X). In Port Prolog, this level of interface is obviated; all predicates, whether built-in or normal, use the same means for synchronization — namely, the read-only annotation (X?).

5

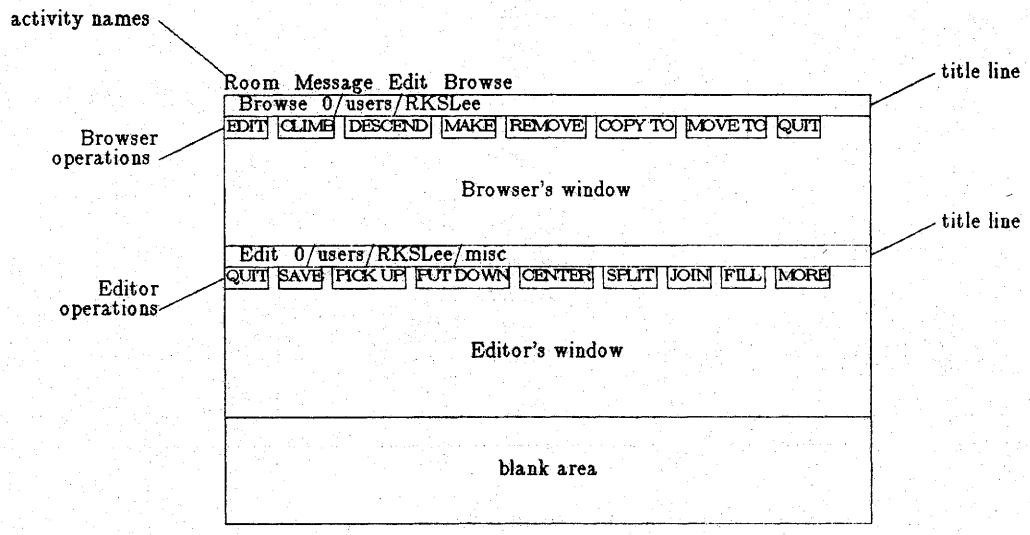
The Port Prolog Language & User Interface

This chapter describes the syntax and structure of Port Prolog programs and the interpreter's user interface. We first provide an overview of the Waterloo Port implementation environment.

5.1 The Waterloo Port Environment

This section briefly describes the environment, Waterloo Port, which was used to implement Port Prolog. Waterloo Port is an operating system developed by the Software Portability Group at the University of Waterloo. The name "Port" refers to the portable nature of the system across different machine configurations; that is, it can be made to run with different processors and peripheral devices without major overhauls to the system. The development of Port Prolog was done on a Port system configured for an IBM† personal computer workstation with a monochrome display and a mouse. The following discussion successively addresses Port's user interface, processes, and networking capabilities. We also describe some features of the Port programming language that influenced the design of Port Prolog.

Figure 5-1: The Port screen format



† IBM is a trademark of the International Business Machines Corporation.

5.1.1 The Port User-Interface

Port has a screen-oriented user-interface [Didur et al. 84]; interaction with the system is performed by *pointing* at and *selecting* objects on the screen with a cursor driven by the mouse. Port supports multiple windows on a screen and concurrent execution that enables several programs to be not only executing simultaneously, but also being viewed simultaneously. Figure 5-1 illustrates a screen occupied by windows. The top line of the screen is reserved for *activity names*; each name corresponds to a window. A window can be vertically enlarged or shrunken but cannot overlap other windows. A window that is not visible on the screen is *hidden* and is brought onto the screen by selecting its activity name. In Figure 5-1, *Room* and *Message* are hidden.

A Port *activity*, which is made up of one or more processes, may or may not use a window. A Prolog interpreter is an activity, for example. An activity that uses windows usually formats its windows to conform to "standard" Port windows. The top line of a window is called its *title line*. The title line contains the name of the activity and some parameters used by the activity. The second line of the window contains a row of *operations*. Selecting an operation starts the corresponding action. For example, selecting QUIT causes the activity to terminate and the window to disappear. The activity can present the rest of the window in any format. Displaying the title line and operations, interpreting the selection of an operation, growing and shrinking of a window, buffering and echoing of input, and displaying of output are handled by the activity that owns the window.

5.1.2 Port Processes

Port is a multi-process operating system that supports concurrent activities. Process communication and synchronization is achieved with blocking interprocess communication primitives: *send, receive* and *reply*. Dynamic process creation and destruction are performed using two process management primitives: *create* and *destroy*. These primitives and the notion of a "process" are described in more detail in Section 4.1 in Chapter 4.

There are several typical process structures that prevade the system. They are named after the anthropomorphic structures that they closely resemble [Cheriton 82]. This structuring clarifies the responsibilities of a process. Some sample processes are:

- **server.** A *server* is a process that handles requests (usually from any other process).
- **proprietor.** A *proprietor* is a process which owns a resource. All access to that resource must be made through this process.
- **worker.** A *worker* is a process whose sole purpose is to offload the work of another process (usually its creator).
- **administrator.** An *administrator* is a process that coordinates the activities of other processes. It usually enlists the help of worker processes to service requests.

These definitions are useful in Chapter 6, when we examine the process structure of Port Prolog.

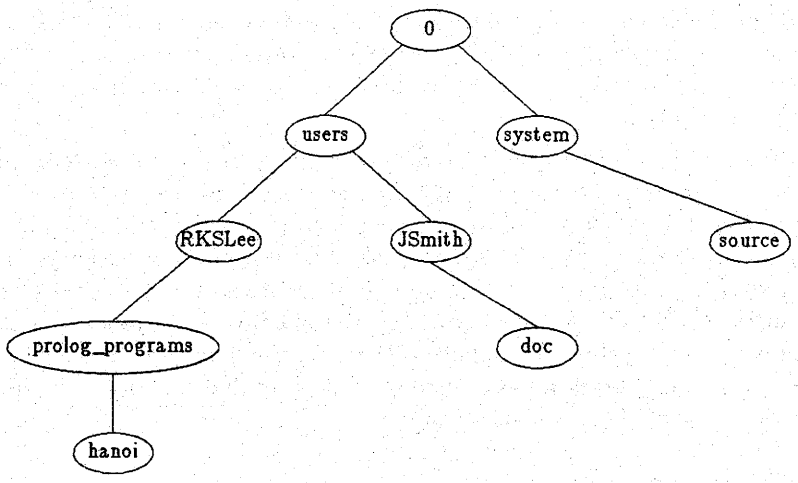
5.1.3 Networking in Port

One of the research directions of Port is to investigate various aspects of network operating systems. Several Port workstations may be connected together to form a network that permits resource sharing. Synchronous message-passing for local process communication is also available for remote process communication. However, the syntax of the local and remote interprocess communication primitives differ. This distinction forces the programmer to decide at programming time the physical location (either local or remote) of each process in his activity. This problem may be circumvented by having all processes use local interprocess communication primitives and relegating the local-to-remote message-passing translation procedure to special worker processes.

5.1.4 The Port File System

Port's file systems are tree-structured. There can be as many as ten file systems, each numbered from 0 to 9. Each file system has a root file from which the rest of the files in the system branch. Files that are direct descendants of the root file can themselves have descendant files, and so on. Any file in

Figure 5-2: An example of a Port file tree



the tree may contain data. Figure 5-2 shows a portion of a file tree. Every file has a *file type*. Some sample file types are *text, commands, holon, function, code86* and *predicate*. A file type indicates the nature of a file's contents. The behavior of certain file manipulating operations is dependent upon file types. For example, they permit the operating system to choose an appropriate editor when the user wishes to edit a file. File types may also impose restrictions on a file's name. This property is a consequence of the integration of the Port programming language and the file system. The Port language is described in the next section.

A file in a Port file system is identified by its unique *pathname*. For example, in Figure 5-2 the leftmost file is identified by the pathname

0/users/RKSLee/prolog_programs/hanoi.

The pathname shows the location of the file with respect to the root of file system "0." Each crossing of a level in the hierarchy is denoted by a "/." The last component in the pathname ("hanoi" in the example) is the *filename* of the file.

5.1.5 The Port Programming Language

The Port programming language is a descendant of the BCPL family of languages [Bonkowski et al. 84]. It is similar to the C language in both syntax and semantics, although Port is more strongly typed than C. The language is closely tied to the Port operating system. For example, the message-passing primitives are an integral part of the language. The Port language also makes extensive use of the file system.

In the Port language, components of a program such as a function or an external variable are placed in individual files which are hierarchically arranged. The hierarchy is used to portray the relationships between the program components. For example, placing a component beneath a function "hides" it from all other parts of the program outside the function's subtree. Also, the name and type of a component is represented by the filename and file type of the file in which the component resides.

Components of a program can be grouped into a *holon* (a module). A holon is a set of logically related components. It consists of a subtree of files, the root of which is of type "holon." The root contains an *export list* that specifies which components in its subtree can be referenced by other components. For example, a floating point package may be made into a holon that contains and exports functions which perform floating point arithmetic; special buffers and internal routines can be hidden and used only

within the holon. Holons may exist within a program or be *imported*. A special file, called the ".holons" file, allows a Port program to include other holons not defined in the program; the .holons file contains the pathnames of all holons to be imported.

5.2 The Port Prolog Language

5.2.1 Syntax

The syntax of Port Prolog follows Shapiro's language definition of Concurrent Prolog. The Port Prolog syntax differs in that comma's in functors and calls are omitted. Figure 5-3 contains the Backus-Naur-Form productions of Port Prolog syntax. An example of Port Prolog syntax is illustrated by the *partition* clauses in Figure 5-4. It contains the definition of the *partition* predicate used in a Port Prolog implementation of quicksort.‡ The definition consists of three clauses.

A clause in Port Prolog has the following format:

<procedure head> *<guard sequence>* | *<goal sequence>* ;

<procedure head> is a call representing the head of the procedure. A sequence consists of a (possibly empty) list of calls. *<guard sequence>* contains the guard calls while *<goal sequence>* contains the goal calls. If the guard sequence is empty, then the "|" may be omitted. During clause execution, the calls in the guard sequence are evaluated concurrently. The goal sequence calls, which are also evaluated concurrently, are processed only if the calls in the guard sequence all succeed. A query has no procedure head nor guard sequence; it only consists of the goal sequence. An assertion is a clause with no guard or goal sequence.

The following are definitions of the terms of Port Prolog:

- **name.** A *name* is a sequence of at most 32 characters. The first character of a name must be a lower case letter.

- **variable.** A *variable* is a sequence of at most 32 characters. The first character of a variable must be an upper case letter, or a "_." A variable can be annotated by a "?" suffix, in which case it is called a *read-only* variable. Evaluation of a call is suspended until its read-only variables are instantiated. A void variable is denoted by "_."

- **string.** A *string* is a sequence of at most 256 characters. A string is delimited by double quotes.

- **integer.** An *integer* is a string of numeric characters. The valid number range is -32767 to 32767.

- **character.** A *character* is a single character enclosed by single quotes.

- **functor.** A *functor* has the form:

  <name> (*<argument list>*).

  <argument list> contains at least one term and at most twenty terms.

- **list.** A *list* has the following syntax:

  [*<list head>* | *<list tail>*]

  [*<list of terms>*]

  <list head> can be any term. *<list tail>* must be a list. *<list of terms>* may contain one or more terms separated by comma's. An empty list denoted by "[]."

5.2.2 Program Structure

This section describes how clauses are structured in a Port Prolog program. Port Prolog adopts the Port language's philosophy of integration with the file system. The advantages of this integration are also discussed.

---

‡ Translated from a Concurrent Prolog implementation of quicksort found in [Shapiro 83], page 11.

Figure 5-3: Backus-Naur-Form productions for Port Prolog syntax

<clause> ::= <call> [<sequence> '|'] [<sequence>] ';'.

<query> ::= [<sequence>] ';'.

<sequence> ::= <call> { <call> }.

<call> ::= <functor> | <name>.

<functor> ::= <name> '(' <term> {<term>} ')'.

<list> ::= '[' <term> { <list tail> } ']' | '[]'.

<list tail> ::= ('|' <list> | { ',' <term> }).

<term> ::= <name> | <string> | <integer> | <character>
| <variable> | <list> | <functor>.

<name> ::= <lower case letter> {<valid symbol>}.

<variable> ::= <normal variable> | '_' | <read-only variable>.

<normal variable> ::= <upper case letter> {<valid symbol>}.

<read-only variable> ::= <normal variable> '?'.

<integer> ::= ['+' | '-'] <digit> {<digit>}.

<character> ::= '"' <ascii character> '"'.

<string> ::= '"' <ascii character> {<ascii character>} '"'.

<ascii character> ::= any symbol from the ASCII character set.

<valid symbol> ::= <lower case letter> | <upper case letter>
| <digit> | '_'.

<digit> ::= '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'.

<lower case letter> ::= 'a' | 'b' | 'c' | 'd' | 'e' | 'f' | 'g' | 'h'
| 'i' | 'j' | 'k' | 'm' | 'n' | 'o' | 'p' | 'q' | 'r'
| 's' | 't' | 'u' | 'v' | 'w' | 'x' | 'y' | 'z'.

<upper case letter> ::= 'A' | 'B' | 'C' | 'D' | 'E' | 'F' | 'G' | 'H'
| 'I' | 'J' | 'K' | 'M' | 'N' | 'O' | 'P' | 'Q' | 'R'
| 'S' | 'T' | 'U' | 'V' | 'W' | 'X' | 'Y' | 'Z'.

Figure 5-4: An example of Port Prolog syntax

```
partition( [X|Xs] A Smaller [X|Larger] )
     A<X | partition( Xs? A Smaller Larger );

partition( [X|Xs] A [X|Smaller] Larger )
     A≥X | partition( Xs? A Smaller Larger );

partition( [] _ [] [] );
```

5.2.2.1 Source Files

A predicate must be stored in a file that has the same filename as the predicate name. For example, the clauses that define the *append* predicate must reside in a file with the filename "append" and the file type "predicate." In other words, all clauses in that file must have the head predicate name *append,* but the arity of the clauses may differ.

By isolating predicates in this way, Port Prolog forces the user to organize his Port Prolog programs. The naming convention of the file makes predicates easy to locate. The user no longer needs to search through several large files looking for predicates with the same name. This naming scheme also catches errors caused by mis-typing the head predicate name of a clause. Since Port allows several editors to display their contents on the screen simultaneously, the user can view several groups of predicates at the same time.

5.2.2.2 Modules

A module is a collection of logically related predicates. A module in Port Prolog is a subtree of files; each file contains clauses belonging to the module. For example, the *append* predicate may belong to the *lists* module, which may contain other predicates for list manipulation such as *reverse, sort,* etc. A predicate within a module can be referenced by any other predicate within the module, but may be hidden from predicates in *other* modules. A predicate can be used by predicates in another module only if it is being *exported* by its home module.

The global nature of Prolog programs have been under attack for its insensitivity to the needs of managing and debugging large Prolog programs. Modules in Port Prolog provide a 2-level organization structure by which the user can use to manage large Port Prolog programs. Predicates in Port Prolog may be *local* or *global*. Local predicates cannot be referenced outside the module while global predicates can be referenced by any module.

