THE DESIGN OF A DISTRIBUTED INTERPRETER FOR CONCURRENT PROLOG

by

Chun Man Tam

Technical Report 84-18

November, 1984

ABSTRACT

Prolog is a programming language based on predicate logic. Its successor, Concurrent Prolog, was designed to meet the needs of a multiprocessing environment to the extent that it may be desirable as a succinct language for writing operating systems. Here, we demonstrate the feasibility of implementing a distributed interpreter for Concurrent Prolog using traditional programming tools under a multiprocess structuring methodology. We will discuss the considerations that must be made in a distributed environment and how the constructs of the language may be implemented. In particular, several subtle pitfalls associated with the implementation of read-only variables and the propagation of new bindings will be illustrated. In addition, a modification to Shapiro's treatment of read-only variables is proposed in an attempt to "clean up" the semantics of the language.

(The discussion will centre around a primitive version of an interpreter for the language written in Zed (a language similar to C) on an Unix-like operating system, Verex. Although a brief introduction of Prolog and Concurrent Prolog will be given, it is assumed that the reader is familiar with the paper A Subset of Concurrent Prolog and Its Interpreter by E.Y. Shapiro [Shapiro83].)

The Design of a Distributed Interpreter for Concurrent Prolog

by

CHUN MAN TAM

B.Sc., The University of British Columbia, 1981

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF

THE REQUIREMENTS FOR THE DEGREE OF

MASTER OF SCIENCE

in

THE FACULTY OF GRADUATE STUDIES

Department of Computer Science

We accept this thesis as conforming

to the required standard

THE UNIVERSITY OF BRITISH COLUMBIA

September, 1984

© Chun Man Tam, 1984

CONTENTS

1.0 Introduction **1**

1.1 Motivation for Logic Programming 2

2.0 Sequential Prolog **5**

2.1 Procedural Semantics of Prolog 6

2.2 Declarative Semantics of Prolog 8

2.3 Associated Problems with Prolog 10

3.0 Introducing...Concurrent Prolog! **13**

3.1 Guards, Clean Up These Cuts! 15

3.2 Read-Only Variables 17

3.3 Perpetual Processes 22

4.0 Overview of Target Machine/Environment **23**

5.0 Design of the Interpreter **27**

5.1 Representation of Axioms 27

5.1.1 Compacted Representation of Axioms 28

5.1.2 Structure-Sharing 30

5.2 Unification Under Our Representation 32

5.3 Division of Workload Between Processes 35

5.3.1 Goal Process (Or-Node) 35

5.3.2 Clause Process (And-node) 36

5.4 Dealing with Successes/Coping with Failures 39

5.5 Mutual Exclusion Considerations 46

5.6 Read-Only Variables 49

5.7 Supporting Dynamic Databases 50

6.0 Evaluation and Conclusions 53

6.1.1 Possible Optimizations 56

6.1.2 Summary 57

7.0 References 61

LIST OF ILLUSTRATIONS

Figure 1. Sample Prolog Code - Append (list1, list2, result). 5

Figure 2. Comparison of (P->Q) with (¬P or Q) 7

Figure 3. Sample And-Or Solution Tree 7

Figure 4. Concurrent Prolog Equivalents of A Conventional Language . 14

Figure 5. Sample Axiom for Concurrent Prolog 16

Figure 6. Example of the Commit Operator 17

Figure 7. Verex Communication Primitives 24

Figure 8. A Lisp CONS-cell 28

Figure 9. Compacted/Contiguous Representation of Axioms 29

Figure 10. Structure Sharing: Skeleton and Variables of An Axiom . . . 31

Figure 11. Reference Loops - Unifying Two Unbound Variables 33

Figure 12. Process Tree from An And-Or Tree 36

Figure 13. Deadlock: Broadcast by Parent Clause-Process 41

Figure 14. Deadlock: Broadcast by Reporting Process 42

Figure 15. Creating a Broadcast Server 43

Figure 16. Mutual Exclusion Problems of "Cross-Layer" Reference Loops 47

Figure 17. Publicizing References 48

Figure 18. BNF description of Syntax Accepted by Current Interpreter . 54

Figure 19. Performance Measurement on a Bounded-Wait Merge 55

ACKNOWLEDGEMENTS

I would like to thank the many professors that I have had the pleasure of meeting during my years at the University of British Columbia especially those who have introduced me to the fields of logic programming and operating systems. In particular, I would like to thank my supervisor, Dr. H. Abramson, and A. Kusalik for their guidance and helpful comments in the researching this thesis.

1.0. INTRODUCTION

There are currently three major classes of programming languages. These are:

1. Procedural (von Neumann) Languages
2. Functional Languages
3. Logic Programming

Procedural languages such as Pascal, Algol, Fortran, Cobol, and PL/I have "side-effects" as their underlying theme.

- A variable is assigned a value.
- A variable may be bound to zero or more values.
- A program is a description of a sequence of actions or procedures to be performed (hence the name "procedural").

Functional languages (the most predominate being LISP) are based on Church's Lambda-Calculus. They are (ideally) characterized by:

- No side-effecting
- Values return via functions only (rather than side-effects on "global" variables)
- Exactly one value is returned by a function
- Programs resemble the definition of desired computations.

Logic programming languages are primarily based on predicate-calculus. Major characteristics of logic programming include:

- No side-effecting
- Programs resemble clauses/axioms of first-order logic and are simply statements of facts.
- The computation of a program can be considered as invoking a mechanical theorem prover to assert the desired goal.
- "Results" of a computation are obtained by instantiating the unbound terms of a logical relation (predicate).

1.1 MOTIVATION FOR LOGIC PROGRAMMING

The major advantages of logic programming languages over the other two classes can be seen by comparing the above list of attributes:

- Programs written in both functional and logic languages resemble the definition of the problem rather than the algorithm from a procedural language. Intuitively, it appears easier to state the definition of a problem than to devise the algorithm to solve it.

- Since logic programs are simply lists of facts, they can be verified more easily. One need simply examine each statement, independent of all other statements, and decide whether it is a correct representation of the domain. For a procedural language, we must also understand all the inter-dependencies between each statement.

- A variable in a procedural or functional language may be assigned zero or more times. It is the responsibility of the programmer to initialize any variable before using it in a computation and to assure that variables are not assigned inadvertently. In a logic program, a variable may be bound at most once. Hence, we need not worry about some errant portion of code invalidating a legitimate value. Furthermore, an axiom still holds even if one of the variable is uninitialized; the result of the computation would then be expressed as a relationship to the variable (eg.father(x), y+1) rather than concrete values such as John or 3.

- For those who have noted that the differences stated so far between a functional language and a logic language are only minor ones, it is suggested that logic programming is, in fact, more powerful. Since a logic program "returns" values via arguments of a predicate, a "procedure" may return a set of values whereas a function, by its definition, may return exactly one value. Also, since logic languages are simply statements, there may be similar statements (each defining a specific case of the problem); if more than one statement can result in the success of the goal (ie. multiple solutions), then the choice of statements is random. A goal invoked several times may result in several indeterminate results.

As the desire for verifiable programs increases and the cost of computing power decreases, logic programming is gaining much interest. Prolog (PROgramming in LOGic) [Kowalski74] [Warren77] is a programming language which is considered to fall into this category. This thesis examines the feasibility of applying Prolog in a multiprocessing environment by implementing a distributed

interpreter for a variant of the language, Concurrent Prolog, introduced by Shapiro [Shapiro83].

In the next chapter, an overview of Sequential Prolog is presented, including a discussion of its bad points along with the good. Chapter Threeintroduces the features of Shapiro's Concurrent Prolog. The target system, Verex, is described in Chapter Four followed by Chapter Five which discusses the problems and design decisions of the implementation. An evaluation of the interpreter is presented with performance measurements in the concluding chapter.

2.0 SEQUENTIAL PROLOG

A program in Prolog [Kowalski74] [Warren77] [Nilsson80] is a set of "statements" of the form[1]

```
P  <-  Q1 & Q2 & ... & Qn.
```

and can be viewed in two ways:

1. as an implication rule where Q1, Q2, ..., Qn are the antecedents and P is the consequent,
2. as a procedure declaration with P as the head of the procedure and Q1, Q2, ..., Qn as the body.

```
Append ( *list1,  nil,  *list1 ).
Append ( *head1.*tail1,  *list2,  *head1.*sublist )
  <- Append ( *tail1,  *list2,  *sublist ).
```

Figure 1. Sample Prolog Code - Append (list1, list2, result).

In the above example, only the definition of append is given:

1. When the second list is empty, the result is simply the first list.

---

[1] We will use the syntax from Prolog/MTS [Goebel80] and augment it where necessary. (The infix operator "." is the constructor predicate which can be thought of as a concatenation symbol.)

2. Otherwise, the new list is simply the head (or first element) of the first list followed by the list formed by the concatenation of the tail (or remainder) of the first list with the second list.

There were no statements instructing the interpreter how to construct a new list. The definition of the problem is itself the program.

2.1 PROCEDURAL SEMANTICS OF PROLOG

From the programmer's point of view, there are two major differences between Prolog and a procedural language:

1. There may be multiple "procedure declarations" with the same name. When a procedure P is to be executed, a two-way pattern matching scheme called unification is first used to pick which procedure is to be invoked. Then, execution proceeds recursively with each "statement" in the body of the chosen declaration. (The system is said to have resolved [Robinson65] from the head of the clause to the system consisting of the terms in the body of the clause.) If one of these statements causes program failure, the system backtracks to the nearest decision point and a new choice is tried.

2. A formal argument of a procedure may either be an input parameter or an output parameter even though there is no specific indicator (such as the reserved word VAR in Pascal to designate a "value-result" parameter). The choice need not be made in advance. Instead, the choice is determined at

procedure invocation, allowing some programs to be ran "backwards". For example, the procedure ADD (x,y,z) may be ran as

- ADD (1, 2, z) to yield a value of 3 for z
- ADD (x, 3, 5) to yield a value of 2 for x
- ADD (1, y, 1) to yield a value of 0 for y.

