CONCURRENT PROLOG IN A MULTI-PROCESS ENVIRONMENT

Rosanna K.S. Lee
Randy Goebel

Department of Computer Science
University of Waterloo
WATERLOO, Ontario, Canada
N2L 3G1

Research Report CS-85-09
May 1985

Concurrent Prolog in a Multi-process Environment†

*Rosanna K. S. Lee**
Randy Goebel

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 simple yet powerful transformation of Prolog that incorporates concurrency, communication, synchronization and indeterminacy. Here we report on the development of a computation model for Concurrent Prolog which uses processes communicating via message-passing. A prototype implementation, *Port Prolog,* has been programmed in *Waterloo Port,*** — a multi-process programming environment that supports concurrent activities. The Port Prolog computation model defines the process structure, communication paths, binding environment, and synchronization procedures for Concurrent Prolog. The interpreter is a partial implementation of the model. A summary includes some performance measurements made with the prototype.

1. Motivation and objectives

A most appealing characteristic of logic programs is their inherent admissibility of parallel execution. Substantial effort has been expended exploring this potential. One such effort is Shapiro's definition of *Concurrent Prolog* — a simple yet powerful transformation of Prolog that incorporates "concurrency, communication, synchronization and indeterminacy" [10, p. 9].

Here we examine the issues surrounding the design and implementation of a concurrent Prolog interpreter, *Port Prolog,* in a message-based multi-process environment. Our overall objective is to determine the feasibility of employing message-based processes to build a Concurrent Prolog interpreter. By exploiting an environment that supports multiple processes, concurrency and networking, we seek to develop the understanding needed to implement a truly parallel interpreter. A related objective is to fully define and validate Shapiro's informal description of a distributed Concurrent Prolog machine. The design of Port Prolog is based on this description, rather than Shapiro's sequential implementation [10].

The major implementation problem is the management of the binding environment of shared variables for AND-parallelism. OR-parallelism is more straight-forward to implement, but practical use requires special operations for its control. We do not explore the potential advantages of incorporating stream and search parallelism into the model [4], even though the current implementation is structured so as to distribute I/O and database retrieval responsibilities to different processes.

†This paper will appear in the Proceedings of the IEEE 1985 Symposium on Logic Programming, Boston, Massachusetts, July 15-18.
*Now at SRI International, 333 Ravenswood Avenue, Menlo Park, California, USA 94025.
**Port and Waterloo Port are trademarks of the University of Waterloo.

2. Concurrent Prolog

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 [3] except that the G_i's need not be executed sequentially. In fact, all atoms 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 are no mode declarations. 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 the latter has only one annotation for its variables: the *read-only* "?" symbol, as compared with the Relational Language's input "?" and output "^" annotations. Evaluation of an atom containing *read-only* variables is suspended until the variables are instantiated. (More specifically, unification involving a read-only variable suspends until the variable is bound — a deviation from the definition by Shapiro [10]. Tam [11] gives some convincing arguments supporting this modification.)

3. The Port process model

It is useful to describe parallel systems in terms of *processes*. A *process* is the execution of a self-contained code segment and its associated data structures. It shares no memory with other processes, regardless of their kinship. The abstraction we are interested in is realized in a message-based system in which data is shared between processes only via *message-passing* [1,2]. Each processes is identified and addressed by a unique *process identifier*, or *id* for short.

Process management is performed dynamically with two primitives: *create* and *destroy*. Any process can *create* a new process: the creator is called the *parent* process and the newly created process is called the *child* process. 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.

3.1. Communication primitives

All process communication is done with three basic message-passing primitives: *send*, *receive*, and *reply*:

```
send( process_id, request, reply_msg )
receive( process_id, request )
process_id = receive_any( request )
reply( process_id, reply_msg )
```

These primitives provide synchronization between communicating processes. A process that issues a *send* is suspended until the destination process, identified by *process_id*, receives the *request* and replies with *reply_msg*. The destination process may execute any number of instructions before it replies.

There are two types of *receive*: specific and general. A process that issues a specific *receive* is blocked until the process identified by *process_id* decides to acknowledge with a *send*. After the acknowledging *send*, the receiver is free to continue execution. When the receiver subsequently does a *reply*, the awaiting sender process (specified by *process_id*) is unblocked. The semantics for the general receive, *receive_any*, is similar to the specific *receive* except that the process doing a *receive_any* is suspended until some process sends to it. This primitive returns the id of the sending process so that the receiver knows where to reply.

An application's *process structure* refers to an application's processes and their interaction. Many issues are involved in the design of an application's process structure; some considerations are process load and functionality, concurrency, and memory usage. Communication costs and context switching are also considerations that must be addressed. The synchronous nature of the message-passing primitives requires context switching between communicating processes. Because message-

passing is significantly more time consuming 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. Port Prolog process structure

Port Prolog's execution model is based on Shapiro's informal description of a distributed Concurrent Prolog machine [10]. His model uses three types of process: *conjunction-process, goal-process*, and *clause-process*. The responsibilities of the corresponding Port Prolog processes are similar to those defined by Shapiro [10], 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 also described in fig. 1.