5.2.2.3 Exporting Predicates

The root file of a module subtree may be of file type "predicate" or "holon." If the file type is "predicate," only those predicates in the root file are exported; *all* other predicates defined beneath the root file of the module are local. This structure of a module can be used in the case where only one predicate needs to be global and the rest local. For example, a module that sorts a list needs to only export the *sort* predicate; hence, this module can be structured with the clauses defining *sort* located in a file of type "predicate" and the remainder of the predicates defined in files located beneath *sort*'s file. A root file with file type "holon" contains an export list specifying all predicates in the module that are global. For example, if the export list is

append(3) *reverse*(2)

then only the predicates *append* with three arguments and *reverse* with two arguments are global. An export list consisting of only "*" indicates that *all* predicates in the module are global.

5.2.2.4 The Source Tree

The tree of files that is given as the "source" to the Port Prolog interpreter is referred to as the *source* tree. Predicates in the source tree form a module. This module is always searched first when the interpreter receives a query. If the query cannot be answered by predicates in this module, then other modules (if any) are consulted.

5.2.2.5 Inclusion of Other Modules

A user may include other modules with the source module by creating a file with the filename ".holons" directly beneath the root file of the source tree. The pathnames in this .holons file specify the modules to be included. This inclusion is done at the source level — the pathnames refer to the source of the modules. For example, a .holons file may contain

0/users/RKSLee/prolog_programs/system
0/users/RKSLee/prolog_programs/lists
0/users/RKSLee/prolog_programs/graphics.

This results in the inclusion of three modules: *system, lists,* and *graphics*. Predicates in a source tree that has this .holons file can reference any predicate exported by these three modules. In general, predicates in all four modules may reference any predicate in its own module plus any predicate exported by the other three.

5.2.3 Compilation

Port Prolog transforms its source into internal code that is used while the interpreter is running and kept in files when the interpreter is terminated. This internal code is a syntax error-free, compact representation of the source.† When the interpreter is started up, the source (source tree plus modules included via the .holons file) is *compiled* into this internal code and stored in files in the *.code* tree. The .code tree files are referenced when predicates defined in them are required for solving queries entered by the user. Because the compilation process is time-consuming (depending on the number of source files), the .code tree can exist between interpretive sessions so that the entire source need not be re-compiled each time the interpreter is invoked.

5.3 The Port Prolog User Interface

The Port Prolog interpreter requires two parameters: the pathname of the source tree and a flag indicating whether the source requires compiling. If the compile option is specified, the interpreter examines the files in the source tree and included modules and compiles them into internal code. This code is then stored into the .code tree. During compilation, the interpreter displays the pathname of the each file as it is being processed.

When the compilation is completed, or if the compile option is off, the interpreter is ready for queries. The interpreter prompts the user for input, accepts input from the user, and replies with appropriate answers. The interpreter prints a "yes" if the query entered is successful; if the query fails, the interpreter responds with a "no." If the query is successful and variables are present in the query, then the query is reproduced with the variables replaced by their corresponding instantiations.

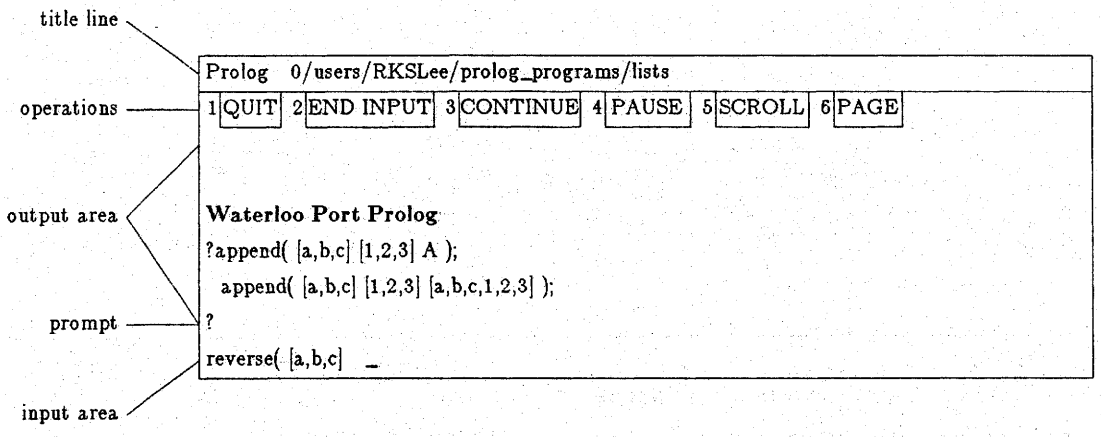
5.3.1 The Port Prolog Window

The Port Prolog window conforms to the format of other window-oriented activities on Port. It has a title line, a set of operations, and an input/output area. The title line contains the activity name "Prolog" and the pathname of the source tree. There are six operations as shown in Figure 5-5.

- QUIT terminates the interpreter and causes the interpreter's window to disappear from the screen. All processes belonging to the interpreter are destroyed.
- END INPUT sends a "end-of-input" signal to the interpreter. This also terminates the interpreter, except the window remains until QUIT is selected. This allows the user to view the contents of the window

† This internal code format is discussed in Chapter 6, Section 6.2.

Figure 5-5: The Port Prolog window



without tying up (memory) resources used by the loaded interpreter.

- PAGE and SCROLL are "control" operations. They do not activate any operation when selected; rather, they set the window into a "mode." Only one of PAGE and SCROLL is operative at any one time (i.e. can be selected). When PAGE is in effect, the window continues to scroll as new lines are displayed until a window-full of output is displayed. The window can be switched into page mode by selecting the PAGE operation. This has the effect of rendering the SCROLL operation operative and the PAGE operation inoperative.

- CONTINUE automatically becomes operative when the window is in page mode and full. Selecting CONTINUE resumes the scrolling of the window until another window-full of output is acquired.

- PAUSE suspends output to the window. The window resumes scrolling when the CONTINUE operation is selected.

Queries may be entered when the cursor is anywhere within the window and is echoed in the *input area* — the last line of the window. The rest of the window is reserved for output. Input is not processed by the interpreter until a carriage return is pressed. Local line editing may be performed before the carriage return is pressed. When the carriage return is pressed, the input line is displayed on the line just above the input area and the output lines scroll up by one line. The input area is then emptied and ready for more input.

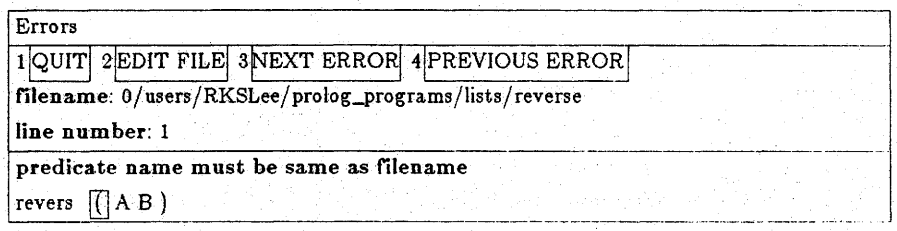
The input area is reserved for the user to enter queries. The interpreter attempts to solve these queries using the clauses stored in the .code tree. There is no facility for adding assertions or clauses dynamically.

5.3.2 The Errors Window

If a syntax error is encountered during compilation, an *errors window* appears and explains the error. Figure 5-6 depicts an errors window. Subsequent syntax errors are viewed with the same errors window.

The errors window contains the customary title line and four operations: QUIT, EDIT FILE, NEXT ERROR, and PREVIOUS ERROR. The rest of the window is used to display four items: (1) the message describing the error, (2) the line of text and column position showing where the error occurred, (3) the line number of the text in the source file, and (4) the pathname of the file in which the error occurred.

Figure 5-6: The errors window



- QUIT makes the errors window disappear.
- EDIT FILE invokes an editor to edit the file containing the error displayed.
- NEXT ERROR and PREVIOUS ERROR cause the next and previous errors (if any) to be displayed, respectively.

5.4 Preparing and Running a Port Prolog Program

The typical sequence of operations for interacting with the Port Prolog interpreter is:

(1) prepare the source tree;

(2) supply the Port Prolog icon‡ with the appropriate parameters;

(3) start the Port Prolog activity;

(4) enter queries to the interpreter;

(5) terminate the Port Prolog activity.

The following discussion assumes the reader is familiar with the Port *Browse, Edit* and *Room* activities [Didur et al. 84].

The source tree is prepared using the Port Browse and Edit activities. Depending on the predicates to be exported, create a file of type "holon" or "predicate" using the Browser. If the file is of type "holon," use the Editor to put in the export list. If the file is of type "predicate," use the Editor to enter the clauses with the same predicate name as the filename. Then, descend using the Browser and create files of type "predicate" for each predicate in the program. If there are other source modules that need to be referenced, use the Browser to create a ".holons" file (of type "text") and put in their pathnames.

Before the interpreter is started, it must be supply with the appropriate parameters. Using Room, examine the Port Prolog icon and enter the pathname of the source module and a flag indicating whether the source needs compiling. The interpreter is started by activating the Port Prolog icon. The interpreter then displays the Port Prolog window. If the compile option was specified, the interpreter prints the predicates being compiled; any error encountered during compilation results in the creation of the errors window from which the user can view the error(s) and possibly edit the guilty file(s). The interpreter then prompts the user for input; the user respond by entering the queries to be solved. Output from the interpreter can be controlled by selecting the CONTINUE, PAUSE, PAGE and SCROLL operations. Finally, the interpreter is terminated by selecting the QUIT operation.

‡ See [Didur et al. 84] for a description of icons and how to start activities in Port.

6

Implementation

In this chapter, we describe the implementation of the Port Prolog interpreter. We start with an overview of the interpreter by presenting the process structure of the system, the files used, and the format of the internal code. We then give detailed descriptions of the major components of the system.

6.1 Process Structure

The Port Prolog interpreter is a multi-process Port activity. Not only is the run-time solver comprised of several processes, so is the rest of the system. For example, a group of processes manages the database while other processes provide services such as parsing, window management, etc. This division of responsibility among several processes has notable advantages.

Because each logical component of the interpreter is implemented as a process, the interpreter as a whole becomes flexible and easier to maintain. Since information sharing is possible only through message-passing, the interfaces between the processes are very distinct. Consequently, it is easy to replace one process with another. For example, the solver processes could be replaced by others that implement different run-time strategies. The database component could be replaced by one that uses another searching algorithm, a different file structure, or even a different physical device. In fact, the process structure used by Port Prolog was adopted from a sequential Prolog on Port. The difference between the two structures lie in the solver processes and the parser; sequential Prolog's parser need not handle guard sequences and its solver implements the ABC algorithm.

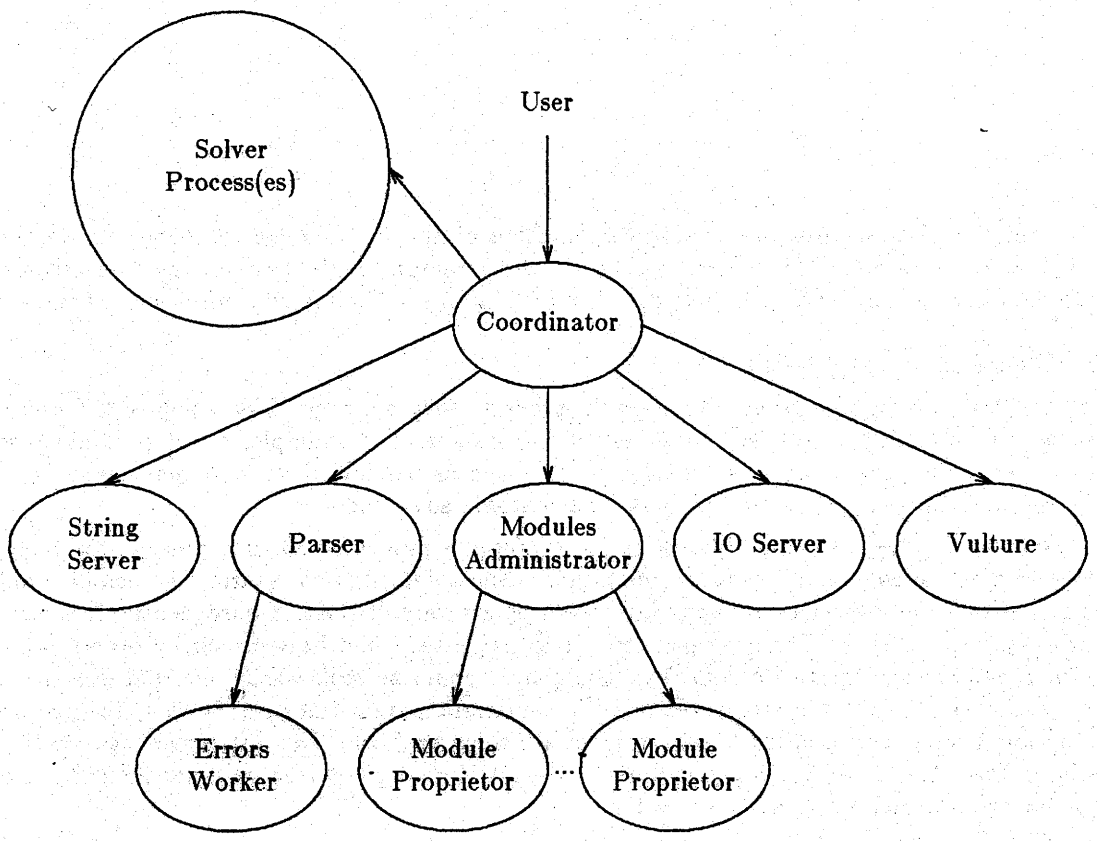
We give a brief overview of each process in the interpreter here; these processes are detailed in subsequent sections. When the interpreter is started up, the *coordinator* process is created. The coordinator creates the rest of the interpreter and oversees the handling of queries from the user. Queries are entered via the Port Prolog window, which is managed by the *io server*. The io server passes the query in clear text to the *parser,* which transforms the input into internal code understandable by processes in the rest of the system. During this transformation, the parser enlists the help of the *errors worker* and the *string server*. If errors are encountered, the parser asks the errors worker to inform the user of his mistakes. Strings of characters are bulky and inefficient to handle. To compensate for these unfortunate characteristics of strings, the string server maintains a string table that maps strings to compact representations that are easier to manage. Processes that manipulate strings use these compact representations. The internal code produced by the parser is passed on to the solver component for evaluation.

In Port Prolog, the solver component consists of a hierarchy of processes with three generic process types: *conjunction-process, goal-process* and *clause-process*. The functions of these processes are described in Chapter 4. In order to evaluate a query, these processes require information from the database of clauses.

Management of the database is the responsibility of the *modules administrator* and the *module proprietors*. Each module of source is managed independently by a module proprietor. There are as many proprietors as there are modules in the source. The solver component asks its local module proprietor to search for a predicate. If it is not found, then the solver asks the modules administrator to name the module proprietor which exports the predicate. If such a module proprietor is found, it becomes the new local proprietor and the solver converses with it until another "switch" is necessary.