This two-way capability is a powerful one in that it turns uni-purpose procedures into multi-purpose ones.

P	¬P	Q	P->Q	¬P or Q
T	F	T	T	T
T	F	F	F	F
F	T	T	T	T
F	T	F	T	T

Figure 2. Comparison of (P->Q) with (¬P or Q)

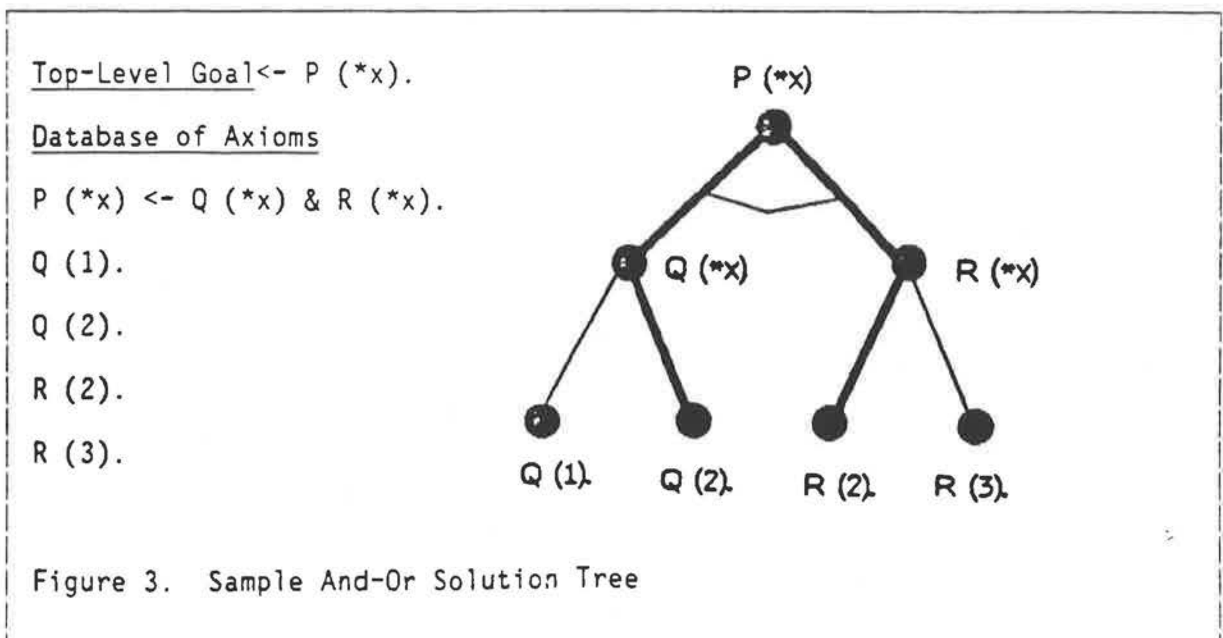


Figure 3. Sample And-Or Solution Tree

2.2 DECLARATIVE SEMANTICS OF PROLOG

Wearing our logicians' caps, the Prolog interpreter is a theorem prover working on a a subset of first-order logic known as Horn Clauses. These are clauses with exactly one negation. (Recall that P -> Q is equivalent to (¬P or Q) as in Figure 2 on page 7). Each clause or implication rule is a statement about what is true about the domain at hand rather than a description of how to compute the solution.

A "solution" is obtained by specifying a clause (possibly with unknowns) as the goal and asking Prolog to find an instance of the unknowns for which the goal can be asserted. Prolog must backchain from the goal using the "reverse" of Modus Ponens (implication). In order to assert the goal Q, in P -> Q, one must assert P, recursively. If P can be asserted then Q must have a truth value of TRUE otherwise TRUE -> FALSE yielding FALSE will invalidate the definition of P -> Q as an axiom. On the other hand, if P cannot be asserted, we cannot comment on the truth value of Q[2]; FALSE -> TRUE and FALSE -> FALSE both yield TRUE.

Thus, computation amounts to searching an And-Or Solution Tree (see Figure 3 on page 7) for a path from the root to one of its leaves. Or-nodes indicate where more than one possible axiom (disjunct) may contribute to the solution. And-nodes of the tree "tie" together the conjuncts of an axiom. (For each

---

[2] [Reiter79] suggests the use of the Closed World Assumption to treat the failure to find a solution as negation and, thereby, assigning the truth value of FALSE to Q.

conjuct, a sub-path must be found spanning the height of the subtree rooted at the And-node.) The tree is obtained by the following sequence:

1. Take the goal as the root (an Or-Node)

2. Find all axioms in the database whose heads are potentially unifiable with the goal and create an And-Node for each, "tying" them to the root.

3. For each of these axioms, treat each term in its body as a new subgoal and recursively create subtrees for them. If an axiom does not have have a body, it is known as a <u>base</u> or <u>ground</u> axiom and forms a leaf of the tree.

(The tree is, of course, an implicit one; that is it is never generated but is implicit in the execution of the interpreter with conjunctions implied by sequential solution of brother-goals and disjunctions implied by backtracking.)

2.3 ASSOCIATED PROBLEMS WITH PROLOG

So far, I have claimed Prolog to be a "logic programming language". In reality, it is only a good approximation. In order to have true logic programming we need true parallelism in trying all choices (disjuncts) and in solving all conjuncts. Prolog is merely a sequential simulation of this process. It accomplishes disjunction by backtracking (where necessary) and conjunction by sequentially resolving each conjunct. Whenever a disjunct proves unsuccessful, any bindings made by that path is undone and an alternative path is tried. (As in first order logic, a disjunction fails when all of its disjuncts fail.) After solving one conjunct, the next subgoal in the conjunction is attempted. (A conjunction fails when any of its conjuncts fails.)

This sequential simulation can either be inefficient using breadth-first searching or susceptible to non-termination (even when the goal is provable!) if depth-first searching repeatedly picks an errant path as its first choice. The strategy employed by Prolog is that of depth-first search supplemented by some control constructs to eliminate some of the duplication of effort between axioms. These constructs are the "cut" operator, a specific order of execution and the introduction of side-effects by allowing creation / destruction of axioms:

Order of Execution Prolog has chosen to define a specific order of execution: the database is searched in order of axiom entry and conjuncts are solved from left to right. The specific search order allows knowledge gained from one axiom to

"flow" to another in the database. Consider two axioms P <- Q & R and P <- ¬Q & R, in that order. If Q causes failure, the second axiom will still necessarily solve ¬Q. By relying on the order of search, the second axiom may be rewritten as P <- R to avoid the duplication of effort.

Axiom Creation and Destruction

Axiom creation and destruction in Prolog is analogous in power to the inclusion of EVAL as a program-callable function in Lisp. In an interpretive system, they allow using user-generated data interchangeably with program source code. In Prolog, however, a major use of these constructs is to maintain state information - using axioms as a form of static variables and defeating the advantage of the single-binding logical variable.

The "Cut" Operator

The "cut" ("/") operator allows the program to direct the execution of Prolog. Since the Prolog interpreter usually has an "If I can solve the problem, don't worry about how I go about computing it" attitude, a major objection to the language is that it lacks the control which is essential in exploiting the knowledge built into the programs. Thus, the cut is used to give more control back to the programmer and to direct the interpreter away from irrelevant choices.

On encountering a "cut", all backtracking information is discarded. All choices made at the time will never be undone. The interpreter commits itself to the current search path. Traditionally, the "NOT" predicate has been implemented using the "cut" and relies on the ordering of axioms as:

```
Not ( *x ) <- *x & / & FAIL.
Not ( *x ).
```

If x can be asserted, then the interpreter commits to the choice of the first "Not" axiom and subsequently causes failure. Otherwise, the second is chosen and the "Not" succeeds.

We have discussed the common usages of each of the three control constructs but not only are they beyond the scope of first-order logic, their use significantly reduces the verifiability of the software. They force the programmer to correctly sequence axioms and to understand the relationships between one axiom and another. Worst of all, the intent of a "cut" is often obscure and eliminates the "two-way" aspect of procedures. In addition, these constructs rely on assumptions which virtually eliminate the possibility of porting the same program to a distributed system.

3.0 INTRODUCING...CONCURRENT PROLOG!

To address some of the criticisms of Sequential Prolog, Shapiro, in his paper A Subset of Concurrent Prolog and Its Interpreter, introduces a variant of Prolog which is endowed with the elegance of multi-process structuring and raises the possibility of using a logic-programming language for the development of operating systems. Below, we briefly examine the properties of Concurrent Prolog and how it qualifies for its intended role of a systems programming language.

Shapiro lists four essential elements for a concurrent programming language: concurrency, communication, synchronization, and indeterminacy. Concurrent Prolog supports these as follows:

Concurrency By simultaneously solving the terms of a conjunction. Each term in the body of an axiom can be regarded as a program/procedure invocation. By executing each program with a separate (possibly, virtual) processor, each becomes a Concurrent Prolog process.

Communication Via the unification of shared variables. When a term with an unbound variable, say P(x), is unified with a term having a constant as the corresponding argument, say P(3), information is being communicated from the latter process to the former.

Synchronization Is accomplished by the introduction of Read-Only variables. A process having a read-only variable must block itself until

the variable has been instantiated by some other process. (We will discuss read-only variables further in a following section.)

Indeterminacy By simultaneously exploring all paths which may provide a solution to the goal. Assuming random allocation of processing power (due to different speeds of actual processors, random processor allocation, etc.) and multiple solutions to a goal, the actual solution returned is a function of time.