A conjunction process is created for the task of solving a conjunction of atoms $A_1, A_2, \cdots A_m$. The conjunction-process creates a goal-process for each A_i, $(1 \le i \le m)$. Each goal-process is responsible for finding a solution to its atom A_i; it does so by creating a clause-process for each clause with the same predicate name and arity as A_i. The clause-processes then attempt to unify A_i with their clause heads. The successful clause-processes proceed to solve the guard sequence of their clause by spawning a conjunction-process to evaluate the atoms of their guard sequence. The conjunction-process solves the sequence by the method now being described — that is, by spawning goal-processes for each atom in the sequence. If the guard conjunction process succeeds, it notifies its parent clause-process. The first clause-process that informs its parent goal-process of success becomes the *candidate* clause-process and all other clause-processes are destroyed by their parent goal-process. The goal-process then destroys itself. The candidate clause-process proceeds to solve the goal sequence of its clause by spawning another conjunction-process. 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 A_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 a clause with the same predicate name and arity as its goal A_i. A clause-process fails if it cannot unify its A_i with the given clause or if its child conjunction-process fails. Upon failure, a process notifies its parent process and then destroys itself.

5. Binding of Variables

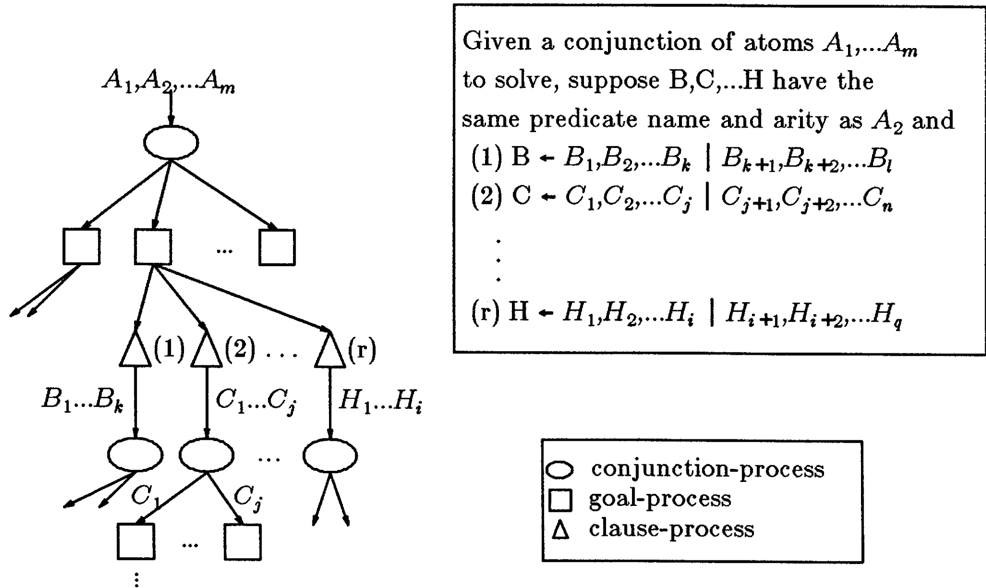
The binding of variables in Port Prolog is done distributively — there is no central data structure that maintains all variable bindings (cf. the environment stack of sequential Prolog). This decentralized approach makes the system less dependent upon tightly-coupled processor architectures, and less prone to communication bottlenecks and consistency problems due to different processes instantiating the same variables simultaneously. However, there are 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 — a reference must identify the process that has the bindings as well as provide directions as to how to recover them. Therefore, Port Prolog will copy bindings whenever they are first referenced, so that future references by the same process need not repeat the task.

Another disadvantage of the distributed binding environment is the difficulty of creating and dereferencing the bindings. Which process does a process query when it requires variable bindings? The answer 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

†In Port, a null function call requires approximately 0.02 ms whereas the minimum cost of a send/receive/reply cycle is 2.54ms [12].

(a) Before ANY *commits*



(b) After ALL A_i's have been *commit*ted

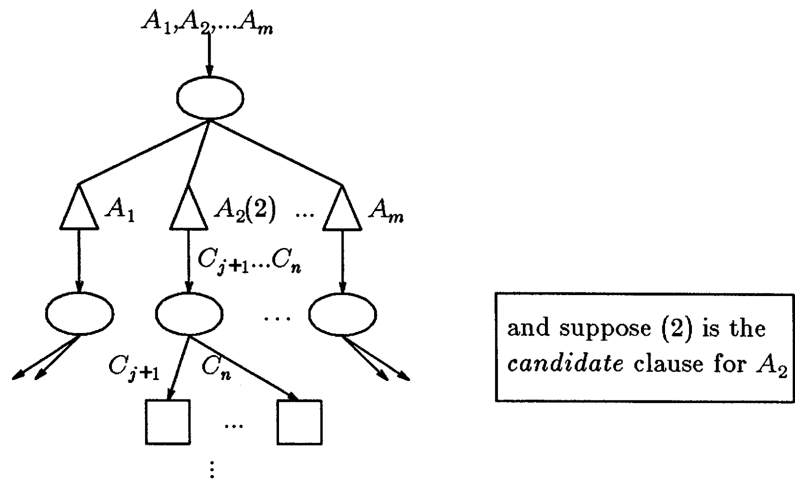


Figure 1. Port Prolog process structure

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 is based on: (1) creation and distribution of a *tag* for unbound variables, from which processes can derive information on how to eventually obtain bindings; (2) classification of unbound variables; and (3) an algorithm for retrieving bindings for unbound variables.