The interpreter is terminated at the request of the user. The io server calls upon the *vulture*

Figure 6-1: Genealogical process structure of Port Prolog



process to destroy all the processes created during the invocation of the interpreter.

6.2 Internal Code

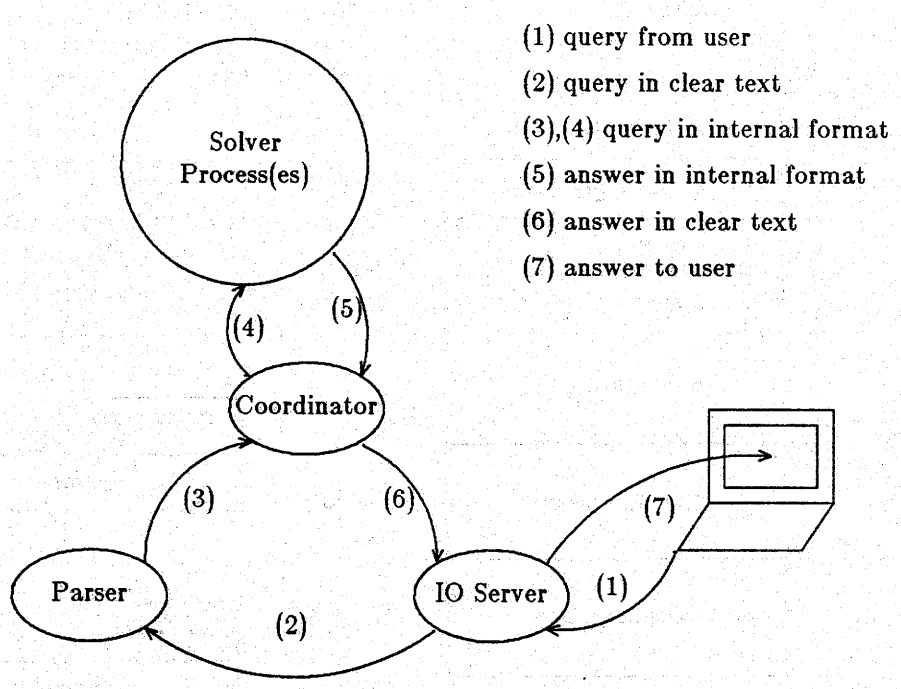
The parser translates source into *internal code* that is used by other processes in the system. Because this internal code is used by and communicated between processes, its design must be modular, compact, and efficient to use by different processes.

The following discussion describes the objects of internal code format and how they relate to each other. The first byte of an object identifies its type. There are two categories of objects — those with fixed sizes and those with variable sizes. The second field of a variable size object contains the size of the object as measured in bytes. All "pointers" embedded in an object are relative offsets.

- **clause.** The largest object of internal code format is a *clause*. All other objects are embedded in clauses. A clause contains either the definition of a procedure or a query. It consists of four major components: procedure head, guard sequence, goal sequence and accounting information. Accounting information refers to the size of the clause (in bytes), number of variables in the clause, and pointers to the locations of the guard and goal sequences. A pointer field (NEXT_PROCEDURE) is reserved so that clauses belonging to a predicate can later be put into a linked-list in the predicate table.

- **call.** A *call* object contains the body of a call. The call body is either an atom or a functor. The body can also be an integer, in which case it is a system call and the number denotes the code identifying the system call.

Figure 6-2: Processing of a query



- **list.** A *list* object consists of two main halves: the head and tail of the list. The head is a term while the tail may contain one or more sequentially packed terms.

- **functor.** A *functor* object contains the predicate name, arity and arguments of the functor. The name is in string descriptor form. The arguments are stored sequentially in the object.

- **integer, atom, character.** There are three simple objects: *integer, atom,* and *character*. An integer object stores the value of a decimal number; an atom object stores the string descriptor of the atom's name; and a character object stores the ascii code of the character.

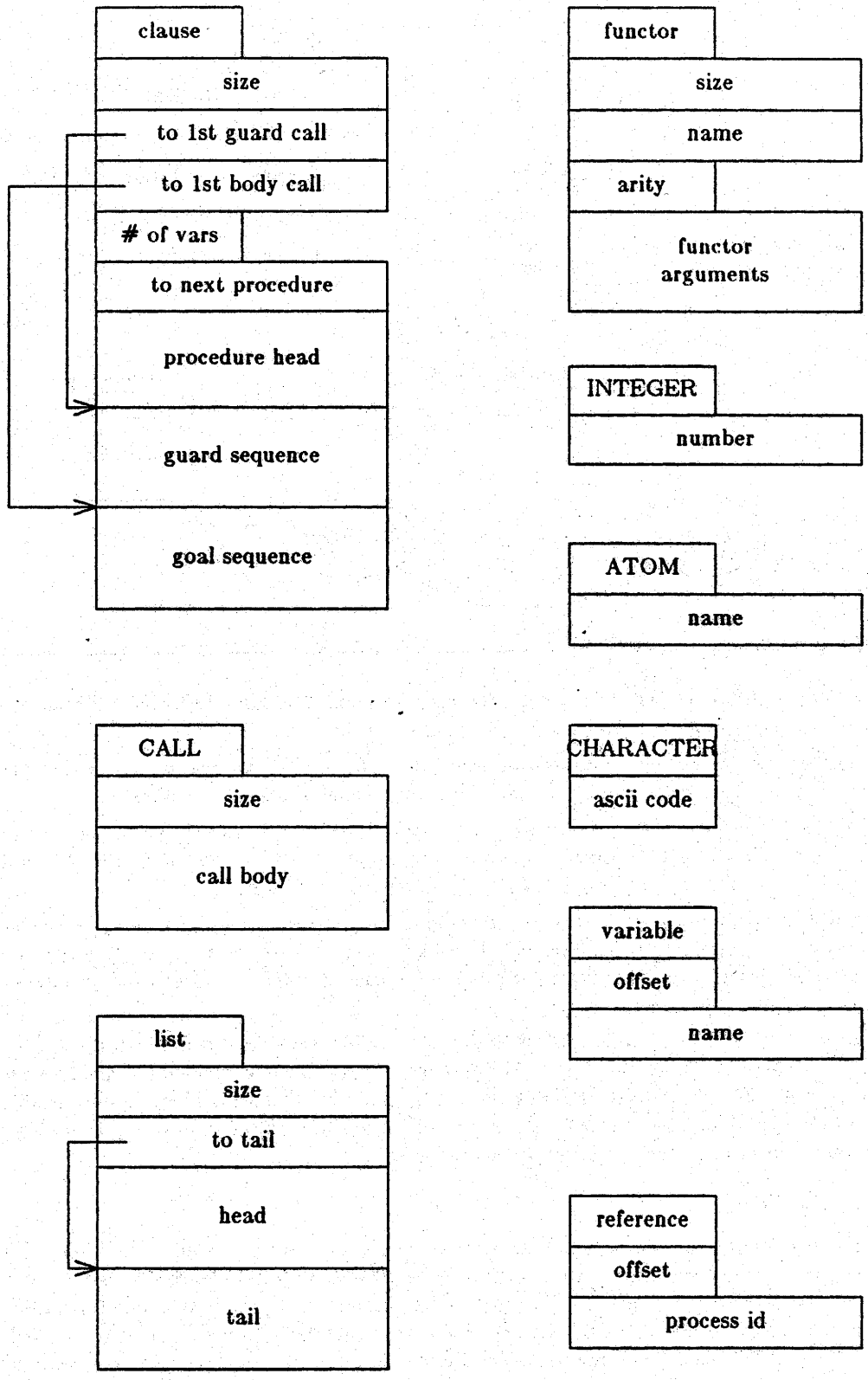
- **variable.** A *variable* object stores information about a variable — this includes the type of the variable (read-only or normal), the numeric offset of the variable with respect to other variables in the clause, and the string descriptor of the variable's name. Variable objects are used for variables that have not yet been instantiated.

- **reference.** A *reference* object is used for a variable that has been instantiated to another unbound variable. There are also two types of reference object, one for read-only variables and one for ordinary variables. A reference object contains both the numeric offset that identifies a variable within a clause and the process id of the potential producer of values for the variable. (For a description of processes as potential producers, see Chapter 4, Section 4.6).

6.3 The .code Tree

The interpreter creates a *.code tree* of files immediately beneath the source root file. This tree is used by the interpreter to store information about the source so that the source need not be referenced (and re-compiled) every time it is used. This external storage is also used to avoid having to load the entire source into memory. Only clauses that are accessed are loaded. Though not implemented, an algorithm could be devised to delete from memory those clauses not in use and loaded them when referenced. This means that Port Prolog programs whose total size exceeds mainstore can still be executed. The current physical memory limit in the IBM PC Port environment is 64 Kbytes of memory for the data of a

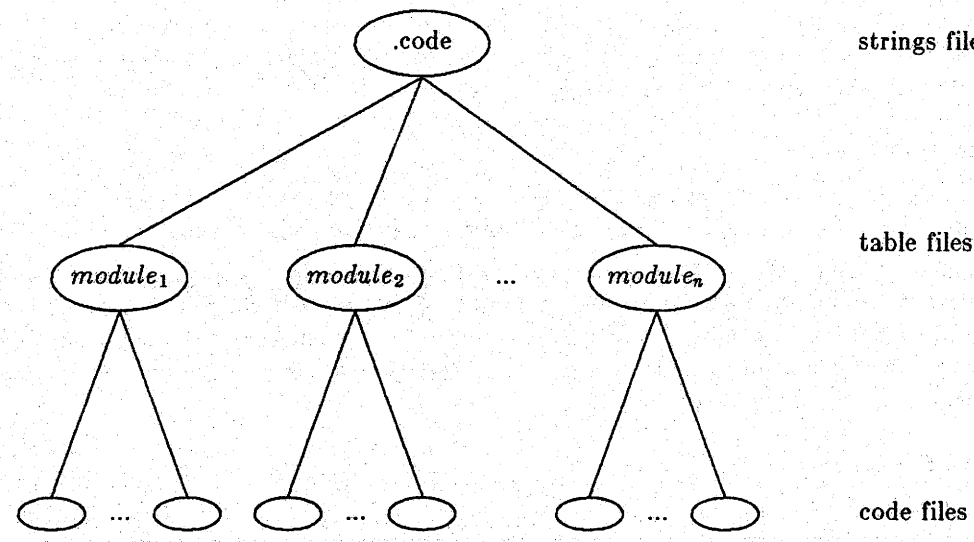
Figure 6-3: Internal code format



process.

There are three kinds of files in this .code tree: *strings* file, *table* files, and *code* files. Files in this tree have the Port file type "code86."†

Figure 6-4: The .code tree



These files are arranged hierarchically (as shown in Figure 6-4) with the strings file as the root, the table files as the children of the strings file, and the code files as the grandchildren (leaf nodes). The following discussion defines each type of file and its purpose.

- **strings file.** The strings file has the filename *.code* and is located directly beneath the source root file. It contains the string table used by the rest of the files in this tree. All strings in the table and code files are in the form of a *string descriptor* — a unique, compact, coded representation of a string (see Section 6.6). This file contains one-to-one mappings of string descriptors to strings.

- **table files.** For every module referenced by the .holons file and the source module, there is a table file (located under the strings file) that contains information about the clauses stored in that code files of the module. There is an entry in the table for every code file in the module (i.e. for each code file located underneath the table file). This table is loaded into memory at initialization time and is used to determine whether a predicate is in the module. In this way, since the table provides an accurate picture of what is stored in the code files, no superfluous file activity is required.

- **code files.** There is a code file for each predicate in the module. A code file stores the code for all the clauses of a predicate. The format of a clause in a code file is as specified for the internal code object *clause*.

† The only criterion for choosing a file type for these files is that it be different from "holon" and "predicate" so that the process that compiles a Port Prolog program can distinguish between files that contain predicates and those with internal code format. The type "code86" was chosen because it was the only type available at the time the interpreter was designed whose name vaguely described the nature of the contents of the files. The new Port file type "binary" may be more appropriate.

6.4 Coordinator

When the interpreter is started up, a *coordinator* process is created which is responsible for creating and coordinating all other processes in the interpreter. The coordinator creates the vulture, the string server, the parser, the modules administrator, and the io server. Each of these processes is then initialized by the coordinator.

After the initialization phase, the coordinator is ready to collect queries from the user. The coordinator obtains a conjunction of goals from the user via the parser. It then forwards the goals to the solver process(es) for solving. When an answer is produced, the coordinator translates it to an ascii representation and asks the io server to print it in the window. The coordinator is then ready for another conjunction of goals from the user. When the user decides to terminate the session, the coordinator tells the io server to destroy the interpreter's window and processes.

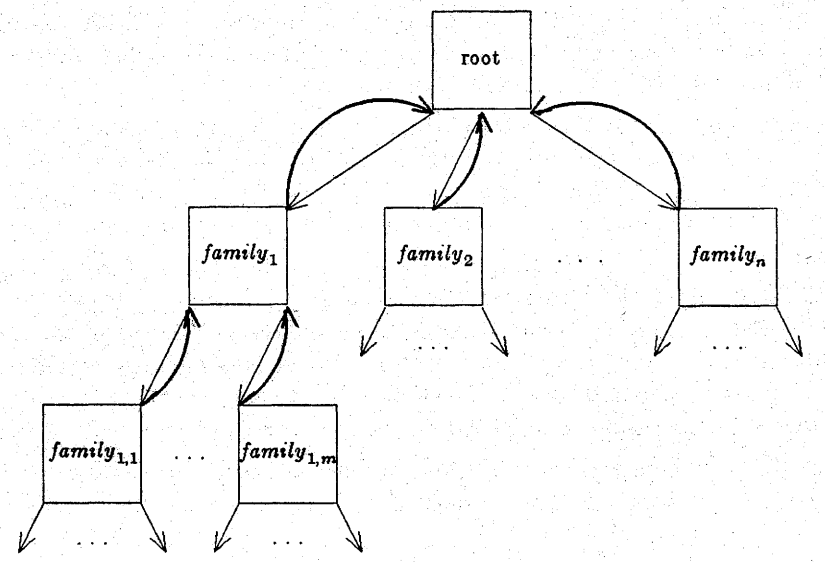
6.5 Vulture

The vulture process, as the name suggests, is a process that waits for other processes to die and then "cleans up" after them. It is created by the coordinator process when the interpreter is started up.

The main task of the vulture is to keep track of the kinships among processes so that a *family* of processes can be identified and destroyed. A family of processes is defined as a process and all of its descendants. Destruction of a family of processes is necessary for eliminating process trees of clause-processes when a candidate clause-process is found. This destruction mechanism also facilitates destroying the interpreter's processes when it terminates.

6.5.1 Data Structures

Figure 6-5: Vulture process tree



- **vulture process tree.** Relationships among processes are remembered using a threaded n-ary tree. Subtrees of the vulture's process tree correspond directly to Port process "family" trees. Nodes in the n-ary tree are added as processes are created and deleted as processes are destroyed. The root of the vulture's process tree is reserved so that more than one tree of processes can be maintained. In other words, several disjoint families that are not created by the same parent process can be accommodated by

naming processes that are starting new process trees as the root's children. Hence, the vulture can record up to "n" independent families.