Procedure	Axiom
Procedure Call	Solving a Goal
Binding Mechanism	Unification
Process	Unit Goal
System	Conjunction of Goals
Process State	Value of Arguments
Communication	Unification of Shared Variables
Process Synchronization	Read-Only Variables

Figure 4. Concurrent Prolog Equivalents of A Conventional Language

3.1 GUARDS, CLEAN UP THESE CUTS!

We have already mentioned that one of the biggest criticisms of Sequential Prolog is the use of the cut symbol ("/"). Its range and asymmetry often lead to obscure programs. Shapiro recognizing that the control aspect of the cut must be included in a concurrent language, to help synchronize processes, decided to provide a cleaner version called the commit operator ("|"). An example on the use of the commit operator is given in Figure 6 on page 17.

The commit operator is patterned much after the guarded-if of [Dijkstra76]. It splits the body of an axiom into two parts: the guard and the main body (Figure 5 on page 16). In trying a particular axiom as an alternative, the interpreter must solve its guard before solving the body and once the guard of one of the disjuncts is solved, all brother-disjuncts are abandoned.

The reason that the commit is a cleaner operator than the cut is due to its symmetry. The asymmetry appears in two forms:

1. In Prolog, choices of a disjunction are tried sequentially (usually from left to right in the current subtree). When the cut operator is encountered, Prolog commits to the current path. All branches on one side of this path have already failed and the cut "kills" the remaining branches on the other side.

```
H <- G1 & G2 & ... & Gm | B1 & B2 & ... & Bn.
```

Here H is the head of the clause,

G1, G2, ..., Gm are the guards of the clause, and

B1, B2, ..., Bn are the terms in the body of the clause.

When no guards are required, the axiom may be written as

```
H <- B1 & B2 & ... & Bn.
```

Figure 5. Sample Axiom for Concurrent Prolog

In Concurrent Prolog, all guards are executed in parallel and the commit operator effectively "kills" the branches on both sides of the current path.

2. A cut has an effect only when it is executed. If two alternatives exist with only one containing a cut, the second alternative is, in effect, a default. It is this default assumption that renders the program useless in a concurrent system. Because of the differences in (virtual) processor speeds, the default may be mistakenly chosen even if the first alternative would have succeeded. The commit operator, however, forces every alternative to be guarded by some (possibly empty) clause. The success of a goal will always necessitate the elimination its brother alternatives.

```
Database

1.  a  (*x) <- b1 (*x) | c (*x).
2.  a  (*x) <- b2 (*x) | c (*x).
3.  b1 (*x) <- d  (*x) | fail.
4.  b2 (*x) <- d  (*x) | true.
5.  d  ( 1).
6.  c  ( 1).

Top-Level Goal

<- a (*x).
```

Figure 6. Example of the Commit Operator

In solving the top-level goal in Concurrent Prolog, resolution may proceed with axioms A1 and A2 in parallel. The guard in A1 must be reduced using axiom A3 while the guard in A2 must use axiom A4. Suppose A3 solves d (*x) with d (1) before A4 can solve its guard. A3 commits but the scope of committment is local and the choice between A1 and A2 is NOT affected. Thus when A3 fails and causes the failure of A1, the interpreter is free to solve the system using A2.

As an analogy, "A cut is to a commit as an if-then-else is to a guarded-if". Dijskstra argues that every alternative should be associated with a guard instead of having a default philosophy of the if-then-else. He points out that the lack of symmetry of the if-then-else may lead to errant choices in concurrent system. Here, Shapiro uses the same arguments for the use of the commit operator over the cut.

3.2 READ-ONLY VARIABLES

The second extension to Sequential Prolog is the introduction of read-only variables. A variable designated as read-only is the synchronization mechanism

between two processes. Read-only variables classify processes into writers and readers. A process trying to unify an unbound variable in another process with a value can be thought of as a writer process while the latter process as the reader. The read-only annotation ("?") indicates that the current process cannot write to the variable. The reader process must wait until the variable has been instantiated by some other process before proceeding further with the unification.

Formally, if X? is a read-only variable and Y is any variable then Shapiro defines the unification of X? with Y as follows:

1. If X? has been instantiated[3] and Y has been instantiated then unification proceeds as usual with the bindings of X and Y.

2. If X? has been instantiated and Y unbound then Y becomes instantiated with the binding of X.

3. If X? is unbound and Y has been instantiated then unification fails until X? becomes instantiated by some other process sharing X without the read-only annotation.

---

[3] In the context of read-only variables, we only require that the predicate name or principal functor be determined for a variable to be considered instantiated. This was a design decision in Concurrent Prolog to allow for partially determined messages.

[4] When two variables are uninstantiated but are bound to each other, they are said to reference each other.

4. If X? is unbound and Y is unbound then unification succeeds with X and Y referencing[4] each other. Moreover, Y inherits the read-only property from X?.

Two points need to be considered with this definition. First, the definition makes the success or failure of unification time-dependent. Unification may fail at a given time due to read-only variables but may succeed at a later time. In actual practice, a process may unify the two terms repeatedly or eliminate the busy-wait by implementing the third case in a manner such that the process owning Y must block or suspend until X? has been instantiated.

Second, the definition of the last case (X? unbound with Y unbound) is a somewhat controversial one. It states that after unifying a term such as P (x?) with the head of

P (*y) <- G (*y) ... | ...

the variable y inherits the read-only property and the interpreter is allowed to continue to reduce the remainder of the axiom. Several questions concerning this scheme need to be addressed:

- If the primary intent of read-only variables is to synchronize processes by suspending a process until all of its read-only variables are instantiated, is it desirable to allow unification with an unbound variable to succeed and continue?

- Applying Shapiro's definition recursively, if the variable y later becomes bound to some other unbound variable, the latter variable should also

inherit the read-only property. Is it desirable to propagate the read-only property throughout the solution tree?

- Shapiro suggests the use of read-only variables in the head of axioms causing these variables to be strictly output variables. (Such axioms are eliminated from further consideration if the corresponding argument in the goal is a literal.) That is, an axiom of the form

  ```
  P ( x?, *y ) <- ...
  ```

  may only be invoked by a goal of the form

  ```
  <- P ( *x, *y ).     or
  <- P ( *x, literal ).
  ```

  BUT not of the form

  ```
  <- P ( literal, *y ).
  ```

  At the top-level this strategy may be acceptable but elsewhere in the tree, a read-only variable in the head of a clause may cause the read-only property to propagate (possibly several levels) up the tree, not only down it. The upward propagation may cause a process which was previously unblocked to become suspended. For example, if in

  ```
  P ( *x ) <- Q ( *x ) & R ( *x ).
  Q ( *y ) <- S ( *y ).
  S ( z? ) <- ...
  R ( 1 ).
  <- P ( *x ).
  ```

  Q (*x) is solved prior to R (*x) being solved, then the process trying to unify R (*x) with R (1) will suspend due to the variables y <u>and</u> x

inheriting the read-only property from z. Is this form of "dynamic suspension" desirable?

It is likely that the answers to all of the above questions are all "NO".

- It is much simpler to implement a unification algorithm that suspends when attempting to unify an unbound read-only variable with another unbound variable; there is no need to propagate the read-only property up or down the tree.
- The concept of suspending the process until the instantiation of the read-only variable is a more natural definition.
- It is not clear what forms of problems require the additional properties specified by Shapiro.
- More important, one of the major virtues of logic programming is that one statement or axiom may be inspected independently of the other axioms in the database. One should be able to determine whether the unification of a term containing an unbound read-only variable with the head of a given axiom will result in suspension without having to "trace" the outcome of the axiom.

Hence, the remainder of this paper shall replace the last part of Shapiro's definition with

- If X? is unbound and Y is unbound then unification suspends until X? becomes instantiated.

3.3 PERPETUAL PROCESSES

In Sequential Prolog, a primary use of axiom creation and deletion was to save state information. In a concurrent language, this scheme is no longer necessary; a process that stays activated throughout the life of the program never "forgets" its own state. (The operating system would, at least, restore the state when it reactives the process.)

In Concurrent Prolog, a perpetual process is simply an axiom that resolves to itself with possibly different arguments.

eg. P (x.y) <- P (y).

Any local variables would then be the state of the process. The resolution of the original axiom can be considered to be a state transition. (In our example, a state transition takes place in which the state changes, say, from x to y.) With such a simple, well-understood construct, Concurrent Prolog can thus be extended to capture the elegance of an object-oriented language and the definitive power of state transition diagrams.

4.0 OVERVIEW OF TARGET MACHINE/ENVIRONMENT

It has been suggested that Concurrent Prolog can be used as a multi-processing systems programming language. In order for this idea to become reality, the interpreter will necessarily require and provide the same multi-tasking capabilities found in conventional operating systems. Because of this systems-language/operating systems duality, it may be advisable to construct the interpreter using the same design principles!

One school of thought [Cheriton79.1,81] advocates the use of multi-process structuring:

> "Multi-process structuring is the use of several processes to structure programs. The term process is used to mean an entity that executes actions sequentially and deterministically. A process can logically execute concurrently with other processes...."

The Verex operating system [Cheriton79.2], a descendent of Thoth [Cheriton79.1], is based on this design principle. It provides inexpensive or light-weight processes:

- low overhead process creation / destruction,
- low overhead process switching, and
- inexpensive interprocess communication.

```
Send ( id, message )

    Sends the message to the process identified by
    id and  blocks until the destination process
    acknowledges with a reply

id = Receive ( message )

    Blocks until a message is received from any process
    and returns the sender's id and the contents of the
    message buffer

id = Receive ( message, specific_id )

    Similar to the above except the invoking process
    is blocked  until a message is received from the
    specified id

Reply ( message, id )

    Acknowledges the sender identified by id with
    the contents of the invoker's message buffer (the
    invoker does NOT become blocked)

Forward ( message, from_id, to_id )

    Forwards the message received from from_id
    to the  process to_id. To the process
    to_id, it would be as if from_id had
    sent the message directly (Note that the invoking
    process may have alter the message before forwarding it.)
```

Figure 7. Verex Communication Primitives

Furthermore, Verex provides process communication via blocking messages (see Figure 7 on page 24). Processes communicate with fixed-length messages; the sender of the message is blocked until the receiving process acknowledges the message with a reply. Cheriton argues that this is a more natural and more powerful form of interprocess communication:

- Other than for synchronization purposes, processes often have a requirement to communicate data. On non-message-based systems, one must

first use semaphores, etc. to synchronize the readers and writers of the message buffers and then perform the actual transfer of data, making communications awkward. It is more natural to associate a message with the synchronization mechanism.