5.1. Tags

As in sequential Prolog, the unification of two free variables must determine which variable is designated 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 designated the "producer"), the other variable will also get the same binding. In sequential Prolog, one variable can refer to another by using a pointer. In Port Prolog, because variables may be located in different processes, and hence in separate address spaces, a "pointer" must identify the process containing the variable as well as uniquely identify the variable within that process. In Port Prolog, this "pointer" is known as a *tag*.

Tags provide a direct way for accessing the variable's binding. Without this information, a process must traverse the accessible process structure until the variable's binding is found. 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.

Variables are classified into four types: REFERENCE, FREE, READ_ONLY_REFERENCE, and READ_ONLY_FREE. There are essentially two groups: read-only and normal variables, with further distinction between their "state of binding" (FREE or REFERENCE). FREE means that the tag associated with the variable has not been set (i.e. the tag is empty). Processes that require values for FREE variables must ask their immediate ancestor. REFERENCE variables are those that contain tags indicating which process is likely to have values for the variables.

5.2. Storage of Bindings

Variable bindings are maintained in both conjunction-processes and clause-processes. Each process builds environment tables to store the bindings and information relevant to identifying variables in other processes.

Variable bindings from a sequence of subgoals are stored in the conjunction-process that is solving the sequence. This enables descendant processes to access (via message-passing) variables shared among atoms in the sequence. For example, for the sequence

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

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 record the bindings of variables in the clause and those in the goal atom. These environments are also used to map variables in the clause to those in the atom. For example, suppose a clause-process is given

atom: *father(Child, Father)*

clause: *father(C, F) ← 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.

5.3. Retrieval and setting of bindings

There are two messages for the retrieval and setting of bindings: *get binding* and *set binding*. Any process may initiate retrievals of bindings, except that binding retrieval for read-only variables is initiated by goal-processes. Conjunction-processes poll their parent clause-processes to update their variables so that unannotated variables may be instantiated (for stream parallelism). Bindings may also be retrieved during unification and clause evaluation in clause-processes.

A process initiates a retrieval by querying another process with a *get binding* message; the *target* process is identified by the variable's type and tag. The target process is the parent process 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 bound yet. If it has, then the value is returned; if the variable is FREE, for a read-only variable, the target process *suspends the sender* until the needed binding is available. The sender is not suspended for normal variables; instead, it is informed that the variable is unbound. If the variable is of type REFERENCE, then the process in the tag of this variable becomes the new target process and follows the same procedure, until a value or a variable of type FREE is found. Internal dereferencing (i.e. within a process) may be necessary before deciding whether a variable is bound, FREE or REFERENCE.

Note that during this "chaining" procedure, the processes along the path are suspended until the chaining ends. At that time, the chain of processes are successively unblocked with replies. If the variable is FREE, the last tag in the chain is included in all the replies so that subsequent inquires regarding this same variable need not go through the entire chain of processes.

Clause-processes and conjunction-processes may instantiate variables in other processes using the *set binding* message. After a clause-process has been committed, it may start sending *set binding* messages. Conjunction-processes may send *set binding* messages any time they receive updated information regarding variables.

A process instantiates a variable in another target process by sending a *set binding* message. As with retrieving bindings, the target process is determined by the variable's type and tag. For example, when a clause-process commits it sends bindings for instantiated variables occurring in the goal to its grandparent conjunction-process. If the variable is of type REFERENCE, then the process identified by the tag is also notified. The sender of the *set binding* message is suspended until the receiver of the message verifies that the new binding is consistent with bindings already received for that variable. If the receiver is permitted to distribute the binding, the binding is forwarded to the process identified by the tag of the variable in the receiver process, if any. In this case, the sender is suspended until the the new target process replies with the consistency status of the binding. This "chaining" procedure is similar to that employed for the retrieval of bindings described earlier.

For REFERENCE-REFERENCE instantiations in which two variables contain different tags, the owner of *one* of the variables has to update its local version of the variable to indicate this new link. The selected process is sent a *set binding* message. If the variable in the owner process is of type FREE, then the variable simply becomes type REFERENCE and the tag of the other process adopted. If the variable is of type REFERENCE, then the tag of the other process becomes the variable's new tag; in addition, the process identified by the old tag is notified of the update (via a *set binding* message). This results in another chaining effect which terminates when a variable of type FREE is encountered, or when a variable binding is found. For the latter case, the value of the binding is returned back through the chain, causing the processes along the path to instantiate the variable.