Each node in the vulture process tree has a process id identifying its corresponding process, a list containing pointers to son nodes, and a thread pointing to its parent node. The thread is to facilitate updates to the parent node when son nodes are deleted.

6.5.2 Requests

Processes of the interpreter that do the creation and destruction are responsible for sending requests to the vulture so that the tree can be updated. The vulture handles four types of request: CREATE_PROCESS, DESTROY_PROCESS, DESTROY_FAMILY, and DESTROY_ALL.

- **CREATE_PROCESS.** Given a process id, the vulture considers the sending process (the one that sent the CREATE_PROCESS request) to be the parent of the named process. An entry in the tree is created for the new child process. If the parent process does not own a node in the tree, then it becomes one of the root's sons. The clients of the vulture are responsible for ensuring that no cycles develop in the tree. The vulture does not check for loops in its tree.

- **DESTROY_PROCESS.** Given a victim's process id, the vulture removes the victim's entry in the tree and destroys it. The victim can only be destroyed if it has at most one son. If it has a son, then the son is promoted to its position — in other words, the victim's son becomes the son of victim's parent. This request is used when a goal-process is no longer needed (i.e. when a candidate clause-process for the goal has been found).

- **DESTROY_FAMILY.** Given a process id, the vulture removes the subtree rooted by the given id and destroys all processes in that subtree.

- **DESTROY_ALL.** The vulture destroys all processes recorded and destroys itself.

These requests may be issued by *any* process. Requests are processed using tree traversal functions which insert and delete entries from an n-ary tree.

6.6 String Server

The string server maps sequences of ascii characters to *string descriptors* and string descriptors to sequences of characters. A string descriptor is a 16-bit value that uniquely identifies a string. For example, "this is a string" may be mapped to the string descriptor "$2AB4_{\text{hex}}$."

At initialization time, the string server is given the name of a file — the strings file — and a bit indicating whether the specified file contains an up-to-date string table. A string table contains the mapping between strings and string descriptors. If the strings file is up-to-date, its table is read in and transformed into an internal string table; otherwise, the name of the strings file is remembered for later reference and the internal string table is initialized as empty.

6.6.1 Requests

When initialization is completed, the string server is ready to handle three types of request: GET_STRING, SAVE_STRING and WRITE_STRING_TABLE.

- **GET_STRING.** Given a string descriptor, the string server returns the sequence of characters identified by the string descriptor. A null string is returned if the string descriptor is not in the string table.

- **SAVE_STRING.** Given a sequence of characters, the string server saves the sequence in the string table and creates a string descriptor for it. No new entry is created if the string is already present in the string table. The string descriptor is returned to the sender. An "invalid" string descriptor is returned if the string table is full.

- **WRITE_STRING_TABLE.** The string server writes out its internal string table into the strings file specified at initialization time. This request is usually issued after compilation of the source.

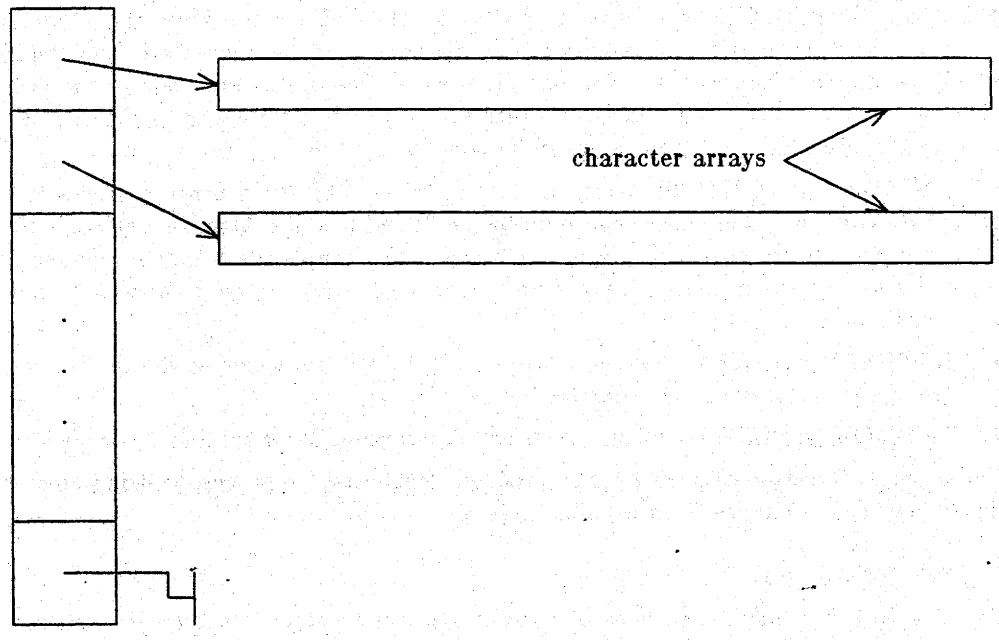
These requests may come from *any* process. The string server is destroyed when the interpreter is terminated.

6.6.2 Data Structures

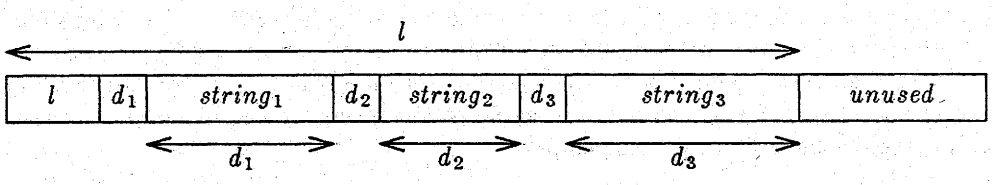
● **string table.** The major data structure is the *string table;* it is an array of *character arrays*. Initially, only one array is allocated; other character arrays are created as needed. The first two bytes of a character array is reserved for the number of bytes being used in the array.

Figure 6-6: String table and string descriptor format

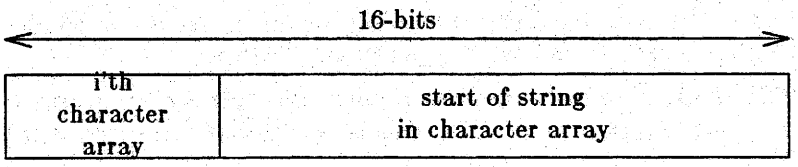
(a) String table



(b) Character array



(c) String descriptor format



Each entry in a character array is represented by a length byte followed by the string. Entries are stored sequentially in the order they are saved.

● **string descriptor.** String descriptors are used for easy equality comparison and compact representation of strings. A string descriptor uniquely identifies a string *and* helps locate the string so that the mapping of string descriptor to string can be done as efficiently as possible. A string descriptor

consists of two parts: identification of the character array that contains the string, and the location of the string in the array.

6.6.3 Files

The string server reads (writes) from (to) the strings file. The strings file contains the string table of all the strings that the string server has been asked to permanently record. Each entry in the file has the following format:

<string descriptor> *<string length>* *<ascii string>*.

6.7 IO Server

The io server‡ manages the Port Prolog window. The form of the Port Prolog window is outlined in Section 5.3.1 in Chapter 5. Any process wishing to do any input (output) from (to) the window must go through the io server.

The io server is created by the coordinator process at interpreter start-up time. The io server is given the pathname of the source tree so that it can put the name on the title line of the window.

6.7.1 Requests

The io server then starts serving requests from either the keyboard or clients wishing to do input/output.

- **INPUT_ARRIVED.** This request is sent by the keyboard. Upon receiving this request, the io server checks the contents of the message, which may contain cursor position information and keys pressed by the user, and act appropriately to update the window.
- **READ_BYTES.** This request is sent by any process that wants to receive input from the keyboard. Input is returned line by line — the io server must wait until the user hits a carriage return before forwarding the text. This allows the user to edit the input line before submitting it to the interpreter. If a line of input is available, it is dispatched to the sender of the READ_BYTES request. Otherwise, the requester is suspended until the user enters a new line.
- **APPEND_BYTES.** This request is sent by any process that wants to output to the window. The string of bytes that accompanies the APPEND_BYTES message is displayed on the window.
- **TAKE_OFF_EH.** This request is sent by the coordinator to the io server to terminate the interpreter session. The io server destroys the window, then tells the vulture to destroy all processes registered with it. This termination procedure is also followed when the user selects the QUIT operation.

6.8 Parser

The parser is responsible for transforming user input (and source) into an internal format that is compact and can be efficiently handled by other processes of the interpreter. The syntax of the Port Prolog language is given in Chapter 5. There are four main components to the parser: syntactic analyzer, lexical analyzer, error handler, and request processor.

6.8.1 Lexical Analyzer

The lexical analyzer, *lex*, transforms user input into *tokens*. Symbols, which may be composed of several characters, are recognized by lex and translated into tokens. Invalid symbols that cannot be mapped onto tokens are noted but not explicitly reported; instead, an "invalid" token is generated. All errors are reported during the syntactic phase. A token is made up of a *token type* and, optionally, a

‡ The io server is a version of the *Invoker* program written by the Software Portability Lab. The major difference is that the io server has been relegated the additional task of asking the vulture process to destroy all processes when the user hits the QUIT operation. The Invoker, however, responses to QUIT by destroying its creator and itself, which is insufficient for this case. Another difference between the Invoker and the io server is their initialization procedures. Having a process to manage the Port Prolog window, instead of just using the Invoker, has the advantage that operations such as TRACE, COMPILE, LIST, could be added later on.

token value. For example, a token of type "INTEGER" has a decimal number as its value. Tokens are generated upon demand. The interface to lex is through a function that retrieves the next token in the current input stream.

In addition to generating tokens from the input stream, lex also remembers a token's position with respect to the current input state to aid in the diagnoses of possible lexical and syntactical errors. This information includes the input line containing the token, and the line number and character position at which the token is found.

6.8.2 Syntactic Analyzer

The syntactic analyzer employs a top-down recursive descent algorithm for parsing Port Prolog clauses. The control flow of the analyzer follows strictly that of the syntax. For example, at the highest level, it is looking for a clause. A clause may be either a procedure or a query. In order to find a procedure, it must first locate a head and subsequently guard and goal sequences. To locate the head, the analyzer must obtain a literal that is either an atom or a functor, and so on. The simplicity of Port Prolog's syntax permits the analyzer to use such a straight-forward algorithm. The BNF production rules for Port Prolog syntax is given in Section 5.2 in Chapter 5.

The syntactic analyzer acquires tokens from the lexical analyzer and checks for their compliance with the syntax rules. Tokens are translated into internal code. The format of this internal code is given in Section 6.2.

6.8.3 Error Handling

Errors may be encountered during parsing and tokenizing. In lexical analysis, only one type of error can occur — an invalid symbol in the input stream. While parsing, the syntactic analyzer may be confused by the incoming stream of tokens if they do not conform to the syntax rules. When an error is encountered (regardless of its type or where it occurred), remaining tokens are discarded until a clause terminator is found, at which point parsing of a new clause begins. If input is being accepted from the Port Prolog window when the error occurred, the input line containing the error is echoed with the offending token highlighted and accompanied by an error message explaining the cause of the error. If the error occurred in a source file, the parser sends an error code and sufficient information to describe where the error occurred to an *errors worker*. The parser creates an errors worker if one does not exist. It is then up to the errors worker to inform the user of his mistake.

This approach of notifying the user (cf. messages in the Port Prolog window) was adopted because (1) it ensures that the error message is visible, and (2) it provides more help to the user than simply a message. During compilation, the interpreter may produce many lines of output showing the predicates being compiled. Oftentimes, the user would set the window into "scroll" mode and let these messages scroll off the window. If an error was encountered and the corresponding message appears in the Port Prolog window, the message scroll off the window before the user notices it. Making the error message visible through a separate window which only appears when an error is encountered ensures that the error message is not overlooked. Because the errors worker is dedicated to displaying messages, it can also aid the user in correcting the error. The function of the errors worker is described in Section 6.9 in this chapter; the layout of the errors window is described in Section 5.3.2 in Chapter 5.

6.8.4 Requests

At initialization time, the parser is given the string server's process id and the io server's process id. The parser then awaits requests from any process. Its clients are the coordinator and the module proprietors. The coordinator wants to parse queries from the user whereas the module proprietors wish to parse source files. The parser handles three types of request: SET_INPUT_FILE, PARSE_INPUT, and PARSE_EXPORT.

- **SET_INPUT_FILE.** The parser makes the file specified in this message the "current input stream." If no file is specified, the Port Prolog window is the default stream. Input is obtained from this stream until another SET_INPUT_FILE request is received or until the end of this file is reached. When the end of a file is reached, the current input stream is switched to the Port Prolog window until another

SET_INPUT_FILE is received. This message also contains a bit telling the parser to expect either queries or procedures.

- **PARSE_INPUT.** The parser reads from the input stream and transforms a clause into internal code. The parser then returns to the sender of the request the internal code, and the name and arity of the clause's head predicate. If input is exhausted, the sender is notified of the condition.

- **PARSE_EXPORT.** The parser reads the export list from the input file and returns it to the sender. There is one export list per file.

The above requests are used in the following predefined way. During compilation, a module proprietor that encounters a module with a root file of type "holon" first sends the parser the pathname of the root file along with a SET_INPUT_FILE request. It then sends a PARSE_EXPORT request in order to obtain the export list. Finally, for each source file, the proprietor sends a SET_INPUT_FILE request to set up the file as input, and then repeatedly sends PARSE_INPUT requests until the file is exhausted. To accept queries from the user, the coordinator first sends to the parser a SET_INPUT_FILE request with a null pathname and a bit indicating queries are to be parsed. This ensures that the parser takes its input from the Port Prolog window. The coordinator then sends PARSE_INPUT requests to receive the queries.

6.8.5 Files

The parser reads from source files of type "predicate." The contents of these files follow the syntax of the Port Prolog language as described in Chapter 5. The parser also reads export lists from files of type "holon." The format of a holon file is a sequence of terms with the form

<name> (*<arity>*).

6.9 Errors Worker

The errors worker is created by the parser when the parser encounters a syntax error in a file. The errors worker does not know about any process in the interpreter. When it is created, it puts an errors window on the screen. The format of this window is described in Chapter 5. The errors worker's duties are to buffer error messages sent by its creator and to display them as directed by the user.

6.9.1 Data Structures

- **error descriptor**. Error messages sent to the errors worker are recorded in a doubly linked-list of error descriptors. An *error descriptor* contains sufficient information to display an error message in the format shown in the errors window. This includes an error code that describes the nature of the error. The error code is mapped to a corresponding error message when the descriptor is displayed. Data pertinent to *where* the error occurred are also in the descriptor. This includes the name of the file containing the error, the error's (x,y) coordinate within the file, and the input line that contains the error. Finally, an error descriptor has forward and backward pointers to doubly link it to the list of error descriptors. This enables the user to forward and backward traverse over the errors.

6.9.2 Requests

The errors worker handles requests from the parser and from the keyboard.

- **APPEND_BYTES.** This request is sent by the parser when an error is encountered. The message consists of an error code, input line in which the error resides, the name of the file containing the line, and the error's (x,y) position in the file. This information is packed into an error descriptor and appended to the list of descriptors.