- The Verex communication scheme can, in fact, be used to simulate semaphores. Hence, it is as powerful as a semaphore-based scheme.

- Blocking-Sends are a natural part of the Remote Procedure Call or Server concept. A conventional program wishing a subtask to be performed invokes a procedure; a Verex process wishing the help of a server simply sends a message and is unblocked when the server has fulfilled the request with a reply.

- Since the sender is automatically blocked, there can be at most one outstanding message per process.[5] The operating system need not concern itself with the problems of dynamically allocating new message buffers.

By exploiting low-overhead processes, it has been demonstrated that Cheriton's concepts are both feasible and attractive:

- [Cheriton79.1] designed and implemented the portable multi-user operating systems Thoth and Verex.
- [Lockhart79] furthered the work of Cheriton, by designing a verifiable system kernel.

---

[5] Concurrency is accomplished via multiple processes rather than multiple messages.

- [Deering83] mapped the state-transition diagrams of the X.25 protocol specifications onto Verex processes, and thus significantly improved the verifiability of his implementation.
- [Boyle82] designed and implemented a distributed version of Verex and showed that Cheriton's design can indeed be implemented on a multi-processor system. In fact, a complete distributed version of Verex, called the V-system, was implemented at Stanford [Cheriton83].

For the implementation of the Concurrent Prolog interpreter, the Verex system seems well suited for the task. It's two most attractive features, inexpensive creation/destruction and low-overhead process switching makes the system ideal for simulating the breath-first searching necessary for Concurrent Prolog.

5.0 DESIGN OF THE INTERPRETER

Two major criteria govern the design of the Concurrent Prolog interpreter:

1. Recognizing that a distributed system will necessarily incur more overhead, due to process scheduling and process switching for example, the interpreter must minimize storage and execution time in order to provide the maximum computing power for the resolution process.
2. The design must not preclude the interpreter from being distributed over several physical processors and multiple address-spaces.

5.1 REPRESENTATION OF AXIOMS

When designing any database, the representation of the data and its relationships is of primary concern. In the case of Concurrent Prolog, the problem is even more critical (if we are to consider the requisite duplication of data across multiple address spaces). We must find a representation which will optimize both storage requirement and execution time.

Usually, it is a trade-off between execution speed and memory size: increasing the amount of redundant data and increasing storage needs will usually increase execution speed, and vice versa. Luckily, there are at least two optimizations available to us.

5.1.1 Compacted Representation of Axioms

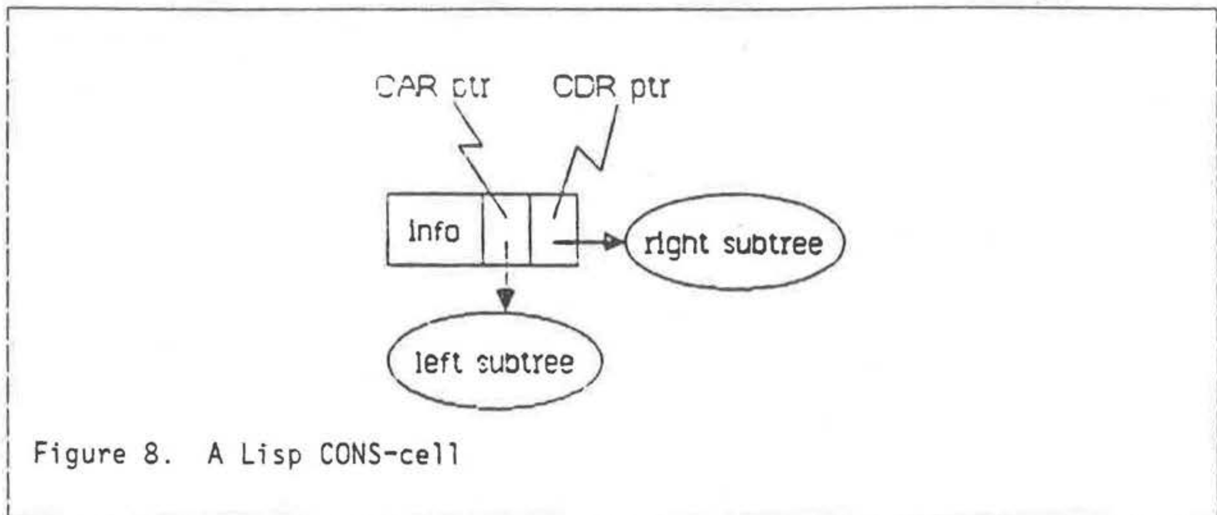


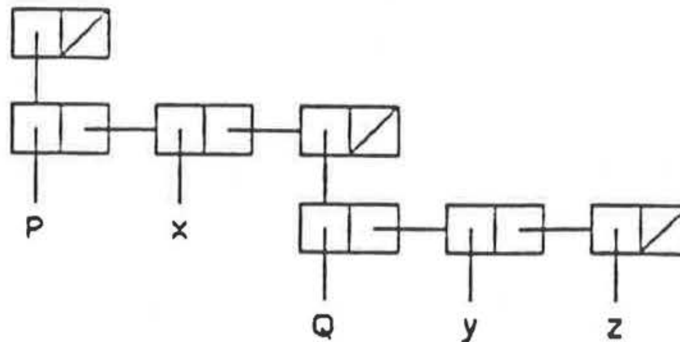
Figure 8. A Lisp CONS-cell

First, we can improve on the representation of tree-like structures such as Lisp expressions and, in our case, Prolog axioms by converting explicit information to implicit knowledge. In particular, explicit CAR and CDR links (Figure 8) to the next item should be replaced by contiguous storage of information whenever possible. In this implementation of Concurrent Prolog, such links have been removed where possible. Succeeding elements are placed contiguously; the cell at position i has an implicit CDR pointing to position (i+1). The structure normally pointed to by the CAR of a cell is now inserted "in-line". For a single-cell element, nothing has changed. But sub-trees now appear in-line with a indicator at the front of the sub-tree pointing to the cell immediately following the last cell occupied by the sub-tree (see Figure 9 on page 29). This compaction scheme reduces our overhead in three ways:

1. Storage is not wasted for links.

P (x, Q (y, z)).

Lisp Representation:



Compacted Representation:

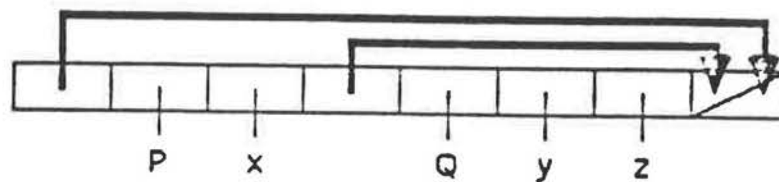


Figure 9. Compacted/Contiguous Representation of Axioms

2. Execution cycles are not required to dereference links (nor to page-in a larger working set under a virtual memory system).
3. Main memory becomes less fragmented. Allocation is done in blocks instead of in small cell-size chunks. (Less fragmentation will likely result in increased performance of any garbage-collection scheme and decrease paging activity.)

5.1.2 Structure-Sharing

The second improvement to our representation is a technique known as structure-sharing (see Figure 10 on page 31). This is a scheme developed by Boyer and Moore [Boyer,Moore72] [Warren77] for Sequential Prolog which is also applicable to Concurrent Prolog in minimizing both storage and execution time. The technique divides an axiom into two parts: the invariant part (or the skeleton of the axiom) and the variables of the axiom.

The variables in the skeleton are replaced by "place-holders". The first unique variable is replaced with a "0", the second with a "1", and so on. The replacement numbers are, in fact, offsets into a vector of the variables of the axiom. So, an axiom is now represented as pointers to two vectors: a skeleton vector and a vector of variables. This representation has the following advantages:

- The separation of skeleton and variables allows the non-changing skeleton to be shared between processes residing in the same address space; thus storage is reduced by not having to duplicate axioms with common skeletons.

- When one of the variables become instantiated, only one location has to be updated even if the variable appears more than once in the axiom. Also, additional storage is not required for subsequent appearances of the variable.