5.4. Unification

Tags are built in clause-processes, as a result of unification of unbound unannotated variables. (unbound read-only variables are handled by goal-processes). The unification algorithm is similar to Levy's [8], with its concept of *demand driven copying*. However, variables occurring in structures are handled differently and variables may reside in different processes.

There are two types of reference in Port Prolog's model: pointers and tags. Pointers are used within a process whereas tags are used between processes. Within each process, each variable process is associated with a pointer and (space for) a tag. Pointers are initially null and are set during unification and clause evaluation. Tags are null for FREE variables, and contain a process id for REFERENCE variables. Tags may be used to access variables any time but are used to instantiate variables only after the clause-process has been committed.

In accordance with the two kinds of variable reference, unification proceeds at two levels: locally within a process and globally among the processes. These two levels distinguish what happens *before* the clause-process commits and *after* the clause-process commits. Theoretically, unification occurs only in the clause-process when the given goal and head of the given clause are being unified. However, some form of "matching" is necessary when bindings are received for variables in both

clause-processes and conjunction-processes. Hence, "before" refers to matching of the goal and head before the commit; whereas "after" refers to matching after the commit in the clause-process and *any time* the conjunction-process receives bindings. The previous section describing the retrieval and setting of bindings is basically the gist of the global unification strategy. The rest of this section describes features of the local unification algorithm that are designed to support the global scheme (e.g. constructing tags).

Unification of the goal and the head of the clause proceeds as in any sequential Prolog interpreter with the following deviations. First, the types of variables (FREE or REFERENCE) are ignored and only the status of the variable's pointer is used. Pointers are set so that head variables always point to goal variables in null-null confrontations. Second, if a head variable is bound to a goal variable, a tag is constructed for the head variable using the clause-process's process identifier and variable location. Similarly, tags are constructed for head variables occurring in structures bound to goal variables. This allows other processes to provide bindings for the head variables, and avoid processes from inadvertently instantiating a goal variable before the corresponding clause is committed.

As Levy indicates [Levy84], the Concurrent Prolog language does not specify when values of variables should be propagated. Likewise, the promptness with which bindings should be retrieved is not defined. Therefore, when a clause-process encounters an unbound variable during local unification, it may choose to retrieve the binding immediately, or wait until just before it commits. Note that the clause-process must retrieve bindings for all unbound variables and check their consistency with local bindings before committing.

When a clause-process commits, the binding environments of the goal variables and the head variables are "unified." This procedure is also carried out upon the arrival of *set binding* messages, albeit only for the variables involved and not for the entire environments. A head variable whose tag is still original (i.e. contains the clause-process's process identifier and the variable's location) and is bound (via its pointer) to a goal variable acquires the goal variable's tag. For a FREE goal variable bound to a FREE goal variable, a tag is constructed for one of the variables using the owner's‡ process identifier and the other variable's location in the owner process; the other variable remains FREE. The owner of the variables is notified. A FREE goal variable bound to a REFERENCE goal variable takes on the REFERENCE variable's tag and the owner of the previously FREE variable is notified. When unification involves an unbound goal variable and a value, the value is distributed, as described above.

6. Port Prolog communication model

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

6.1. Conjunction processes

A conjunction-process manages the proof of a given sequence. It creates a goal-process for every atom in the sequence and passes the atoms to their corresponding goal-processes. After completing these initialization duties, the conjunction process 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 sequence. It is designed to always record and then immediately pass on bindings to its parent to make them available for shared non-read-only variables. When all atoms have succeeded, the conjunction-process returns any variable bindings not yet sent to its parent process and then destroys

‡By definition, the owner is the immediate ancestor of the current process.

itself.

During service mode, the conjunction-process must also remember the state of execution of each atom. The possible states are: UNCOMMIT, COMMIT, SUCCEED, and FAIL. A atom is in the COMMIT state if the evaluation of clauses with the same predicate name and arity has produced a candidate clause. A atom is in the UNCOMMIT state if a candidate clause has yet to be found. A COMMIT atom is always associated with its candidate clause-process whereas an UNCOMMIT atom is always associated with its goal-process. Initially, all atoms are in the UNCOMMIT stage. A atom is in the SUCCEED state if the candidate clause has been successfully solved. The fourth state, FAIL, is never recorded because once an atom in the sequence fails the conjunction-process itself fails and states are no longer needed.

6.1.1. Conjunction process incoming messages

Variables may be instantiated with the *set binding* message. When a *set binding* message arrives, the conjunction-process records the binding and passes it onto any process that was waiting for it. A *set binding* message is also used to provide a binding for a variable that has already been instantiated but contains unbound sub-components. In this case, the new binding is unified with the old binding and the appropriate processes are notified (see previous section).

Other processes may obtain binding information from the conjunction-process by sending it a *get binding* message. A *get binding* message contains two parts: the first identifies the variable (say, X) that *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 *get binding* message.

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

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

A *fail* message indicates that an atom 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 atoms in its sequence (by destroying the processes which are doing the evaluations), informs its parent of the failure, and destroys itself.

The *set binding,* messages 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.