- **INPUT_ARRIVED.** This request is sent by the keyboard on behalf of the user. There are four operations that the worker has to process: QUIT, EDIT FILE, NEXT ERROR, and PREVIOUS ERROR.

  - QUIT. The errors worker destroys the window and itself. If the parser encounters subsequent errors, a new errors worker is created.

  - EDIT FILE. The worker starts up an editor to edit the file specified by the pathname of the current message being displayed.

  - NEXT ERROR (PREVIOUS ERROR) is only operative if a next (previous) message exists. The

worker responds by displaying the next (previous) message in the list.

6.10 Modules Administrator

The modules administrator is responsible for setting up the module proprietors with modules to work on and for providing name look up service of modules.

At initialization time, the administrator acquires the vulture's id, the string server's id, and the pathname of the source tree. It then "sets up" a module proprietor with the source tree's pathname. This involves creating the proprietor and supplying it with the appropriate parameters. When this first proprietor has been initialized, the administrator proceeds to initialize other modules used by the Port Prolog program. It looks in the .holons file of the program to find the pathnames of the other module directories. The administrator sets up a module proprietor for each pathname in the .holons file. To "set up" a module proprietor, the administrator first creates a module proprietor and informs the vulture of the creation. The administrator then sends the proprietor two pathnames: (1) the module's source tree (2) root of the tree of where the "code" of the module is or is to be stored.†

When a module proprietor finishes its initialization duties, it replies to the modules administrator with a linearized (or flattened) export table containing all predicates that are exported by the module.

6.10.1 Data Structures

The modules administrator uses three major data structures: *modules list, predicate table,* and *flattened predicate table.*

- **modules list.** A *modules list* is a linear linked-list that keeps track of all modules in the Port Prolog program. Each node in the linked-list has four fields: the module's name in string descriptor form, the corresponding module proprietor's process id, a pointer to a predicate table containing the predicates being exported by the module, and a pointer to the next entry in the modules list. The order in which the modules list is sorted is unimportant because the length of the list is not expected to be very large.

- **predicate table.** A *predicate table* consists of a table skeleton and clauses. To facilitate the search for a predicate, the skeleton of the table is separated into *predicate name entries* and *arity entries.* The predicate name entries form a linear linked-list; each predicate name entry has a linked-list of arity entries — one entry for each unique arity. Clauses are accessible via arity entries.

A predicate name entry contains four fields: the string descriptor of the predicate's name, a pointer to the head of the list of arity entries, the length of the arity list, and a pointer to the next predicate name entry in the list.

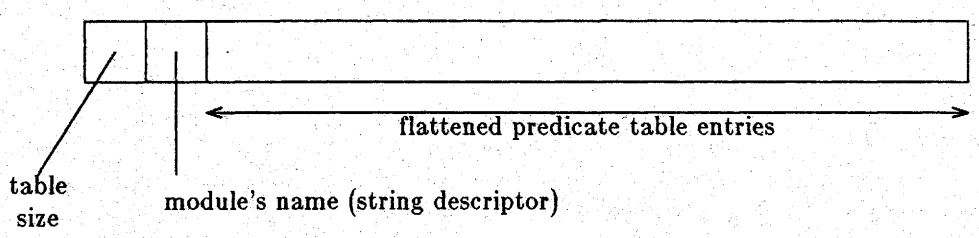
Arity entries are ordered in a linked-list by their arity values. An arity entry contains pointers to the first and last clauses of the predicate, the arity of the clauses, and a pointer to the next arity entry in the list. Since the size of the arity is limited to twenty arguments, the upper bits of this field can be reserved for recording the properties of the clauses in this entry. Only one bit has been defined so far — the global/local bit. The clauses are global if the bit is on.

- **flattened predicate table.** The *flattened predicate table* is introduced in order to facilitate the transmission of predicate tables between processes and the storing of predicate tables in files (see Section 6.3 in this chapter). Figure 6-7 illustrates the format of a flattened predicate table. Basically, a flattened table is an array of bytes. The first two words are reserved for the size of the table and the name of the module (in string descriptor form) that owns the table. The rest of the array contains *flattened predicate table entries.* Each such entry consists of a predicate name (in string descriptor form), the number of arity entries, followed by the arity entries. An arity entry is an 8-bit value that contains the global/local bit and the arity.

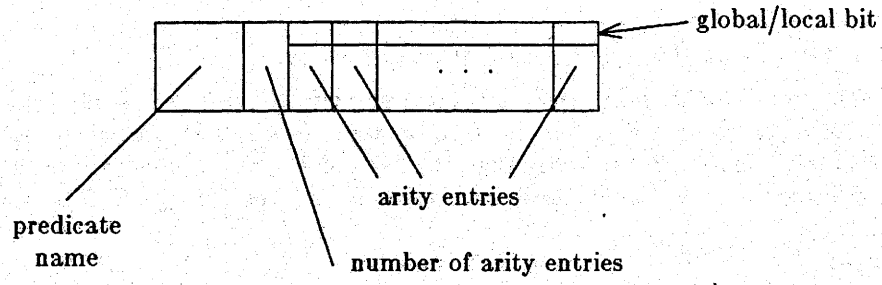
† (2) is the name of a table file located immediately below the strings file. See Section 6.3 in this chapter.

Figure 6-7: Flattened predicate table

(a) Flattened predicate table



(b) Flattened predicate table entry



6.10.2 Requests

After setting up all the module proprietors, the administrator is ready to handle two types of request: FIND_MODULE and NAME_LOOK_UP. These requests come from the solver process(es).

- **FIND_MODULE.** Given a predicate's name and arity, the administrator searches through its export predicate tables and replies to the sender with the modules list entry of the module that exports the target predicate. The administrator replies with an "invalid" modules list entry if the predicate is not found,

- **NAME_LOOK_UP.** Given a module's name (in string descriptor form), the administrator replies with the corresponding module proprietor's process id. If the given name is not in the modules list, then an "invalid" process id is returned.

The modules administrator is terminated when the rest of the processes of the interpreter are destroyed (when the user exits the interpreter).

6.10.3 Files

The administrator reads from the .holons file of the Port Prolog source tree. The .holons file is located immediately beneath the source root file. Each line of the file contains a pathname of a module which is to be included to the program.

6.11 Module Proprietor

A module proprietor is responsible for compiling the source of a module and for storing the code of the compilation so that other processes can access them. Several module proprietors may be created (by the modules administrator) depending on the number of modules used by a program.

When a module proprietor is created, it receives two pathnames from the modules administrator: (1) the root of module's source tree, and (2) the root of the tree of where the code is stored (i.e. the pathname of the module's table file). It also receives information on whether the source needs compiling.

For source that requires compilation, the proprietor first checks the file type of the module's root file; it may be one of two types: "predicate" or "holon." If it is of type "holon," the proprietor asks the parser to parse and to report the export list in the root node. The proprietor then creates entries in its predicate table for every predicate in the export list and marks them as being "global." The proprietor then traverses the source of the module and successively asks the parser to process each file. The internal code generated by the parser is shipped back to the proprietor, which stores the code for each predicate in a separate code file. The code file's filename is obtained by concatenating the name and arity of the predicate into a string. For example, the clauses for the predicate *append* with three arguments are stored in the file "append.03." Code files for a module are stored underneath the table file of the module. When the entire source has been compiled, the proprietor creates a flattened predicate table from its predicate table and writes the flattened version into the table file.

If the source does not require compiling, the proprietor reads the flattened predicate table from the table file and builds a predicate table from the flattened version.

The module proprietor then returns the flattened predicate table to the modules administrator. The flattened table can now be destroyed as it is no longer needed. The predicate table, however, remains because it can be searched more efficiently than the flattened table. The predicate table is used to process subsequent search requests. If a predicate is not in this table, then it is not in this module.

6.11.1 Requests

After the initialization phase, the proprietor awaits requests to search for a predicate. These requests come from the solver process(es).

- **SEARCH.** Given the name and arity of a predicate, the proprietor searchs through its predicate table for the predicate's entry. If the entry is not present, the search (request) fails. If the entry is found, the proprietor checks whether there are any clauses stored under the entry. If there are, these clauses are returned to the sender of the SEARCH request. Otherwise, clauses are loaded into memory from the corresponding code file and then returned to the sender. For example, if the target predicate is *append* with three arguments, the file "append.03" is read and the clauses stored in memory.

The proprietor is destroyed when the interpreter terminates.

6.11.2 Data Structures

The major data structures used by a module proprietor are the predicate table and the flattened predicate table. These are described in Section 6.10 in this chapter.

6.11.3 Files

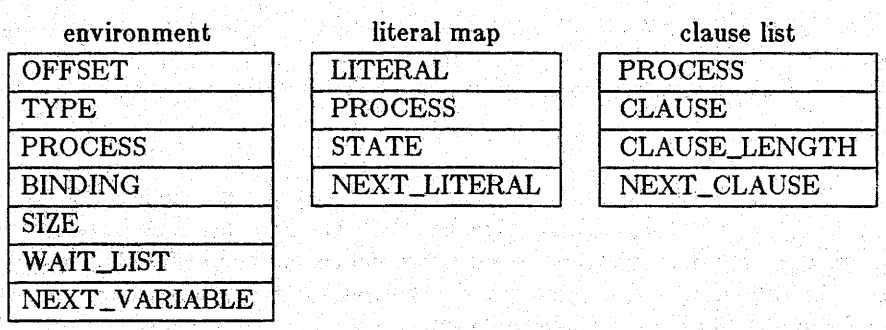
A module proprietor reads source files of type "predicate" and possibly an export list from a file of type "holon" in the source tree. The formats of these files are described in the previous chapter.

A proprietor reads and modifies the table file and code files belonging to its module. The table file has the same format as the flattened predicate table data structure. A code file contains clauses whose format is described in Section 6.2 in this chapter.

6.12 Solver Processes

There are three types of solver processes: *conjunction-processes, goal-processes, clause-processes.* Their functionalities are described in detail in Chapter 4. The following describes their implementation and may contain redundant information already discussed in Chapter 4. They are included here for completeness.

Figure 6-8: Solver processes' data structures



6.12.1 Data Structures of the Solver Processes

- **environment.** The major data structure is the *environment.* It is used by all three types of processes. Goal processes make only temporary use of environments to store bindings for read-only variables. An environment is a linked-list of entries that record information about a single variable. Each entry consists of the variable's offset within its clause, its type (FREE, REFERENCE, etc), the process id of its producer, a pointer to its binding, the size of its binding, a pointer to a list of processes waiting for the variable's binding, and a pointer to the next variable in the list.

- **literal map.** The *literal map* is a linear linked-list used by the conjunction-process. For each literal, it stores a pointer to the internal code of the literal, the literal's state (of evaluation), the process id of the process solving the literal, and a pointer to the next literal in the list. The process specified by the id is either the goal-process or the candidate clause-process of the literal.

- **clause list.** The *clause list* is a linear linked-list used by the goal-process to remember all of its child clause-processes. The CLAUSE and CLAUSE LENGTH fields are used temporarily to store the clauses as they are accumulated by requests to the module proprietor, and before they are sent to their corresponding clause-processes. The PROCESS field stores the process id of the clause-process.

6.12.2 Conjunction-Process

To solve a query from the user, the coordinator creates a conjunction-process. This conjunction-process then initiates other processes to help solve the goal. Subsequent conjunction-processes are created by clause-processes. The conjunction-process first gets the literal-list from its parent and creates an environment for all the variables in the literal-list. It then breaks up the literal-list and creates a goal-process for each literal. A literal map is constructed to store information about each literal.

6.12.2.1 Requests

The conjunction-process, after creating the goal-processes, is ready to handle five types of request:

- **NEW_BINDING.** Given the variable's offset, the conjunction-process searches for the variable in its environment. If the variable is found, the binding and size of the binding is recorded.

- **NEED_BINDING.** Given the offset of a variable, the conjunction-process searches for the

variable in its environment. If the variable has a binding, it is returned immediately to the sender of the request. Otherwise, the sender is added to the waiting list of the variable.

- **FAIL.** The conjunction-process terminates by sending a FAIL message to its creator, and destroys itself and all of its descendants.

- **SUCCESS.** The conjunction-process searches through its literal map for the entry containing the process id of the sender. If found, the STATE of this entry is set to SUCCESS. The conjunction-process then decrements its count of the number of literals yet-to-be solved. If this counter is zero, then the conjunction-process terminates successfully by sending a SUCCESS message to its parent and destroying itself and its descendants.

- **COMMIT.** The conjunction-process searches through its literal map for the entry containing the process id of the sender. If found, the STATE of the entry is set to COMMITTED. The conjunction-process also records the accompanying process id (i.e. the candidate clause-process) in its PROCESS field.

6.12.2.2 Status

The above description of the conjunction-process is a partial implementation of the conjunction-process described in Chapter 4. It handles all but the *patch binding* incoming message. It handles *need binding* messages, but can only deal with one variable per message. The conjunction-process implemented does not send *need binding* messages to poll other processes for newly instantiated non-read-only variables. This means that a goal-process needing a variable binding for a read-only variable may have to wait until the process producing the variable binding decides to send to the conjunction-process, instead of having the conjunction-process polling the producer process for the binding. This may mean a longer delay period between the time the variable gets bound and the time the goal-process receives the binding.

6.12.3 Goal-Process

A goal-process is created by a conjunction-process to solve a literal. The goal-process first gets the literal from its parent and creates a temporary environment for the variables in the literal. For every read-only variable, it sends a NEED_BINDING request to the producer of the variable. The producer is the parent if the variable is free (i.e. not reference), and the process named by its tag if it is a reference variable. After all read-only variables are bound, their bindings are applied to the rest of the literal and the environment is freed.

The goal-process then retrieves all clauses from the database that may unify with the literal. These clauses are stored in the clause list. Then, for each clause in the clause list, the goal-process creates a clause-process and records its process id in an entry in the clause list. A clause-process is not created immediately after each clause retrieval because that would make the order of clause retrieval important. For example, the first clause-process may already be executing before the last clause-process was even created. Hence, our creation order tries to promote fairness and indeterminacy.

6.12.3.1 Requests

The goal-process then waits for messages from its children.

- **FAIL.** If the goal-process has received a FAIL message from all of its children, it sends a FAIL request to its parent and destroys itself. Otherwise, the goal-process decrements the counter of the number of messages to expect.

- **COMMIT.** The goal-process traverses its clause list and for every entry which does not contain the process id of the sender COMMIT, the goal-process destroys the entry's process tree. The sender of the COMMIT request is now the candidate clause-process. The goal-process then sends a COMMIT request, along with the process id of the candidate clause-process, to its parent. It then destroys itself.

6.12.3.2 Status

The goal-process as described in the computation model in Chapter 4 has been fully implemented. However, the goal-process has yet to integrate the features of modules. In other words, it only queries one module proprietor for clauses.

6.12.4 Clause-Process

A goal-process creates a clause-process for every clause whose head has the same predicate name and arity as its literal. When created, a clause-process first tries to unify the clause head with the goal literal. If unification fails, it informs the goal-process. Otherwise, the clause-process applies the substitutions to the rest of the clause and then tries to solve the guard sequence.