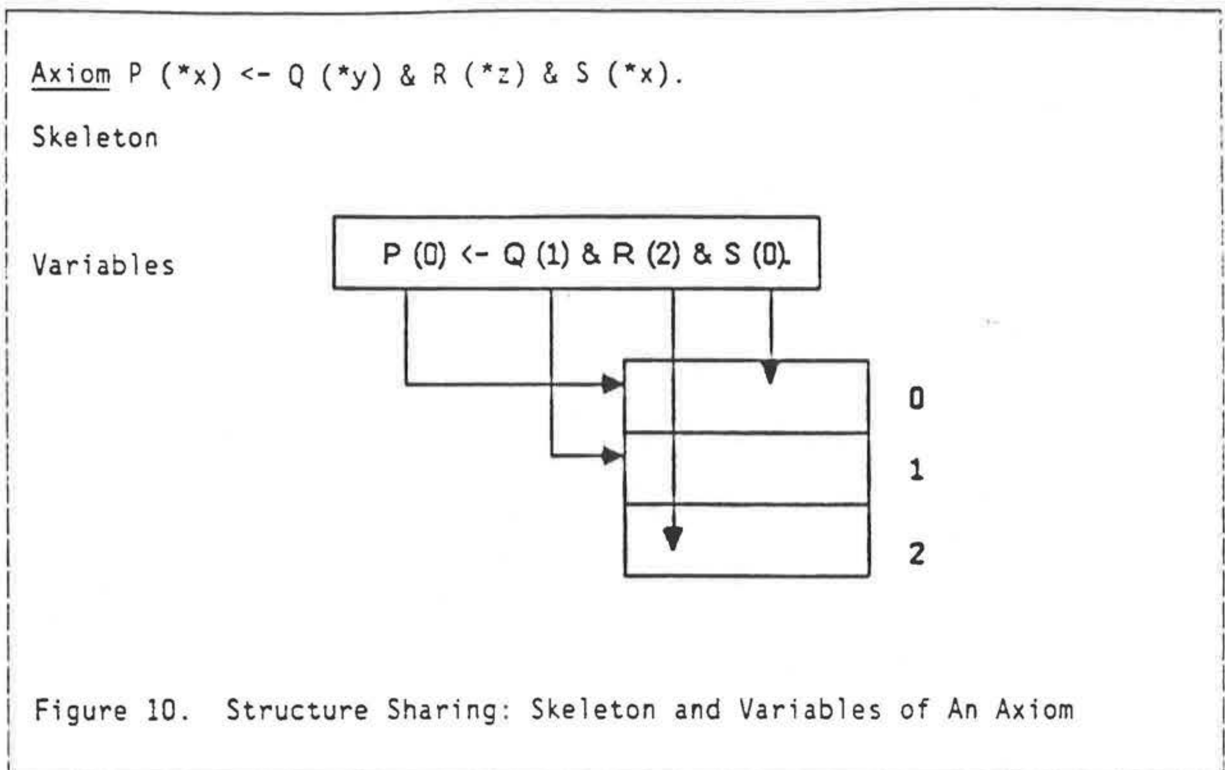


Figure 10. Structure Sharing: Skeleton and Variables of An Axiom

- If the duplication of an axiom is necessary within the same address space, the requirements in both storage and time is proportional only to the number of variables in the axiom. The cells in the skeleton need not be duplicated, only the address of the skeleton need to be copied.

5.2 UNIFICATION UNDER OUR REPRESENTATION

For the most part, unification is simply two-way pattern matching:

- Two atomic terms match only if they have the same value. (Or in our case, each atomic value is given a unique id/address and we need only match id's.)
- Two structures (eg. cons(a,b)) match only if each corresponding sub-structures or sub-terms match recursively.
- If one of the terms is an unbound variable and the other is an atomic value or a structure, then that value is assigned to the variable. (For a structure, the actual value assigned may simply be a pointer/identifier of the form

  Struct (struct_num, offset)

  or a <u>tag</u> [Lee84] of the form

  Struct (process_id, struct_num, offset)

  where process_id is the identifier of the process attempting to solve the structure in question, struct_num is a unique identifier for an axiom and offset is the position of the subterm relative to the beginning of the structure.
- If one or both of the terms is an instantiated variable, then unification proceeds as above using the binding of the variable(s).

The only case that normal pattern-matching fails to handle is when both terms are unbound variables.

Consider the terms:

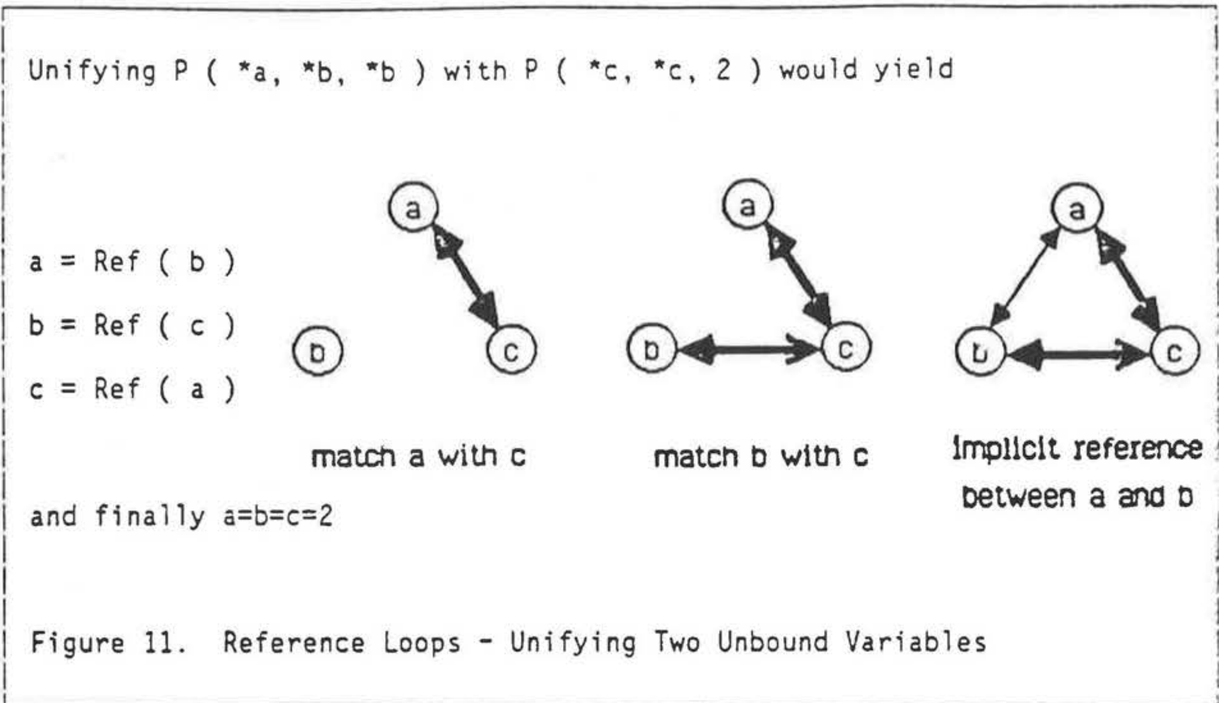


Figure 11. Reference Loops - Unifying Two Unbound Variables

P (*a, *b, *b) and

P (*c, *c, 2).

We would like unification to yield a=2, b=2, and c=2 but (if matching is done in left to right order) we would need to unify two unbound variables. We need a pattern-matcher that would "remember" the relationships between a, b, and c. Initially, a references b, then a and b reference c. If any one of these variables subsequently become instantiated, then all three variables need to be instantiated. A straight forward implementation is to link the three variables together into a circular-list, a reference-loop, by assigning indicators such as Ref(b)[6], Ref(c), and Ref(a) to variables a, b, and c, respectively (see Figure 11).

---

[6] In the actual implementation, the variables would be represented similar to that of a structure (eg. Ref (struct_num, offset)).

When a variable in one reference loop later becomes bound to a variable in a different reference loop, the two loops are merged into one single loop. The ordering within the new loop is unimportant but a self-reference test must be made to ensure that they are indeed two distinct loops. As an example, unifying P (*a, *a) with P (*b, *b) will yield "two" loops joining a and b and an attempt to merge the loops will likely require a great deal of computer time.

One should note that a loop is preferred over a simply-linked list such as those used in [Warren77], [Levy84], and [Lee84]. When one of the variables in the loop becomes instantiated, a traversal of the loop will make sure all variables in the loop are updated. In a simply-linked list such as P? -> Q -> R, if a variable in the middle of the list, say Q, gets instantiated only the variables in the tail of the list are updated. P, in this case, will still not have a binding and may lead to deadlock if the remaining variables have read-only annotations.

5.3 DIVISION OF WORKLOAD BETWEEN PROCESSES

A logical division of tasks is to take the And-Or solution tree (described in a previous section) and create a process for each node in the tree (see Figure 12 on page 36). For each And-node (conjunction of the terms of a clause), create an And-process (or Clause-process). For each Or-node (a choice of solutions in solving a goal), create an Or-process (or Goal-process).

5.3.1 Goal Process (Or-Node)

When invoked with a Concurrent Prolog process (a term in Sequential Prolog), a goal-process executes as follows:

1. It searches through the axioms database for axioms whose head is potentially unifiable with the given term. The searching algorithm may be expedited using the predicate name or principal functor as the primary key and the arity of the predicate as the secondary key.
2. For each of these axioms, it creates a clause-process and invokes it with the given term and the trial axiom.
3. It then waits for one of the clause-processes to commit (i.e. solve the guard clause).
4. If one of the clause-processes does commit, then the goal-process waits for this child to successfully solve the body. If and when the child process reports success, the goal process, in turn, reports success to its parent.