6.1.2. Conjunction process outgoing messages

The conjunction-process sends messages to its parent and other conjunction-processes. A *success* message is sent to the parent process when the conjunction-process have received a *success* message for each atom in its sequence. A *fail* message is sent to the parent when the conjunction-process receives a *fail* message. When the conjunction process sends a *set* or *get binding* message to another process, this message fails if its binding is inconsistent with those of the receiving process; in this case, the conjunction process sends a *fail* message to its parent.

Since all variable bindings that the conjunction-process receives are *commit*ted, any binding received may be forwarded immediately to its parent via the *set binding* or *get 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 prototype implementation described here does not have this feature.

6.2. Goal processes

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

Each clause-process receives the goal-process atom and a clause which may unify with the atom. 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 for 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 atom or no candidate clause-process is found, the goal-process sends a *fail* message to its parent conjunction-process and destroys itself.

6.2.1. Goal process incoming messages

The goal-process is the simplest of the three process types. It handles only 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.

6.2.2. Goal process outgoing messages

All messages from a goal-process are directed towards its parent conjunction-process. Upon the reception of a *commit* message, the goal-process issues its own *commit* message to its parent conjunction-process. This *commit* message contains information that identifies the committed candidate clause-process.

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.

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

6.3. Clause processes

A clause-process is responsible for the evaluation of the given atom using the given clause. The first thing that a clause-process does is attempt to unify the given atom 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 acts 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 it then 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 fig. 2). 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 atom 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

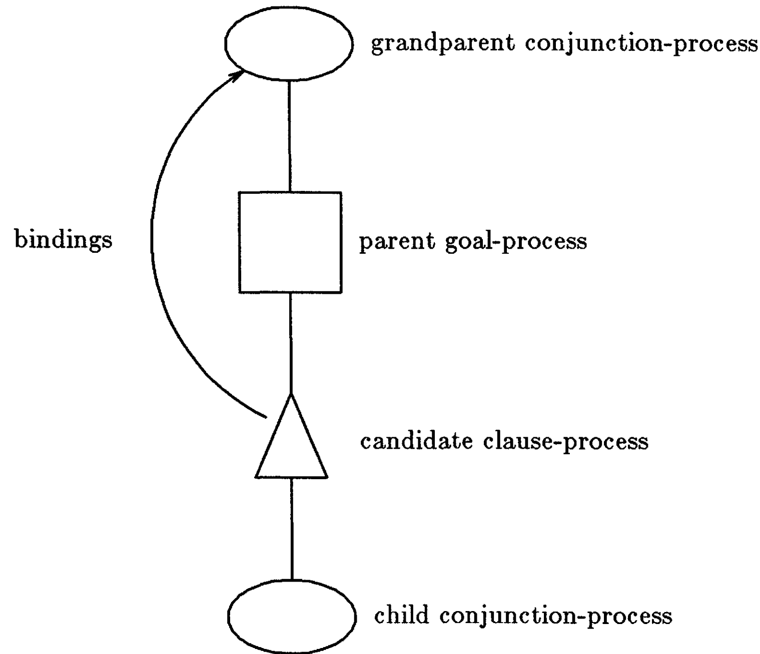


Figure 2. Candidate clause process

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. The clause-process then destroys itself.

6.3.1. Clause process incoming and outgoing messages

A clause-process receives status messages from its child conjunction-process. It receives a *success* or *fail* message depending on the evaluation of the sequence 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. The handling of *set binding* and *get binding* messages is discussed in section 5. through the use of read-only variables.

7. Synchronization

Synchronization in the Concurrent Prolog language is achieved through the use of read-only variables (see fig. 3). Execution of an atom 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 an atom is suspended until all read-only variables (if any) are instantiated. This suspension involves the goal-process sending a *get binding* message to a conjunction-process for every unbound read-only variable (see fig. 3). Suspension is possible due to the blocking nature of the send primitive, as assumed by our process abstraction.

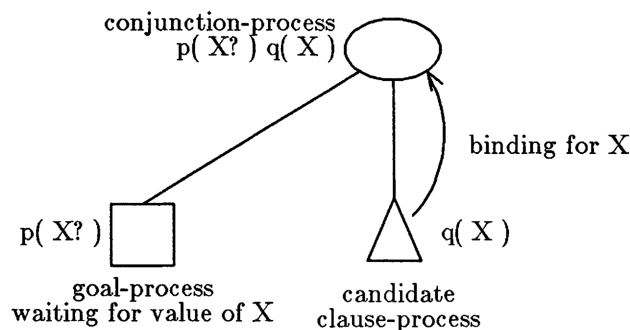


Figure 3. Synchronization of variable bindings

8. System Predicates

System predicates or *built-in predicates* are atoms 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*. Port Prolog handles built-in predicates in processes outside the solver processes to avoid duplicating the code for handling all system predicates throughout the solver process tree. *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 *get 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 use SPP's with 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 is done after successful unification of the given atom 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 atom 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 sequence. In the case of a system predicate, the sequence 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 [10] requires extra synchronization predicates to provide the necessary interface to system functions like I/O. For example, *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