To solve the guard sequence, the clause-process creates a conjunction-process and passes it the guard sequence. The clause-process then awaits a message from the conjunction-process. In the current implementation, it handles NEW_BINDING, SUCCESS and FAIL requests. If a FAIL request is received, the clause-process sends a FAIL message to the goal-process. A SUCCESS request causes the clause-process to send a COMMIT request to the goal-process, from which it receives the process id of its grandparent conjunction-process. The clause-process then sends a NEW_BINDING request to that grandparent for each variable that has been instantiated. The clause-process handles a NEW_BINDING request in the same manner as the conjunction-process does.

After the guard-sequence has been successfully solved, the clause-process creates another conjunction-process and passes it the goal-sequence. It then awaits the conjunction-process to finish solving the goal sequence. When this child conjunction-process is done, the clause-process gathers the variable bindings and forwards them to its grandparent conjunction-process. Finally, it reports the status of the evaluation to its grandparent conjunction-process and destroys itself. The above procedure is described in detail in Sections 4.2 and 4.5 of Chapter 4.

6.12.4.1 Status

The implementation of the clause-process lacks several features described in Chapter 4. It does not have a complete unification procedure. The cases that it does not handle yet are: free-free, reference-free, variable structure-free, variable structure-reference. It also does not handle incoming *need binding* and *patch binding* messages. This means that Port Prolog cannot handle queries in which variables unify each other or unify structures containing variables.

7

Some Performance Measurements

This chapter presents some measurements made to evaluate the performance of Port Prolog. It includes a brief analysis of the measurements. We then compare them with corresponding measurements made on a sequential Prolog interpreter on Port. These measurements hopefully will provide the necessary information for a second iteration on the design of a concurrent Prolog interpreter.

7.1 Measurement Techniques

The measurements presented in this chapter were made using a *Four Channel Event Counter/Timer* [Rapsey 78, Jurchuk 79]. They were obtained by inserting measurement *stubs* into the program to count and time "events" — events being determined by *where* the stubs are placed. These stubs are procedures that write bits to the Counter/Timer device. There are two counters and two timers; all four are referenced via the same address. The position of the bit determines to which channel the bit is sent. The counters/timers can be examined dynamically while the program is running.

The figures in this chapter contain measurements made using the Counter/Timer device. The numbers in each category are the result of ten independent trials which were made with Port Prolog as the only active program in the system. The raw measurements for the trials are given in Appendix A.

The numbers given in this section are based on tests performed on the *stack* program (see Figure 7-1) [Shapiro 83]. This simple program demonstrates OR-parallelism, which is a major contributor to the cost of evaluating a goal. Measurements made on this program also give a calibration of factors such as process management and communication, which are applicable regardless of the benchmark used.

Figure 7-1: Benchmark program

```
stack( S )
   stack( S? [] );

stack( [pop(X)|S] [X|Xs] )
   stack( S? Xs );

stack( [push(X)|S] Xs )
   stack( S? [X|Xs] );

stack( [] [] );


benchmark1:  stack( [push(1),pop(1)] );

benchmark2:  stack( [push(1),push(2),push(3),pop(3),pop(2),pop(1)] );
```

7.1.1 Limitations of the Device Interface

All channels on the Counter/Timer are referenced through a single address. A 0-bit sends a low signal whereas a 1-bit sends a high signal. Therefore, to send a 0-bit to one channel, we would have to know the states of the other channels and provide an appropriate mask so as not to disturb their signals. Because of this limitation and the fact that our measurements involve disjoint processes that do not share global knowledge about the device, we could only use one channel at a time. For example, we can either count the number of process creations or time the length of a process destruction, but cannot do both simultaneously. Since the measurements were taken separately, comparison and correlation of measurements in different categories are subject to small errors.

7.2 Breakdown of Execution Time

Figure 7-2: Evaluation of benchmarks

(a) Execution time

	$benchmark_1$ (ms)		$benchmark_2$ (ms)	
Process Management				
create	124.60	5.40%	270.59	5.37%
setup	174.43	7.65%	368.16	7.30%
destroy	821.14	35.61%	1749.63	34.70%
Sub total		48.66%		47.37%
Communication	328.77	14.27%	702.50	13.93%
Others	854.88	37.07%	1951.30	38.70%
Total	2306.14	100.00%	5042.07	100.00%

(b) Number of processes used

	$benchmark_1$	$benchmark_2$
conjunction-process	4	8
goal-process	4	8
clause-process	10	22
Total	18	38

Figure 7-2a shows the breakdown of the time Port Prolog takes to solve the benchmark queries given in Figure 7-1. Each category of measurements (e.g. "create," "destroy") was collected separately. "Others" and "Sub total" are calculated from the other entries. "Total" is the time the solver processes need to solve the goal. This does not include time for parsing (transforming the user input into internal format) or printing the answer. Each category is explained in detail below. Figure 7-2b shows the number of processes used in each benchmark.

7.2.1 Process Management

The Port *process administrator* is a process which accepts requests to create a process and destroy a process. The cost of using a Port process includes (1) the process administrator creating a process, and (2) initializing the new process' state (data space). The latter is referred to as the "setup" time; it is dependent on the size of the data space and the complexity of the data structures that must be initialized. The

create times shown in Figure 7-2a involve processes whose code are already in memory; hence no input/output is required to load the code; only data space need to be allocated.

The creation time is obtained by placing stubs in the Port process administrator. The destruction time is calculated similarly, except the time for a message-pass is added to each destroy because a destroy request is processed through an intermediate process — the *exception handler*. The time required to initiate the create/destroy request is not recorded in this category; rather, this cost is accounted for in the communication category. For example, before the process administrator handles the create request, a message-pass must be made to forward the create message from the creator to the process administrator. The initiation of a destroy request involves a more complicated procedure which we will not detail here.

From Figure 7-2a, it is evident that process management incurs a large cost. In Port Prolog, a destroy request requires approximately 45ms, a create request 7ms, and process setup 9.5ms.‡ Collectively, these three operations comprise over 47% of the total time the interpreter spends evaluating a goal. This is unacceptably high and suggests the need for a different approach to dynamic process management that uses fewer Port create/destroy operations (see Section 8.4 in Chapter 8).

7.2.2 Communication

The communication entries shown in Figure 7-2a is calculated by counting the number of sends that occur and multiplying that number by the "average" cost of a send (see Appendix A-1). This average is an average based on the priority of the process and the load of the system, and not on the size of the message; the length of the message is assumed to be one byte. There is a cost of 0.03ms for each extra byte sent. Therefore, the values from our trials given in Figure 7-2a are under-estimates because they do not account for the length of the messages being sent.

Communication takes over 13% of the time required to solve a goal. The true communication cost may be as high as 15% or more. Ignoring the process management cost, this is approximately 25% of the total time required to solve a goal. Furthermore, the current version of Port Prolog handles only simple variable bindings, a full scale version which handles more sophisticated bindings (as described in Chapter 4) will push the communication cost higher. Although it is expected that communication will be a major cost factor, and that the Port interprocess communication (IPC) primitives are efficient, considerations must be given to minimize this already substantial percentage, especially if the interpreter is to be distributed over several machines. IPC between processes on different machine is 7 times the cost of local ones [Vasudevan 84].

7.2.3 Other Components

Appendix B contains program "traces" of a conjunction-process, a goal-process, and a clause-process. A trace shows the number of instructions executed by each function in a process. The trace is started right after the process has been setup and terminates when the process is destroyed.

As evident from the traces, a substantial number of instructions are used for memory allocation, which uses the functions: *Zero, Alloc, _Alloc, _Grow_data_segment, Find_a_vector*. This is not surprising as all three types of processes construct many data structures dynamically. This allocation cost can be lowered by statically building the data structures at process setup time, but the cost of initializing a process would then be higher.

Of the three types of process, the clause-process is the busiest. The clause-process spends a considerable portion of its time interpreting the internal code (*Get_procedure_info, Get_function_info*, etc), and performing unification. The performance of the clause-process can be improved by optimizing the code for these functions, possibly by writing them in assembly language.

The goal-process spends most of its time creating clause-processes, retrieving clauses from the database (via the module proprietor), and interpreting the internal code. The performance of the goal-process can be improved by tuning the interface between the goal-process and the database component. In the current version of Port Prolog, clauses are retrieved one at a time from the module proprietor. The

‡ These figures are not atypical in Port, see Appendix A-2.

number of message-passes and process switches can be reduced by packaging all clauses to be transported into one buffer and sending this buffer only. This, however, implies more sophisticated buffer management is needed and may incur more memory allocation costs.

The conjunction-process is the least busy process. This is because it has not been asked to do much. The literal-list in the example that it has to solve consists of only one goal. The conjunction-process spends a major portion of its time interpreting internal code, creating the environment to store the bindings, and creating the goal-process. Like the clause-process, its performance can be improved by optimizing the functions that interpret the internal code.

A measurement which was not taken but which would be very useful is the number of processes competing for the processor. By recording the maximum or average number of solver processes in the ready queue of the Port kernel, we can approximate how much speed up can be attained if multiple processors were used. This statistic could be collected by adding a new Port kernel primitive to "mark" a process (in our case, we would mark Port Prolog processes) and then modifying the Port kernel to count the number of marked processes in the ready queue each time a context-switch occurs.

7.3 Comparisons with a Sequential Prolog Interpreter

Figure 7-3: Sequential Prolog vs. Port Prolog

(a) Program size

	Code Size (bytes)	Data Size (bytes)
Sequential Prolog		
solver	28553	10704
Port Prolog		
conjunction-process	12684	4144
goal-process	14412	4688
clause-process	16364	5296
support processes		
coordinator	12499	4000
vulture	4107	1760
io server	9528	6544
string server	8248	4672
parser	16282	5312
modules administrator	7619	3088
modules proprietor	14942	4896
errors worker	9135	2960

(b) Execution speed

	$benchmark_1$ (ms)	$benchmark_2$ (ms)
Sequential Prolog	59.38	143.20
Port Prolog	2306.14	5042.07

A multi-process structured, multi-window sequential Prolog interpreter has been developed on Port. This interpreter has a sequential run-time solver that uses the ABC Algorithm; other components were separated into processes to provide a more responsive and friendly user-interface. This sequential

interpreter actually shares the support processes used by Port Prolog, with slight modifications made to account for the differences in syntax and internal code. Figure 7-3 shows some measurements comparing the program sizes and execution times of the sequential Prolog interpreter and Port Prolog. Neither interpreters are tuned (i.e. implemented to optimize execution speed, minimize code and data sizes, etc).

7.3.1 Program Size

"Code Size" shown in Figure 7-3a includes both the program code and data. These values were obtained by using the Port *Fstats* (File Status) command. These numbers are on a per process basis. For example, the entries for "goal-process" are for one goal-process, and not for all the goal-processes that were used. Port supports "code sharing," which means that only one copy of the program code exist at any one time in memory; each process has its own data space. "Data Size" entries were obtained by using the Port *Pstats* (Program Status) command while the interpreters were evaluating the benchmark program in Figure 7-1. The data sizes depend on the size of the Prolog/Port Prolog programs being interpreted and the size of the database.

Examining the code and data sizes and the number of processes used, Port Prolog obviously expends much more memory than does sequential Prolog. This partially accounts for the slowness of Port Prolog because a major portion of time is spent searching for free memory. The fact that the Port operating system supports code sharing helps Port Prolog tremendously; otherwise, the memory usage would be much higher. Because many solver processes are used in solving a Port Prolog goal, some means to reduce the size of the solver processes should be investigated.

7.3.2 Execution Speed

Figure 7-3b contains the execution speeds for the two benchmark queries as they are evaluated by the two interpreters. The time used for parsing and input/output is not recorded.

The execution time of Port Prolog as compared with sequential Prolog is almost 50 times slower. If the operating systems overhead is ignored (i.e. only look at the "Others" field in Figure 7-2b), Port Prolog is still over 10 times slower. This remaining difference, after ignoring process management and communication overhead, is likely due to process synchronization overhead and overhead incurred because of multiple processes competing for one processor. Furthermore, if each Port Prolog solver process is viewed as a "stack frame" in a sequential interpreter, these stack frames are much larger and more complicated than those in a sequential interpreter, and hence require more time to initialize.

8

Conclusions

8.1 Summary

In this thesis, we investigated the use of processes to develop a logic programming system — *Port Prolog*.

Chapters 2 and 3 provided the background. First, we identified the different types of parallelism in logic programs. We then surveyed various logic programming systems, examining how each tried to incorporate the different types of parallelism. We examined the language features designed especially to control parallel execution of logic programs and the execution models employed to support these languages.

In Chapter 4, we adopted one of these models — an informal proposal for a distributed Concurrent Prolog machine — and formalized it as the computation model for Port Prolog. This model similarly uses three types of processes: *goal-processes, conjunction-processes, clause-processes*. We defined the duties of each type of process, how they synchronize and communicate via message-passing, the binding environment for variables, and how system predicates are accommodated.

Chapter 5 introduced the implementation environment — Waterloo Port — a multi-process, multi-window programming environment that supports processes and synchronous message-passing between processes. Several of Port's user interface ideas, process structuring techniques, and (language) source structuring techniques were used in the implementation of Port Prolog. The language features of Port Prolog programs and the user interface of the interpreter were described.

In Chapter 6, we described the implementation of Port Prolog. Port Prolog uses the model of computation defined in Chapter 4. The interpreter is a multi-process program which not only uses many processes as its run-time solver, but also many processes to manage its other components. There is a group of processes that manage the database of clauses; these processes are structured so that the database (and corresponding server processes) can be distributed over several machines. Other processes manage windows, the string table and parsing of the source. The solver component is a partial implementation of the model defined in Chapter 4. Its implementation status is discussed in the next section.

Finally, we took some measurements to evaluate the performance of Port Prolog. Among these data, we found that process management utilized approximately 46% of the time required to solve a goal while communication took about 13%. This information tells us where the bottlenecks of the system are. As one would expect, it indicates that we should tune the current model to reduce the communication and process costs. We also discovered from measurements made on some benchmark programs that Port Prolog is almost 50 times slower than a sequential Prolog interpreter running on Port.

8.2 Implementation Status

The processes described in Chapter 6 have been implemented. In addition, all the support processes have been integrated with a sequential Prolog solver process with minor changes to the parser and coordinator. The changes are due to the differences in syntax and the fact that sequential Prolog takes some parameters as maximum stack sizes.

The goal-process as described in the computation model in Chapter 4 has been fully implemented. However, the goal-process has yet to integrate the features of modules. In other words, it only queries one module proprietor for clauses.†

† To complete this feature for the goal-process, the component of the goal-process which communicates with the module proprietor needs to be modified. The following algorithm, which is

The conjunction-process as described in Chapter 4 has been partially implemented. It handles all but the *patch binding* incoming message. It handles *need binding* messages, but can only deal with one variable per message. The conjunction-process implemented does not send *need binding* messages to poll other processes for newly instantiated non-read-only variables.

The implementation of the clause-process lacks several features described in Chapter 4. It does not have a complete unification procedure. The cases that it does not handle yet are: free-free, reference-free, variable structure-free, variable structure-reference. It also does not handle incoming *need binding* and *patch binding* messages.