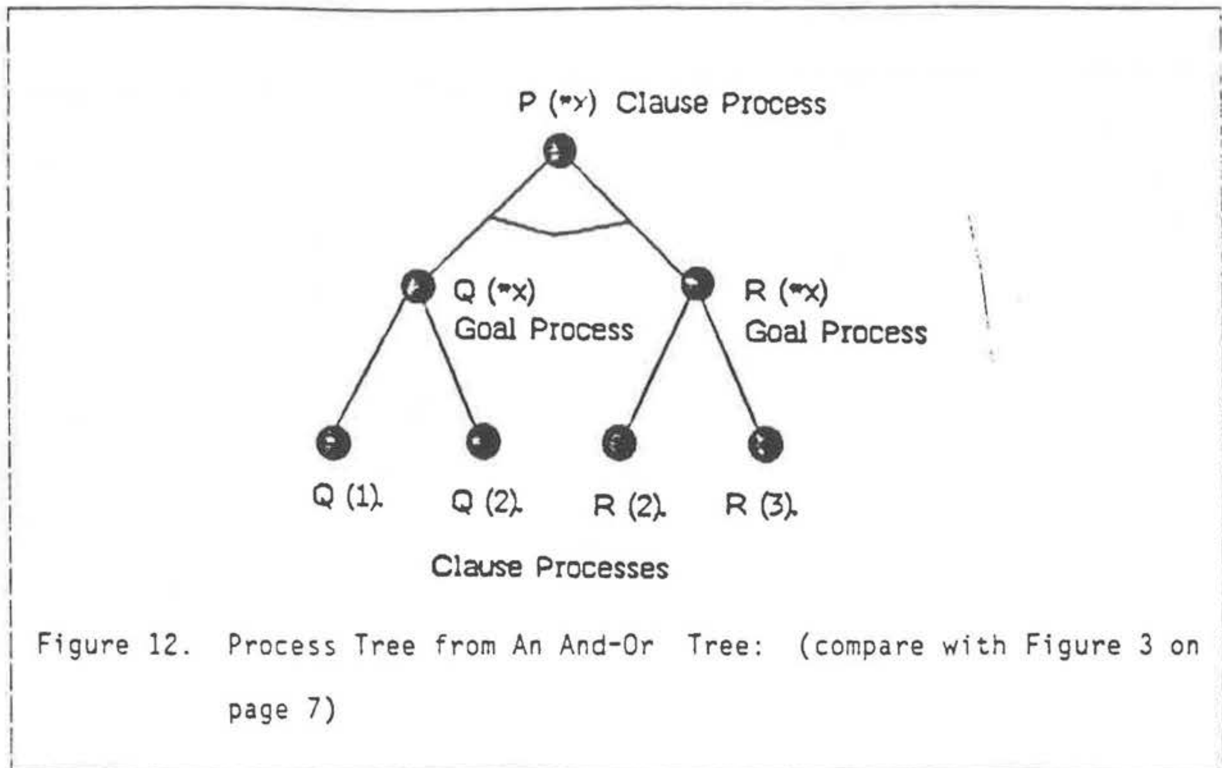


Figure 12. Process Tree from An And-Or Tree: (compare with Figure 3 on page 7)

5. At any time, a child process which does not contribute to the solution of the goal is destroyed and resources allocated to it reclaimed.

5.3.2 Clause Process (And-node)

A clause process is responsible for the solution of a specific term, T such as **cons (a,b)**, with a specific axiom,

H <- G1 & G2 & ... & Gm | B1 & B2 & ... & Bn.

where H is the head of the axiom, G1 through Gm are the guards, and B1 through Bn are the terms of the body.

1. It first makes local copies of T and the given axiom, so that any bindings it generates will not affect other brother clause-processes. Note that if one of the variables references another structure which has unbound variables, that structure must also be copied. Care must be taken to avoid recursively copying the same axioms. For example, if we unified

   1. P (*a, Q (1)). with
   2. P (R (1), *b).

   variable a would be bound to a subexpression within axiom A1 while variable b would be bound to a subexpression within axiom A2. If the implementation copies the entire axiom instead of only the variables in question, then a loop exists between A1 and A2. and a naive implementation would try to create copies of A1 and A2 indefinitely. A simple solution is to modify the copying mechanism to return a <u>mapping</u> between original axioms and their copies. Prior to allocating a new copy of an axiom, a check of the current map must be made to determine whether the axiom has already been duplicated. (More on the use of the map in the Mutual Exclusion section.)

2. It tries to unify T' with H', then for each of the guards, it spawns a goal-process.

3. When all of the goal-processes report success, the clause-process commits and reports to its parent goal-process.

4. After solving the guards, a goal-process is generated for each of B1' through Bn' and the clause-process again waits.

5. When all of its children have reported success, it reports back to its parent. Otherwise, when any child fails, it reports failure.

5.4 DEALING WITH SUCCESSES/COPING WITH FAILURES

If a goal-process commits or terminates successfully, the parent clause-process must make public any instantiations made by the goal-process. First, the scratch-copy from the goal-process must be unified with the global copy from clause-process. Here, only the variables need be unified but direct copying is not sufficient: a brother goal-process may have already instantiated a variable to some particular value and this value must be matched with those from the reporting process for consistency. If unification fails, then all child goal-processes must be destroyed along with the failure of the clause-process.

If unification succeeds, any new information must be made public to ALL the descendents of the clause-process. In addition, if the success of the reporting process causes the clause-process to commit or to complete then a report must, in turn, be made to the parent of the clause-process.

The problem of broadcasting new instantiations is not a trivial one. Suggested solutions include:

Ignore Broadcasting All-Together

We may choose to avoid broadcasting and allow the descendents of the clause process to continue with old information; when descendents commit then let unification filter out incompatible solutions from conjuncts as failure of the

---

[7] This is the same solution (called "Delayed Propagation") suggested by [Levy84].

clause.[7] On the surface, it seems that the only draw back to this scheme is late detection of inconsistent bindings. But this solution will only work if there are no read-only variables. With the introduction of read-only variables, this scheme may lead to deadlock. For example, if the axiom at the clause-process is

P (*a) <- Q (*a) & R (a?).

and the only axioms that are potentially unifiable with Q and R are

Q (2). and

R (2).

We would like the resulting computation to succeed with a=2. But when Q commits with a=2, the clause-process updates its copy of a and continues waiting for R to finish computing but by the definition of read-only variables, the goal-process solving R(a?) remains blocked waiting for the instantiation of a? before continuing.

Let the Parent Clause-Process Handle the Broadcasting

When a goal-process reports with new information, the clause-process can first update the global copy of the variables and then signal each of its descendents (except for the one that initiated this chain of actions) to update their copies. The problem is again one of deadlock. If two child processes report simultaneously, both would be blocked waiting for reply. The child that is serviced first is happy but when the clause-process tries to broadcast to the second child, both child and parent become blocked waiting for each other to reply (Figure 13 on page 41)

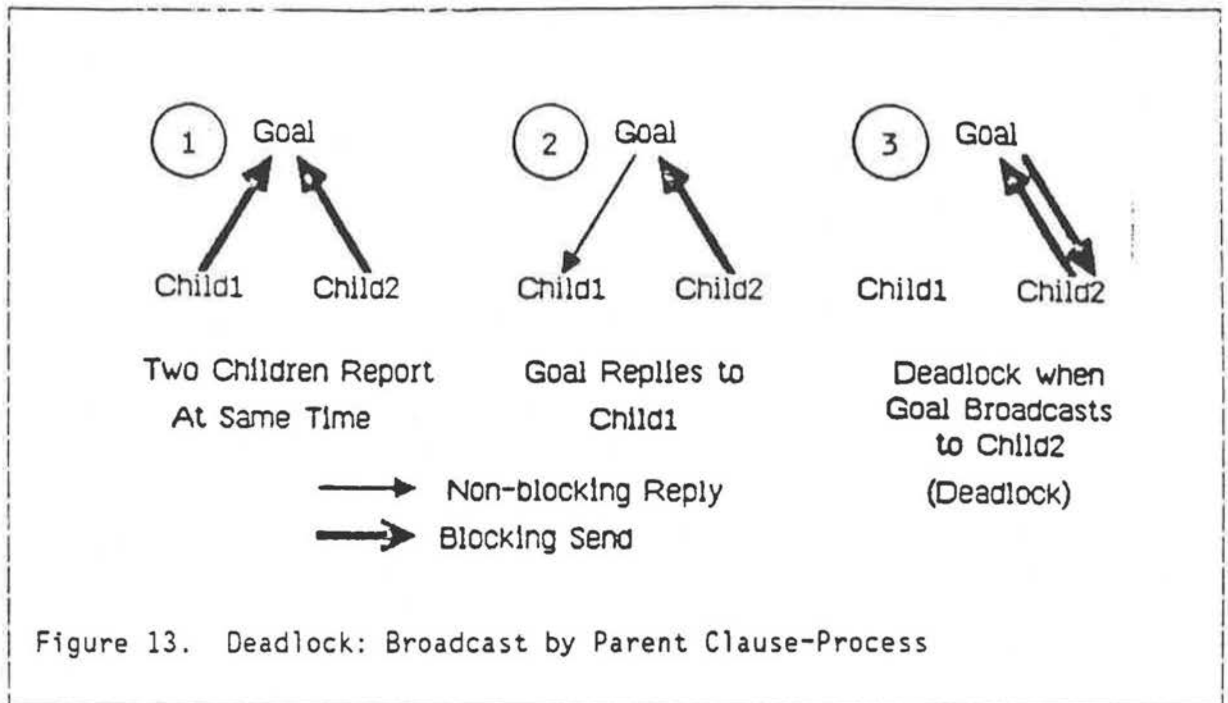


Figure 13. Deadlock: Broadcast by Parent Clause-Process

<u>Let the Reporting Goal-Process Handle the Broadcasting</u>

The clause-process may supply the reporting child with a list of children to signal. A similar situation as for the above scheme arises when two or more goal-processes try to broadcast to each other (Figure 14 on page 42). The problem is that information must be passed both ways: child processes must report upwards and broadcast information must travel downwards. This necessity violates the well-known rule-of-thumb (eg. [Lockhart79] [Deering82] in a Verex environment) that only child-processes should use the blocking send primitive and parent-processes should only use the non-blocking receive primitive to communicate; that is, "send" up the process tree - never down.