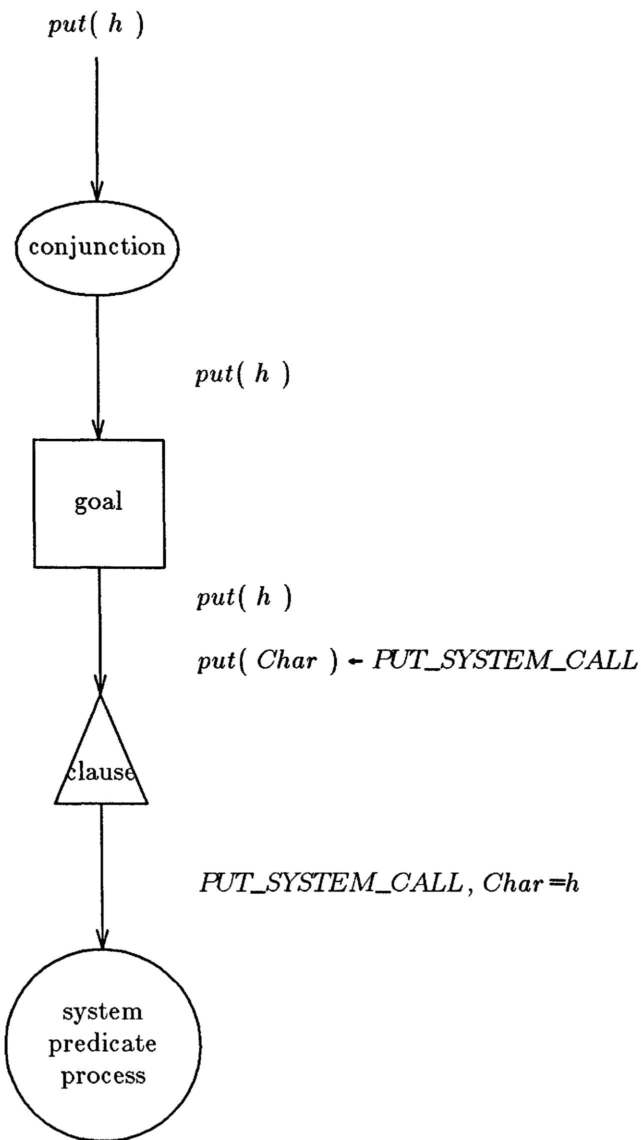


Figure 4. System predicate process

interface is obviated; all predicates, whether built-in or normal, use the same means for synchronization — namely, the read-only annotation (*X?*).

9. Implementation

Here we provide an overview of the current implementation of the Port Prolog interpreter. It implements a subset of the model just described. Further details are provided in [7].

9.1. Implementation environment

Port Prolog is implemented in Waterloo Port [5,9], an operating system developed by the Software Portability Group at the University of Waterloo. It is a multi-process operating system that supports concurrent activities and multiple windows. This allows several programs to be run and viewed simultaneously. The development of Port Prolog was done with a Port system configured for an IBM personal computer workstation with a monochrome display and a mouse.

9.2. Implementation process structure

The Port Prolog interpreter is a multi-process Port activity. Both the run-time solver and the rest of the system are structured as several processes. For example, a group of processes manage the database while other processes provide services like parsing, window management, etc.

Because each logical component of the interpreter is implemented as a process, the interpreter is flexible and easy to maintain. Since information sharing is possible only through message-passing, the interfaces between the processes are well-defined. 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; the sequential Prolog's parser does not handle guard sequences and its solver implements van Emden's ABC algorithm [6].

Here we give a brief overview of each process in the interpreter. 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 I/O 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 to the solver component for evaluation.

In Port Prolog, the solver consists of a hierarchy of processes with three generic process types described above: *conjunction-process, goal-process* and *clause-process*. 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 I/O server calls upon the *vulture* process to destroy all the processes created during the invocation of the interpreter.