The system predicate process has not been implemented. This would require an interface to handle system predicates in the clause-process, but would not disturb the functionality of the clause-process or the rest of the interpreter.

To avoid the drawbacks of a centralized binding environment, Port Prolog supports a distributed binding environment (see Section 4.6 in Chapter 4). As in sequential Prolog, the unification of two free variables must resolve which variable is destinated the eventual "producer" of the binding. Although such a decision cannot be made accurately because it involves future knowledge, some heuristic must be applied so that when one of the two variables is bound, regardless of whether it was destinated the "producer," the other variable will also get the same binding. In sequential Prolog, a variable can refer to another variable by using a pointer. In Port Prolog, because variables may be located in different processes, and hence in separate address spaces, a "pointer" must indicate the process containing the variable as well as uniquely identify the variable within the process. This "pointer" is known as a *tag* in Port Prolog. The binding mechanism is further complicated by the fact that variables in Port Prolog are also used for synchronization (in the form of read-only variables).

Variable bindings are stored in conjunction-processes and clause-processes. Considering the problems noted above, the following information is kept for each variable (see Section 6.12.1 in Chapter 6).

- the type of the variable (e.g. read-only)
- the process id of the producer process
- the offset of the variable within the clause
- a list of processes waiting for the variable's binding
- the variable's binding

A process needing the binding of a variable first sends to the variable's producer process. The producer examines its binding environment; if the binding is available, it is returned to the sender. If the variable references another variable, the producer queries the process named in the variable's environment. This forwarding of queries is stopped either when a binding is found, or when the variable is determined to be free. If the variable is free and is a read-only variable, then the sender is placed in the waiting list of the variable. If the variable is free but normal, the sender is replied to with no binding. Bindings are copied from the producer whenever they are referenced so that future references need not go through the costly routine of extracting them from other processes.

In the implementation, the environments created by conjunction-processes and clause-processes have the contents listed above. Synchronization using a read-only variable is implemented by inserting the sender process into the waiting list of the read-only variable until the binding becomes available. Bindings are sent to the conjunction-process by a clause-process when the clause being solve is committed and when the clause has been successfully evaluated. Bindings are sent to the clause-process (and coordinator)

---

used by the sequential Prolog interpreter on Port, could be applied. A goal-process first queries a module proprietor (made known to the goal-process through its parent) for the predicate. If the predicate is found, then the goal-process proceeds as usual; otherwise, the goal-process sends a look-up request to the modules administrator to find which module proprietor exports the predicate. If the predicate is not being exported by any proprietor, the search fails; otherwise, the modules administrator replies to the goal-process with the process identifier of the appropriate module proprietor. The goal-process then queries this proprietor for the predicate. This proprietor is used (by the children of the goal-process) until a predicate which is not handled by the proprietor is encountered, at which time the above algorithm is applied.

by the conjunction-process when all the literals being managed by the conjunction-process have been successfully evaluated. Currently, only goal-processes query other processes (conjunction-processes) for bindings. The "forwarding" of queries from one process to another has not been implemented.

The implementation demonstrates that variable bindings can be passed between processes, and hence shared variables are possible in our model. Synchronization between dependent literals in a clause can be achieved if the variable involved does not contain other free variables. Section 4.6 in Chapter 4 proposed an heuristic for electing a producer process for the unification of free-free variables. By incorporating this into the unification procedure of the clause-process, we can determine the success of the heuristic by measuring the number of "hits." Also, different strategies can be tried and evaluated against the proposed scheme. By completing the unification procedure and properly forwarding queries from one process to another, we can test whether deadlocks or other complications arise from the binding retrieval algorithm described above. Also, with fully implemented retrieval and unification algorithms, we can measure the communication overhead on a more realistic set of concurrent Prolog programs and observe the cost of shared variables.

The following provides some examples of the features of Concurrent Prolog supported by Port Prolog.

Invocation of *choose(A)* given the following clause definitions:

```
choose( dummy )
        pred1( A? )
        pred2( A );

pred1( bad_choice1 );
pred1( bad_choice2 );
pred1( A );

pred2( choice1 );
pred2( choice2 );
pred2( choice3 );
```

results in a "yes" answer with *A* instantiated to *dummy*. This demonstrates variable synchronization, albeit with one level of variable indirection and with only one variable. The predicate *pred1* does not get to execute until *A?* is instantiated by *pred2*. When *pred1* does get to execute, although they are tried simultaneously (OR-parallelism) the first two *pred1* clauses fail and the third succeeds. This program also shows how a variable binding is passed from a clause-process (the one that solved *pred2*) to its grandparent conjunction-process (which is taking care of *choose(dummy)*), then to a goal-process (solving *pred1*), which passes it onto another clause-process (to solve *pred1(A)*).

The *stack* program given in Section 7.1 of Chapter 7 is another example of a Port Prolog program. Because Port Prolog does not have a full unification procedure, the replacement of any of the arguments of the benchmark queries by variables causes the queries to fail. For example, both *stack([push(1),pop(A)])* and *stack([push(1),A])* fail.

8.3 Experience using the Port Process Abstraction

The notion of independent processes communicating only via synchronous message-passing is a powerful tool for designing systems, especially in systems in which asynchrony and parallel execution have to be controlled. Port's interprocess communication (IPC) primitives and its process abstraction greatly influenced the design of our computation model and were found to be sufficient for our model. The independence of a Port process (i.e. no shared memory) would also lessen the work of re-designing a functionally identical yet multi-processor based implementation. However, the syntactic difference between local and remote IPC primitives may be a hindrance.

The one-to-one IPC primitives provided by Port is adequate, though sometimes limiting. For example, the conjunction-process polls other processes to obtain variable bindings. This is inefficient and, depending on the polling frequency, may cause the system to thrash (doing context-switches). Furthermore, the binding mechanism is contorted to depend on the one-to-one IPC primitives. If the number of indirections needed to find the producer of a variable is large, many message passes through several

processes would be needed before the binding is finally retrieved. Shared memory would, of course, make the binding mechanism easier to define (cf. sequential Prolog), but would defeat the purpose of a distributed model. Something similar to the notion of "sharing" knowledge is the multicast (one-to-many) IPC proposed by Cheriton and Zwaenepoel [Cheriton & Zwaenepoel 84]. This may be *the key* to a simple to conceptualize and easy to implement binding environment.

Multicast IPC allows one process to simultaneously send to a group of processes; a group is identified by a *group id*. The sender is suspended until k replies are received; k is defined as 0, 1 or some integer. Processes wishing to start a group use an *allocate_group_id* primitive and those wishing to join a group use a *join_group* primitive.

With this multicast IPC facility, the binding of variables could use the notion of a group to identify the processes that share a variable. A process instantiating a variable could do a multicast send to the group interested in this variable. In other words, the tag† would contain the group id instead of a process id. A clause-process unifying a variable with another free variable would either form a group for the variable with the owner of the free variable (if one does not already exist) or join the group associated with the variable. The unification of two variables that each already belong to its own group (i.e. a reference-reference instantiation) could be solved by introducing a new message to the solver processes. This new message would ask members of a group to "change" to a different group; this is analogous to changing the pointer of one of the variables to point to the other variable in a reference-reference instantiation in sequential Prolog. This group concept would make the producer's identity irrelevant (and thus we would not have to worry about "incorrect" tags); all members of a group are treated equally.

The use of multicast IPC for handling the binding of variables seems promising. The reliability and efficiency of multicast IPC would ultimately determine its feasibility.

Multicast IPC could also be used to speed up the searching of the database. We could eliminate the modules administrator by introducing a new request to the module proprietor to handle searches for exported predicates. In this case, the goal-process would issue a multicast "export search" send to the proprietors, instead of asking the modules administrator to identify the correct proprietor.

The Port process management primitives, though costly, were also adequate. For a distributed implementation, creation of processes on a remote machine could be done by having a special agent process on each machine to handle remote creates. This special process could be made known to processes on other machines through the Port *name server*.

If the Port kernel had more knowledge about the relationships between processes, the vulture process would not be necessary. For example, the kernel could provide a new destroy primitive that can destroy a "family" of processes. However, this new primitive may be difficult to realize in a distributed implementation. It would also imply the need for *real* remote create and remote destroy primitives, instead of accomplishing such tasks through agent processes.

8.4 Future Work

- **Completion of Interpreter.** Before enhancements can be effectively attempted, the conjunction and clause process abstractions of the proposed model must be completed and debugged. Better variable representation (both in terms of space and access method) within a process (the conjunction or clause-process) is also desirable. To be generally useful, the interpreter requires the implementation of the system predicates process(es) as described in Chapter 4, together with a sufficiently rich set of system predicates. This would include meta level predicates, database manipulation predicates, list and functor manipulation predicates and a debugging package.

- **Performance Analysis.** The performance measurements presented in the previous chapter are minimal and are based on only a few benchmarks. Our measurements on a more extensive set of examples, followed by a detailed analysis of these numbers, would be helpful in examining the feasibility of the computation model. These measurements should provide information such as the ratio of communication to computation performed by each process, frequency distribution of message passes (e.g. whether most

† See Section 4.6 in Chapter 4.

messages are exchanged during initialization or throughout evaluation), types of messages passed (mostly control? or mostly variable bindings?), a record of the activity during the life-time of a process (e.g. whether it is suspended most of the time), the processor utilization, and the number of processes competing for the processor.

An analysis of this information can help to define a better model to accommodate the shortcomings of this model. Similar analyses can be applied to similar computation models and their performance with respect to this model evaluated.

- **Process Management.** The evaluation of a list of goals using conjunction-processes, goal-processes, and clause-processes with the model described in Chapter 4 requires a substantial amount of dynamic process management. These processes are frequently created and destroyed; cost is incurred every time a process is created (time is needed to find resources for creation), and likewise when a process is destroyed (time is required for recovering resources). To reduce the overhead of process management, the interpreter can keep a pool of processes and re-cycle them as needed [Bowen 82, Kasif et al 83, Birrell & Nelson 83]. Instead of destroying a process when it is no longer needed, the process is kept in a "pool," waiting to be re-used. When more processes are needed than can be supplied by the pool, the Port create primitive can then be used to create new processes. When the number of processes in the pool gets large and the processes are not being used, they can be freed by using the Port destroy primitive. Since processes are re-cycled, fewer searches and recoveries of resources by the Port operating system is necessary. The cost of process management is hence reduced.

- **Process Structures.** The process structure of Port Prolog's supporting components is but a first attempt. Other structures which can shorten the initialization time or improve the responsiveness of the system should be investigated. For example, compilation of the source may be sped up by providing each module proprietor with its own parser instead of having several proprietors queue up for one.

- **Swapping.** In our performance measurements, no account was kept of the activity of a process. We do not know, for example, how long a conjunction-process spends suspended waiting to serve requests. But intuitively, the solver processes spend most of their time suspended and exchange messages mostly among its parent and children processes. Swapping — that is, moving processes out of main memory when they are suspended — would be useful in this situation. This would allow more processes to co-exist in the system at any one time; however, context switching might become more expensive as the operating system may have to load the processes from external devices.

- **Tail-recursion Optimization.** Another enhancement which can decrease memory usage is the reduction of the number of processes being used. Tail-recursion optimization strategies in which processes, rather than stack space, are being recovered and re-used are needed if Port Prolog is to be a usable system. Such strategies will draw experience from how tail-recursion optimization for stacks in sequential Prolog is done.

- **Multi-processors.** Because the interpreter is designed using multiple processes, and since Port supports inter-process communication of processes across machine boundaries, we may attempt multi-processor experiments that distribute components of the interpreter over several machines. Such experiments will be facilitated by the fact that a Port process is independent and has its own address space. These can include distributing both the database processes and the solver processes. Simulations on fixed network configurations can give insight into an "optimal" configuration for executing concurrent Prolog. More importantly, these experiments will give a calibration of the potential of concurrent Prolog when the environment is truly parallel.

- **Prolog Kernel.** Services that are often use by many processes can be "factored" and put into a specially designed kernel. For example, every clause-process must do unification. By making unify a special kernel request, a substantial amount of memory can be saved. Also, by isolating unify, it may be easier in the future to speed it up by, say, coding it in microcode. In addition to the customary support for message-passing, this kernel can be designed to dispatch and queue solver processes, perhaps managing the processes in methods similar to those suggested earlier.

- **Integration with Sequential Prolog.** Certain aspects of an application may require sequential evaluation while other aspects may benefit from concurrent execution. With the availability of both sequential Prolog and concurrent Prolog under the same operating system, the potential of integrating

Port Prolog with sequential Prolog should be investigated.

• **Development Tools.** Finally, to make Port Prolog pleasant to use, the interpreter needs tools to help the logic programmer develop his programs. Some predicates, such as the debugging predicates and some database manipulation predicates, should be implemented with the Port environment in mind. They should not mimic corresponding predicates in conventional Prolog interpreters that only make use of teletype terminals. By making use of the multi-windowing facilities of Port, predicates that list the database, edit a set of clauses in the database, add and delete clauses in a session (not dynamically during execution), and aid debugging the execution of a goal can be done in separate windows.

Bibliography

[Birrell & Nelson 83] A. D. Birrell, and B. J. Nelson, *Implementing Remote Procedure Calls*, Xerox PARC Research Report CSL-83-7 (Dec 1983). Also in *ACM Transactions on Computer Systems*, Vol. 2, No. 1 (Feb 1984).

[Bonkowski et al. 84] B. Bonkowski, P. Didur, M. A. Malcolm, G. Stafford, and T. Young, *Programming in Waterloo Port,* Software Portability Group, University of Waterloo, Waterloo, Ontario (Mar 1984).

[Bowen 82] K. A. Bowen, "Concurrent Execution of Logic," *Proceedings of the First International Logic Programming Conference,* Marseille, (Sept 1982) pp 26-30.

[Cheriton & Zwaenepoel 84] D. R. Cheriton, and W. Zwaenepoel, "One-to-Many Interprocess Communication in the V-System," *Proceedings of the ACM SigComm Symposium on Communication Architectures & Protocols*, Pre-print copy (June 1984). Also to appear in *ACM Transactions on Computer Systems*, ACM Press.

[Cheriton 82] D. R. Cheriton, *The Thoth System: Multi-process Structuring and Portability,* American Elsevier (1982).

[Cheriton et al. 1979] D.R. Cheriton, M.A. Malcolm, L.S. Melen, and G.R. Sager, "Thoth, A Portable Real-Time Operating System," *Communications of the ACM,* Vol. 22, No. 2 (Feb. 1979), pp. 105-115.

[Ciepielewski & Haridi 83] A. Ciepielewski, and S. Haridi, "Control of Activities in the OR-Parallel Token Machine," *Proceedings of Logic Programming Workshop,* Portugal (1983) pp 536.

[Clark & Gregory 81] K. L. Clark, and S. Gregory, *A Relational Language for Parallel Programming,* Imperial College Research Report, DOC 81/16 (July 1981). Also in *Proceedings of ACM Conference on Functional Programming Languages and Computer Architecture,* ACM Press (Oct 1981).