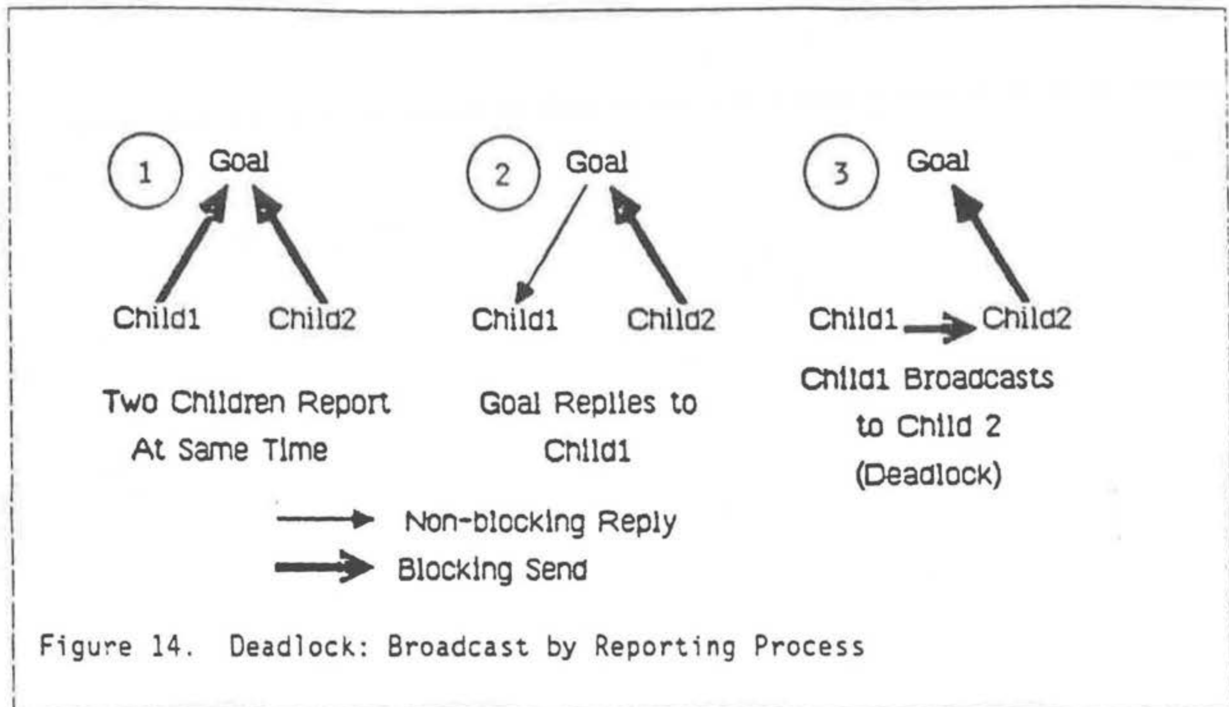


Figure 14. Deadlock: Broadcast by Reporting Process

Create a Broadcast Server

As explained in the previous scheme, the problem lies in messages being sent both up and down the process tree. What is needed is a mechanism in which the parent clause-process can initiate information down the tree without becoming blocked. By exploiting the Verex philosophy of creating multiple servers, there appears to be such a mechanism.

When a clause-process receives a report from a child, it creates a server process. The broadcast server is given a list of processes and an updated copy of the clause-process' axiom and is expected to broadcast to each of the given processes in turn. The clause-process remains free to handle other tasks.

Let's examine, again, the case of simultaneous reports. The clause-process receives one of the reports, updates its axiom, unblocks the reporting child and delegates the responsibility of broadcasting to a new server. It then is free to accept a subseouent report and create a second broadcast server (Figure 15 on page 43). The clause-process is not blocked waiting for broadcasting to complete and neither is the reporting process. Even if multiple broadcast servers exist at one time, no deadlock can result since no process issues a blocking send to these servers. It is the broadcaster that issues the send primitive and all its receivers are guaranteed to become unblocked in a bounded period of time (needed for a new server to be created).

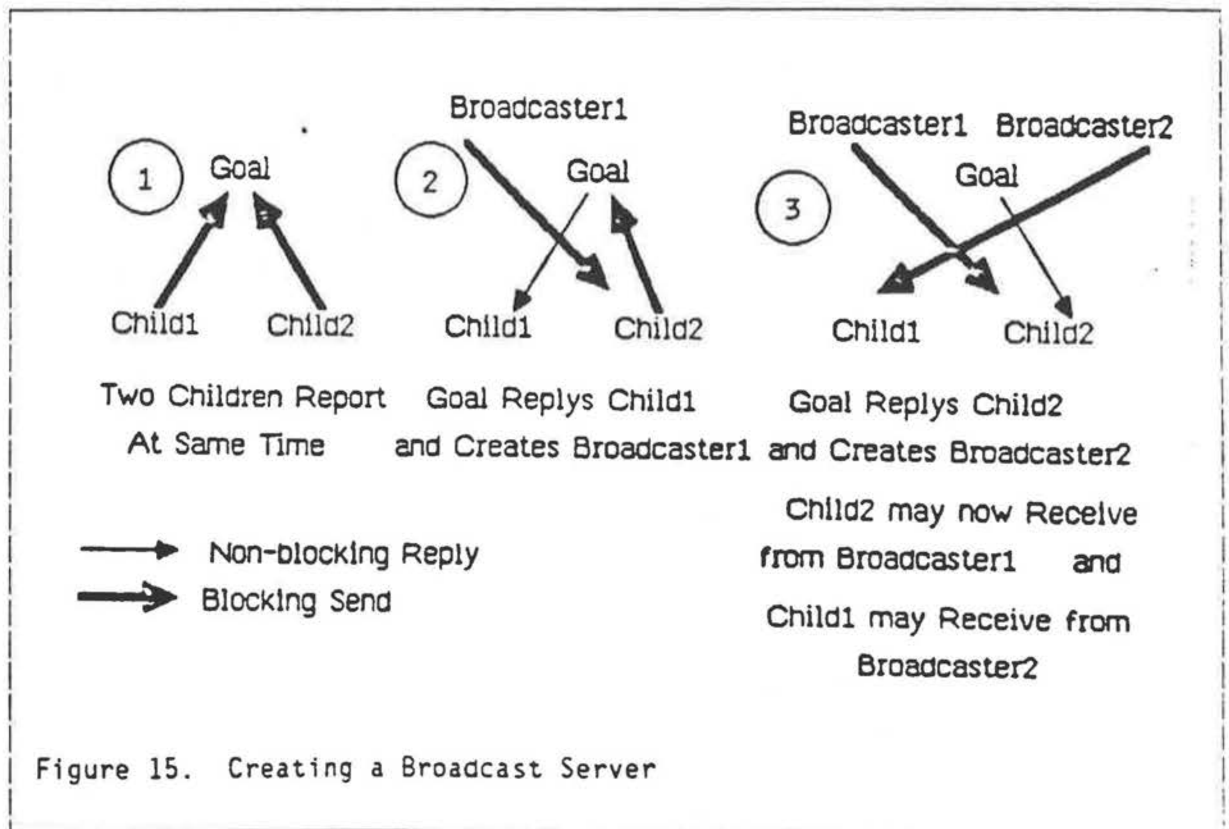


Figure 15. Creating a Broadcast Server

Propagation By Request

Yet another possible solution [Levy84] [Lee84] is to broadcast only to processes which have been suspended or have child-processes suspended waiting for the instantiation of a read-only variable.

Levy's scheme employs a queue of read-only variables. Whenever a process must wait for the instantiation of a variable, it first puts a request in the queue with the variable's identifier and its own process identifier and then suspends itself. When a process instantiates a read-only variable, it must "wake-up" all processes associated with the variable.

Lee suggests that a process should make a direct request via a "need-binding" message to its parent.

- If the parent's copy of the variable has been instantiated, it will allow the requesting process to continue with the new bindings.

- If the variable is unbound and the variable has the read-only annotation, then the parent process waits until one of its other committing child-processes instantiates the requested variable.

- If the variable is unbound and the variable is non-read-only, then the requesting process is allowed to continue but is required to poll the parent until the variable becomes instantiated.

- Finally, if the variable references a variable higher-up in the process tree then the parent is forced to issue a need-binding message of its own.

Either scheme will probably work ... providing enough book-keeping information is available. Consider the system of processes

```
1.  P ( *x, *y ) <- Q ( *x, *y ) & ....
2.  Q ( *z, *z ) <- R ( z? ) & ....
3.  R ( 1 ).
4.  <- P ( *x, *y ).
```

In trying to unify R (z?) with R (1), process 3 sends a need-binding message for z to its parent, process 2. But since z references both x and y, it now must send a need-binding message for instantiation of either of x or y. Continuing up a "degenerate" tree, the overhead in book-keeping will match that of using a broadcast server.

5.5 MUTUAL EXCLUSION CONSIDERATIONS

In the two previous sections,it was mentioned that a scratch copy of the axiom was made by the clause-process. The procedure was necessitated by the need to keep uncommitted values local to the trial process. A more general problem, when dealing with any multiprocessing system, is mutual exclusion - the problem of updating shared data.

Ignoring the problems of multiple address spaces, there is little need to duplicate the axioms database. Normally, there is no mutual exclusion problems associated with the database, since it is read-only. Additions are performed usually at the shell- or top- level when no clause- or goal- processes are present. But when we begin allowing Concurrent Prolog programs to generate new axioms, we would then need to introduce a database server or associate some mutual exclusion mechanism (eg. semaphores) with the axioms to synchronize the creation and reading of axioms. We will examine this topic further in a later section.

A more pressing problem is the mutual exclusion of the scratch-copy axioms between clause- and goal- processes. Consider the conjunction

P (*a) & Q (*a) being committed with the two axioms

P (*b) <- Assign (*b, 2). (note empty guard)

Q (*c) <- Assign (*c, 3). (Assume Assign instantiates the first variable with that of its second argument.)