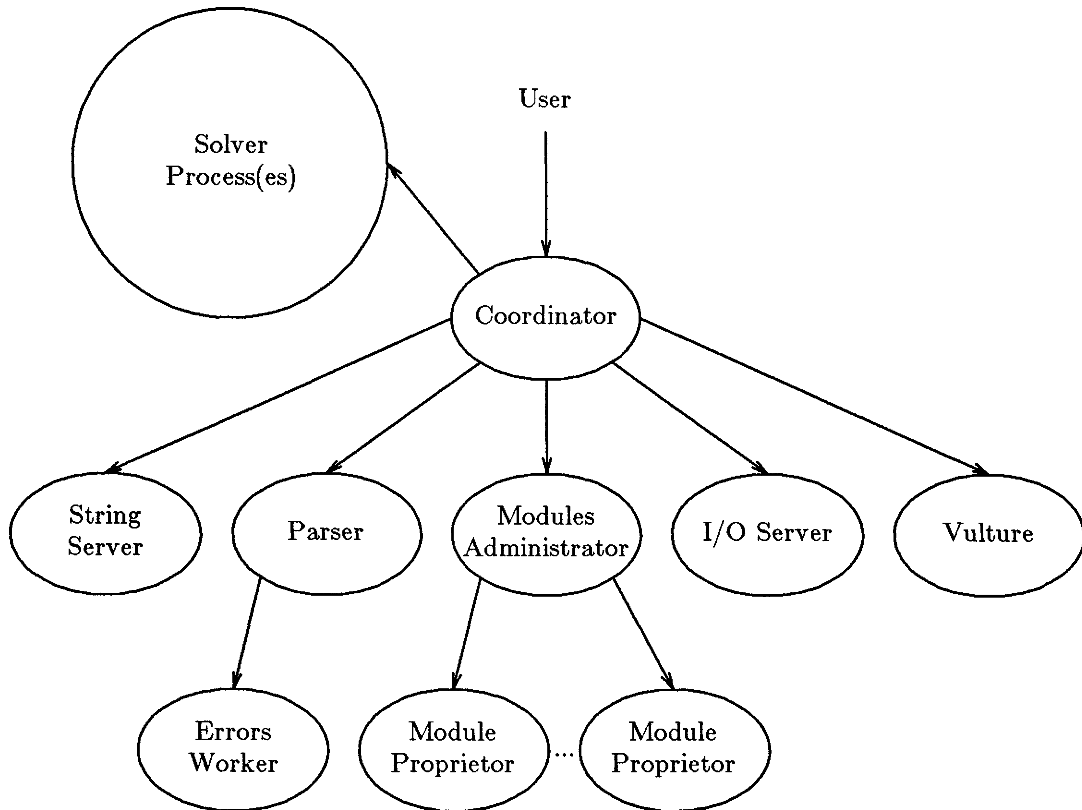


Figure 5. Genealogical process structure of Port Prolog

9.3. The solver processes

9.3.1. Data structures

The major data structure is the *environment*. It is used (not shared) 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.

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

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.

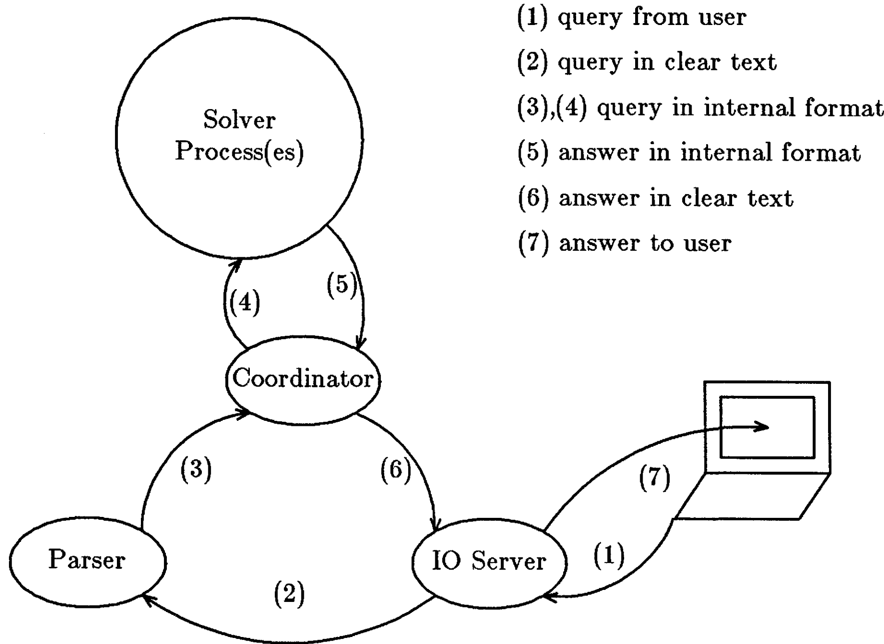


Figure 6. Processing of a query

environment
OFFSET
TYPE
PROCESS
BINDING
SIZE
WAIT_LIST
NEXT_VARIABLE

atom map
LITERAL
PROCESS
STATE
NEXT_LITERAL

clause list
PROCESS
CLAUSE
CLAUSE_LENGTH
NEXT_CLAUSE

Figure 7. Solver processes' data structures

9.3.2. Conjunction process implementation

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 sequence from its parent and creates an environment for all the variables in the sequence. It then breaks up the sequence and creates a goal-process for each atom. A atom map is constructed to store information about each atom.

The conjunction-process, after creating the goal-processes, is ready to handle five types of request. A SET_BINDING request provides a new value. 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. A GET_BINDING request is looking for a value. 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. The FAIL request causes the conjunction-process to terminate by sending a FAIL message to its creator, and then destroy itself and all of its descendants. The SUCCESS request causes the conjunction-process to search its atom 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 atoms 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. A COMMIT request causes the conjunction-process to search through its atom 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.

The above description of the conjunction-process is a partial implementation of the conjunction-process described earlier. It handles a *set binding* message naively, without considering possible inconsistencies or forwarding the message. It handles *get binding* messages, but can only deal with one variable per message. The conjunction-process currently implemented does not send *get 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.

9.3.3. Goal process implementation

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

The goal-process then retrieves all clauses from the database that may unify with the atom. 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.