[Clark & McCabe 79] K. L. Clark, and F. G. McCabe, *IC-Prolog Reference Manual,* Imperial College, London CCD-Publication No.79/7 (July 1979).

[Clark & McCabe 79b] K. L. Clark, and F. G. McCabe, "The Control Facilities of IC-Prolog," *Expert Systems in the Micro-electronic Age,* edited by D. Michie, Edinburgh University Press (1979) pp 122-149.

[Clark et al. 82] K. L. Clark, F. G. McCabe, and S. Gregory, "IC-Prolog Language Features," *Logic Programming,* edited by K. L. Clark and S-A. Tarnlund, Academic Press, (1982) pp 253-266.

[Clocksin & Mellish 81] W. F. Clocksin, and C. Mellish, *Programming in Prolog,* Springer Verlag (1981).

[Conery & Kibler 81] J. S. Conery, and D. F. Kibler, "Parallel Interpretation of Logic Programs," *Proceedings of ACM Conference on Functional Programming Languages and Computer*

Architecture, ACM Press (Oct 1981) pp 163-170.

[Darlington & Reeve 81] J. Darlington, and M. Reeve, "ALICE: a Multi-processor Reduction Machine for the Parallel Evaluation of Applicative Languages," *Proceedings of ACM Conference on Functional Programming Languages and Computer Architecture,* ACM Press, (1981) pp 65-75.

[Date 77] C. J. Date, *An Introduction to Database Systems,* Second Edition, Addison-Wesley Publishing Co., Reading, MA, pp 113-120.

[DeGroot 83] D. DeGroot, *Logic Programming and Parallel Processing,* slides from course on Prolog's potential for parallel computation at Brandeis University (1983).

[Didur et al. 84] P. A. Didur, M. A. Malcolm, and P. A. McWeeny, *Waterloo Port User's Guide,* Software Portability Group, University of Waterloo, Waterloo, Ontario (Apr 1984).

[Eisinger et al. 82] N. Eisinger, S. Kasif, and J. Minker, "Logic Programming: A Parallel Approach," *Proceedings of the First International Logic Programming Conference,* Marseille, (Sept 1982) pp 71-77.

[Haridi & Ciepielewski 83] S. Haridi, and A. Ciepielewski, "An OR-parallel Token Machine," *Proceedings of Logic Programming Workshop,* Portugal (1983) pp 537-552.

[Hoare 78] C. A. R. Hoare, "Communicating Sequential Processes," *Communications of the ACM,* Vol. 21, No. 8, (Aug 1978) pp 666-677.

[Hogger 82] C. J. Hogger, "Concurrent Logic Programming," *Logic Programming,* edited by K. L. Clark and S-A. Tarnlund, Academic Press, (1982) pp 199-211.

[Jurchuk 78] R. W. Jurchuk, *The Four Channel Event Counter/Timer (Model JR-78),* Work Report, Department of Computer Science, University of Waterloo, Waterloo, Ontario (Jan 1979).

[Kahn 82] K. Kahn, "Intermissions — ACTORs in Prolog," *Logic Programming,* edited by K. L. Clark and S-A. Tarnlund, Academic Press, (1982) pp 213-228.

[Kasif et al. 83] S. Kasif, M. Kohli, and J. Minker, "PRISM: A Parallel Inference System for Problem Solving," *Proceedings of Logic Programming Workshop,* Portugal (1983) pp 123-145.

[Kowalski 82] R. Kowalski, "Logic as a Computer Language," *Logic Programming,* edited by K. L. Clark, and S-A. Tarnlund, Academic Press, (1982) pp 3-17.

[Nilsson 82] N. J. Nilsson, *Principles of Artificial Intelligence,* Tioga Publishing Co., Palo Alto, CA. (1982).

[Onai et al. 82] R. Onai, H. Shimizu, N. Ito, and K. Masuda, *The Proposal of a Prolog Machine based on Reduction Mechanism,* First Research Laboratory Technical Report, Research Center, ICOT (1982).

[Rapsey 79] L. Rapsey, *Some Performance Measurements of the Thoth Operating System,* MMath dissertation, Department of Computer Science, University of Waterloo, Waterloo, Ontario (Nov 1978).

[Shapiro 83] E. Y. Shapiro, *A Subset of Concurrent Prolog and Its Interpreter,* ICOT Technical Report

TR-003 (Jan 1983).

[Shapiro & Takeuchi 83] E. Y. Shapiro, and A. Takeuchi, "Object Oriented Programming in Concurrent Prolog," *New Generation Computing,* Vol. 1, No. 1, (1983) pp 25-48.

[Takeuchi & Furukawa 83] A. Takeuchi, and K. Furukawa, "Interprocess Communication in Concurrent Prolog," *Proceedings of Logic Programming Workshop,* Portugal (1983) pp 171-185.

[Treleaven et al. 82] P. C. Treleaven, D. R. Brownbridge, and R. P. Hopkins, "Data-Driven and Demand-Driven Computer Architecture," *ACM Computing Surveys,* Vol. 14, No. 1, (March 1982) pp 93-143.

[Umeyama & Tamura 83] S. Umeyama, and K. Tamura, "A Parallel Execution Model of Logic Programs," *Proceedings of the Tenth Annual International Symposium on Computer Architecture,* Stockholm, (1983) pp 349-354.

[van Emden 82] M. H. van Emden, "An Interpreting Algorithm for Prolog Programs," *Proceedings of the First International Logic Programming Conference,* Marseille, (Sept 1982) pp 56-64bis.

[van Emden & de Lucena 82] M. H. van Emden, and G. J. de Lucena, "Predicate Logic as a Language for Parallel Programming," *Logic Programming,* edited by K. L. Clark and S-A. Tarnlund, Academic Press (1982) pp 189-198.

[Vasudevan 84] R. Vasudevan, *Performance Measurements on the Port Kernel*, Internal Report, Software Portability Laboratory, University of Waterloo, Waterloo, Ontario [in preparation].

[Warren 79] D. Warren, "Prolog on the DECsystem-10," *Expert Systems in the Micro-electronic Age,* edited by D. Michie, Edinburgh University Press (1979) pp 117-121.

[Wise 82] M. J. Wise, "A Parallel Prolog: the Construction of a Data Driven Model," *Proceedings of ACM Symposium on LISP and Functional Programming,* ACM Press, New York, (Aug. 1982) pp 56-66.

Appendix A: Counter/Timer Measurements

1. Breakdown of Execution Time for Port Prolog

2. Create/Destroy Times for Some Common Port Operations

3. Execution Times for Sequential Prolog

Breakdown of Execution Time for Port Prolog

	Total (ms)	Create (ms)	Destroy (ms)	Setup (ms)	Number of Sends
$benchmark_1$	2316.17	127.10	768.18	174.42	118
	2320.33	121.70	769.78	174.42	116
	2298.77	126.61	764.86	174.45	117
	2316.17	123.10	779.39	174.45	117
	2306.83	121.94	764.04	174.44	118
	2289.86	126.55	776.40	174.42	118
	2320.27	120.62	785.33	174.43	118
	2292.60	127.91	776.28	174.45	117
	2303.81	128.89	775.31	174.43	117
	2296.60	121.55	746.06	174.43	117
average	2306.14	124.60	770.56	174.43	117
			+ 50.58		x 2.81
			821.14		328.77
$benchmark_2$	5059.50	280.55	1643.22	368.16	251
	5042.50	281.21	1645.84	368.17	249
	5037.10	272.60	1636.37	368.12	250
	5045.53	263.03	1662.10	368.15	249
	5048.31	271.30	1629.38	368.20	249
	5047.93	269.00	1635.97	368.16	249
	5038.87	254.59	1640.56	368.16	251
	5032.17	269.04	1638.44	368.20	250
	5038.90	289.79	1658.73	368.18	249
	5029.93	254.74	1637.92	368.14	249
average	5042.07	270.59	1642.85	368.16	250
			+ 106.78		x 2.81
			1749.63		702.50

number of processes created/destroyed in $benchmark_1$ = 18
number of processes created/destroyed in $benchmark_2$ = 38

average time for a send = 2.81ms [Vasudevan 84]

Destroy cost = destroy average + (number of destroys x 2.81)

Total communication overhead = number of sends x 2.81

Total process management overhead = Destroy + Create + Setup

Create/Destroy Times for some Common Port Operations

The Port Process Administrator was modified to measure the time required to handle CREATE and DESTROY requests. The following are some measurements. Some creates require loading the code off the hardisk.

Task	Create		Destroy	
	Number	Time (ms)	Number	Time (ms)
Initialization	71	724.11	37	2944.30
(i.e. boot)			1	287.05
Enter Office from Lobby	23	226.70	23	1130.96
Enter Prolog room from Office	8	82.61	8	339.74
Start Port Prolog Interpreter	19	219.97	9	952.76
Tried $benchmark_1$ in Interpreter	18	127.10	18	768.18
		121.70		769.78
		126.61		764.86
		123.10		779.39
		121.94		764.04
		126.55		776.40
		120.62		785.33
		127.91		776.28
		128.89		775.31
		121.55		746.06
Tried $benchmark_2$ in Interpreter	38	280.55	38	1643.22
		281.21		1645.84
		272.60		1636.37
		263.03		1662.10
		271.30		1629.38
		269.00		1635.97
		254.59		1640.56
		269.04		1638.44
		289.79		1658.73
		254.74		1637.92
Kill interpreter			10	3279.07
Browser on Hardisk	8	109.47	5	375.51
Climb on Browser	1	12.07	1	28.55
Exit Prolog room	3	45.13	3	125.42

Execution Times for Sequential Prolog

Benchmark Program:

```
stack( S )
   stack( S  [] );

stack( [pop(X)|S]  [X|Xs] )
   stack( S  Xs );

stack( [push(X)|S]  Xs )
   stack( S  [X|Xs] );

stack( []  [] );
```

Benchmark Goals:

benchmark$_1$: stack([push(1),pop(1)]);

benchmark$_2$: stack([push(1),push(2),push(3),pop(3),pop(2),pop(1)]);

	benchmark$_1$ (ms)	*benchmark*$_2$ (ms)
Execution Times	62.86	145.74
	63.04	148.67
	57.00	144.46
	62.95	138.44
	62.95	144.37
	56.99	144.39
	56.99	144.47
	57.00	144.37
	57.01	138.60
	56.99	138.52
average	59.38	143.20

Appendix B: Program Traces

1. Program Trace of a Clause-process

2. Program Trace of a Goal-process

3. Program Trace of a Conjunction-process

The conjunction-process, goal-process and clause-process were traced with the following benchmark program and goal as input. The trace of the conjunction-process is that of the first conjunction-process created (by the coordinator). The trace of the goal-process is that of the first goal-process created by this conjunction-process. The trace of the clause-process is that of the second clause-process created by the same goal-process.

Benchmark Program

```
stack( S )
   stack( S? [] );

stack( [pop(X)|S] [X|Xs] )
   stack( S? Xs );

stack( [push(X)|S] Xs )
   stack( S? [X|Xs] );

stack( [] [] );
```

Benchmark Goal:

```
stack( [push(1),pop(1)] );
```

Trace of a Clause-process

Number of Instructions Executed	Function Name
7	_Fcn_prolog
7	_Fcn_epilog
27	_Kernel_entry
2	Convert_to_pointer
7	Find_top_of_memory
996	Zero
150	Prolog
42	Epilog
16	Clause_process
28	_Send
90	Dereference
84	Get_list_info
30	_Transfer_from
168	Get_functor_info
75	Get_procedure_info
10	_Reply
12	_Receive
9	_Get_registered_id
23	Size_of_term
10	Unsuccessful_termination
242	Unify
94	Unify_functor
43	Unify_list
28	Initialize
10	Initialize_apply_stack
389	_Alloc
22	_Grow_data_segment
10	U_max
9	Create_literal_environment
27	Look_at_term
96	Record_new_variable
324	Alloc
90	Find_a_vector
55	Create_clause_environment
75	Get_literal_and_clause
8	Initialize_vulture
21	Set_up_window
15	Select_output
15	Select_input
1	Display_message
96	_Allocate_frame
3,463	Total

Trace of a Goal-process

Number of Instructions Executed	Function Name
1	_Fcn_prolog
1	_Fcn_epilog
60	_Kernel_entry
63	Length
2	Convert_to_pointer
660	Zero
7	Find_top_of_memory
23	Prolog
6	Epilog
34	Goal_process
30	_Reply
41	Commit
12	Destroy_process
112	_Send
30	_Transfer_from
42	Get_functor_info
12	_Receive_any
36	Initialize
29	Create_clause_processes
51	Free
35	Create_a_clause_process
15	Initialize_global
26	_Receive
27	Create_process
27	Create
18	_Get_registered_id
52	Get_call_info
31	Get_all_clauses
17	Find_next_clause
72	Alloc
20	Find_a_vector
187	_Alloc
22	_Grow_data_segment
10	U_max
42	Find_first_clause
6	Free_environment
17	Wait_for_input_variables
18	Apply_substitutions
23	Size_of_term
9	Create_literal_environment
27	Look_at_term
10	Initialize_apply_stack
67	Get_literal
8	Initialize_vulture
21	Set_up_window
15	Select_output
15	Select_input
1	Display_message
16	_Allocate_frame
2,106	Total

Trace of a Conjunction-process

Number of Instructions Executed	Function Name
1	_Fcn_prolog
1	_Fcn_epilog
48	_Kernel_entry
61	Length
115	Zero
7	Find_top_of_memory
2	Convert_to_pointer
23	Prolog
6	Epilog
34	Conjunction_process
30	_Reply
70	_Send
15	Success
10	Successful_termination
24	Find_literal_of
24	Commit
24	_Receive
72	Alloc
109	_Alloc
18	_Get_registered_id
11	_Grow_data_segment
5	U_max
20	Find_a_vector
15	_Transfer_from
24	_Receive_any
26	Initialize
38	Create_goal_processes
84	Get_call_info
45	Create_a_goal_process
15	Initialize_global
27	Create_process
27	Create
31	Create_sequence_environment
27	Look_at_term
23	Size_of_term
35	Get_literal_list
8	Initialize_vulture
21	Set_up_window
15	Select_output
15	Select_input
1	Display_message
16	_Allocate_frame
1,223	Total

Trace of a Conjunction-process

Number of Instructions Executed	Function Name
1	_Fcn_prolog
1	_Fcn_epilog
48	_Kernel_entry
61	Length
115	Zero
7	Find_top_of_memory
2	Convert_to_pointer
23	Prolog
6	Epilog
34	Conjunction_process
30	_Reply
70	_Send
15	Success
10	Successful_termination
24	Find_literal_of
24	Commit
24	_Receive
72	Alloc
109	_Alloc
18	_Get_registered_id
11	_Grow_data_segment
5	U_max
20	Find_a_vector
15	_Transfer_from
24	_Receive_any
26	Initialize
38	Create_goal_processes
84	Get_call_info
45	Create_a_goal_process
15	Initialize_global
27	Create_process
27	Create
31	Create_sequence_environment
27	Look_at_term
23	Size_of_term
35	Get_literal_list
8	Initialize_vulture
21	Set_up_window
15	Select_output
15	Select_input
1	Display_message
16	_Allocate_frame
1,223	Total