After creating the necessary clause processes, the goal-process then waits for messages. If the goal-process receives 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.

Upon receiving a COMMIT message, the goal-process traverses its clause list and for every entry which does not contain the process id of the sender of COMMIT, and the goal-process destroys that 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.

The goal-process as described in the computation model has been implemented, except for referencing tagged variables. 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 the component of the goal-process that communicates with the module proprietor needs to be modified. The

required modifications are described in [7].

9.3.4. Clause process implementation

A goal-process creates a clause-process for every clause whose head has the same predicate name and arity as its atom. When created, a clause-process first tries to unify the clause head with the goal atom. 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, clause-processes handle SET_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 SET_BINDING request to that grandparent for each variable that has been instantiated. The clause-process handles a SET_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 implementation of the clause-process lacks several features described earlier. 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 *get binding* messages. This means that Port Prolog cannot handle queries in which variables unify with each other or unify structures containing variables.

The system predicate process has not yet 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.

9.4. Binding environment

In the current implementation, the following information is kept for each variable: 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, and the variable's binding (see fig. 7). 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) by the conjunction-process when all the atoms 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, the mapping of variables in the clause process, and the dereferencing across processes have not been implemented.

10. Example programs

Here follow some examples of the subset of Concurrent Prolog supported by Port Prolog. 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 );
```

invocation of *choose(A)* 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 execute until *A* is instantiated by *pred2*. When *pred1* does execute, 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 below is another example of a Port Prolog program.

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

$benchmark_1$: stack([push(1),pop(1)]);
$benchmark_2$: stack([push(1),push(2),push(3),pop(3),pop(2),pop(1)]);

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.

10.1. Breakdown of execution time

Fig. 8(a) shows the breakdown of the time Port Prolog takes to solve the benchmark queries $benchmark_1$ and $benchmark_2$. Each measurement category (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. Fig. 8(b) shows the number of processes used in each benchmark. These measurements tell 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.

11. Summary

The notion of independent processes communicating via message-passing is a powerful tool for designing systems, especially systems in which asynchrony and parallel execution have to be controlled. Port's interprocess communication primitives and process abstraction greatly influenced the design of Port Prolog's computation model. The independence of a Port process (i.e., no shared memory) would also reduce the work of re-designing a functionally identical interpreter for a multiprocessor implementation.

(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 8. Evaluation of benchmarks

The current implementation demonstrates that variable bindings can be passed between processes, and hence shared variables are possible in our model. In the current implementation, synchronization between dependent atoms in a clause can be achieved if the variable involved does not contain other free variables. 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. (see Tam [11] for a more detailed description of testing for deadlock.) Also, with fully implemented retrieval and unification algorithms, we can measure the communication overhead on a more realistic set of concurrent Prolog programs and further observe the cost of shared variables.

Acknowledgements

We would like to thank Romas Aleliunas, and W. Morven Gentleman for their comments on an earlier draft of this paper.

References

[1] D.R. Cheriton, M.A. Malcolm, L.S. Melen, and G.R. Sager (1979), Thoth, a portable real-time operation system, *ACM Communications* **22**(2), 105-115.

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

[3] K.L. Clark and S. Gregory (1981), A relational language for parallel programming, DOC 81/16,

Department of Computing, Imperial College, London, England, July.

[4] J.S. Conery and D.F. Kibler (1981), Parallel interpretation of logic programs, *Proceedings of the ACM Conference on Functional Programming Languages and Computer Architecture*, October, 163-170.

[5] P.A. Didur, M.A. Malcolm, and P.A. McWeeny (1984), Waterloo Port User's Guide, Waterloo Microsystems Inc., Waterloo, Ontario, February.

[6] M.H. van Emden (1981), An algorithm for interpreting PROLOG programs, CS-81-28, Department of Computer Science, University of Waterloo, Waterloo, Ontario.

[7] R.K.S. Lee (1984), Concurrent Prolog in a multi-process environment, M.Math thesis dissertation, Department of Computer Science, University of Waterloo, August, 133 pages [ICR Report 24; Department of Computer Science Technical Report CS-84-46].

[8] J. Levy (1984), A unification algorithm for Concurrent Prolog, *Proceedings of the Second International Logic Programming Conference*, July 2-6, Uppsala University, Uppsala, Sweden, 333-341.

[9] M.A. Malcolm, B. Bonkowski, G. Stafford, and P.A. Didur (1983), Programming in Waterloo Port, Waterloo Microsystems Inc., Waterloo, Ontario, December.

[10] E.Y. Shapiro (1983), A subset of concurrent Prolog and its interpreter, Technical Report TR-003, Insitute for New Generation Computer Technology, Tokyo, Japan.

[11] C.M. Tam (1984), The design of a distributed interpreter for Concurrent Prolog, 84-18, Department of Computer Science, The University of British Columbia, Vancouver, British Columbia, November.

[12] R. Vasudevan (1984), Performance measurements on the Port kernel, Software Portability Laboratory, University of Waterloo, Waterloo, Ontario [in preparation].