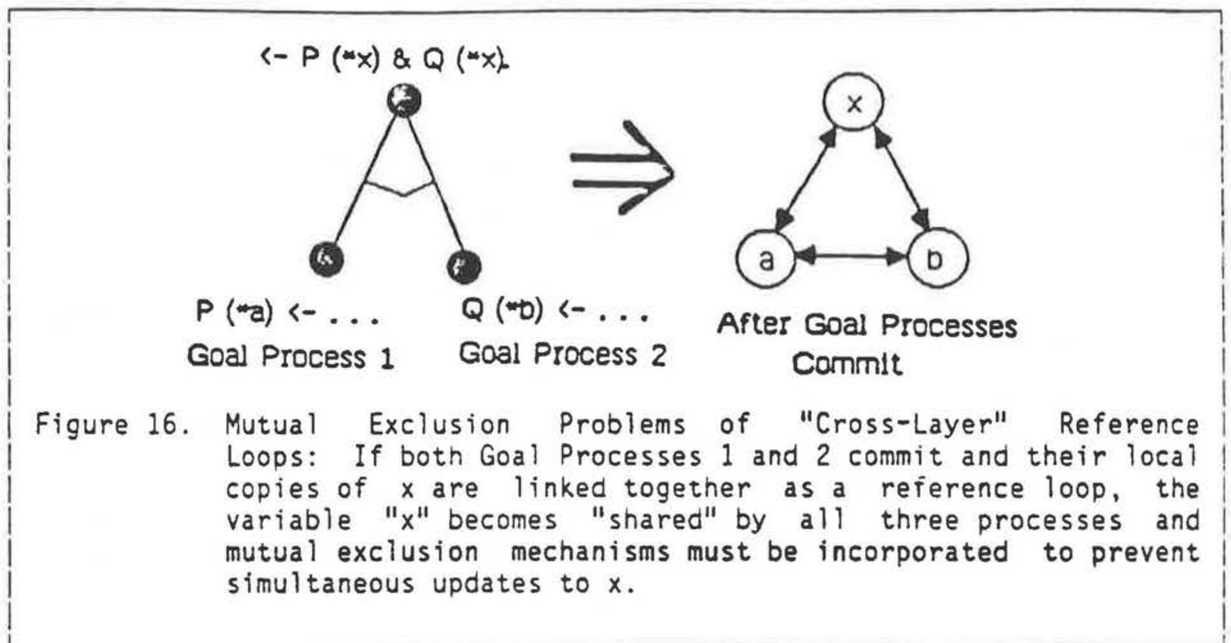


Figure 16. Mutual Exclusion Problems of "Cross-Layer" Reference Loops: If both Goal Processes 1 and 2 commit and their local copies of x are linked together as a reference loop, the variable "x" becomes "shared" by all three processes and mutual exclusion mechanisms must be incorporated to prevent simultaneous updates to x.

In theory, both (empty) guards are solved simultaneously. We can remove the obvious conflict by making the parent clause-process (at and-node) serialize the two committals to yield a reference loop of a, b, c. But by creating an explicit loop, we would then be vulnerable to concurrent updates of the loop by the two child "Assign" goal-processes (see Figure 16). One possible solution requires that we associate a semaphore with each loop and store its id in each cell of the loop. Every update <u>and</u> query must first secure the semaphore for this particular loop.

An alternative to such a complex mutual exclusion scheme is to eliminate the critical section (the loop) all-together by removing <u>interprocess links</u>. In the process of removing interprocess links, however, we must be careful not to remove <u>intraprocess links</u>. Consider the following:

1. P (*x, *y) <- Q (*x, *y) & R (*y).

2. Q (*x, *x).
3. R (1).
4. <- P (*x, *y).

We would like the result to be P (1,1). Hence, in unifying the local copy of Q (*x,*y) in axiom A1 with Q (*x,*x) in axiom A2, we must keep the intraprocess links between *x to *y in A1. We must use the mapping between original axioms and their duplicates (obtained from the copying mechanism) to reproduce any local-copy to local-copy references in the global copy (Figure 17). Only references that can be translated using the "copy-map" should be duplicated. Any other references should be ignored.

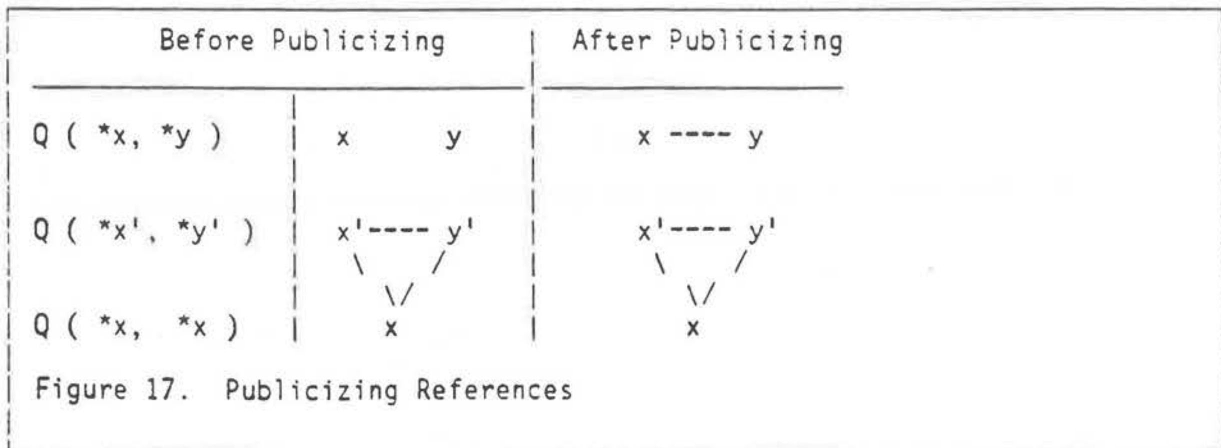


Figure 17. Publicizing References

5.6 READ-ONLY VARIABLES

Read-only variables have already been discussed in the implementation of broadcast servers and the avoidance of mutual exclusion conflicts. Here, we will try to tie up the loose ends of their implementation.

A process trying to unify an unbound read-only variable to some concrete value must first wait for the variable to be instantiated. To eliminate busy-waits, the process should somehow block itself until instantiation. An initial idea is to have a process (such a Name-Server) be responsible for all read-only variables. The owning process would send a request to the server to instantiate the variable. The server would flag the variable as being requested. When some other process instantiates the variable and recognizes the flag, it would send a message to the server which would then unblock the original requestor. The problem with this suggestion is that because of mutual exclusion considerations, cross-layer links have been removed. Unless a process shares and instantiates the requested variable, the variable can never be reached via a reference link and the requestor will wait indefinitely.

The solution currently implemented is to make use of new information as soon as it become available. When some other process instantiates a variable and commits to it, its parent process will make the value public via a broadcast server. The information will eventually filter down the tree to the process owning the read-only variable (which is now waiting only for a message from a broadcast server). If the message is not an instantiation of the read-only variable, then the process continues waiting after updating its copy of the axiom.

5.7 SUPPORTING DYNAMIC DATABASES

As mentioned in preceeding sections, the database of axioms will usually be static. Except for the initial loading of the axioms, we rarely require the power of axiom-creation and destruction since state information may be maintained via perpetual processes. Hence no mutual exclusion mechanisms were needed for the database to synchronize accesses from multiple processes.

Once in a while, however, we would like to exploit the fact that data may be used as part of the executable source code. For example, a professor may have access to a Prolog interpreter but is instructing a course on a form of logic which has a super-operator called "S" which is simply a composition of the normal logical operators. He requires an interpreter for a programming language called "S_Logic" which will accept normal Prolog source plus the "S" operator. One solution is to write the S_Logic interpreter on top of the Prolog interpreter. The S_Logic interpreter will need to read in an S_logic axiom, expand the axiom wherever it encounters the "S" operator, and then pass the new axiom to Prolog. The S_logic interpreter will necessarily create new Prolog axioms when it loads the S_logic database.

By allowing a program to dynamically create and destroy axioms, we now need to synchronize the queries on the database with the addition and deletion of axioms. In line with the Verex philosophy, we may create a Database Server to serialize these requests.

The server will be required to supply three major services in addition to routine maintenance of the database:

1. Find a list of axioms whose head is potentially unifiable with a given predicate.
2. Add an axiom to the database.
3. Delete a specified axiom from the database. (eg. Delete the first axiom whose head is potentially unifiable with a given predicate).

A process requiring one of these services will request it via a message to the database server. The client process will then block itself until the request has been fulfilled.

Note that service #1 now returns all of the axioms that are potentially unifiable instead of only the first axiom of the list. The server should actually create a copy of these axioms to prevent another client from making additions or deletions while the first client is manipulating the list. That is, a process querying the database should work with a "snapshot" of the database taken at the time of query. This is consistent with the view that an Or-node "creates" all its descendants in parallel.

6.0 EVALUATION AND CONCLUSIONS

We have successfully implemented a "first-attempt" interpreter has been successfully implemented on the (multi-user) Verex Operating System running on a TI/990. The interpreter is an experimental one and lacks the power and elegance of those from Edinburgh [Kowalski74] [Warren77], Waterloo [Lee84], and Vancouver [Goebel80]. It lacks almost all of the necessary "built-in" predicates (eg.math. predicates) of a production interpreter and only has syntax resembling that of Lisp (Figure 18 on page 54). In addition, a look at the performance characteristics (eg. Figure 19 on page 55) indicates that it is still premature to consider using the current system to support any substantial piece of Concurrent Prolog program: the maximum number of processes that may be active at one time is too low; there is currently no garbage collection of processor memory (except for the processor stacks) when a process fails[8], and the execution speed is slow.

---

[8] Garbage collection was omitted from the current implementation because it is considered as part of the normal cycle of resource reclaimation when a process is destroyed.

```
<identifier>         ::= a sequence of 1 to 30 characters from
                         the set [ 'A'..'Z', 'a'..'z', '0'..'9' ]

<atom>               ::= <identifier>

<normal variable>    ::= *<identifier>

<readonly variable>  ::= ?<identifier>

<variable>           ::= <normal variable> | <readonly variable>

<variable list>      ::= <variable> | <variable> <variable list>

<function>           ::= ( <atom> [ <variable list> ] )

<argument>           ::= <atom> | <variable> | <function>

<arg list>           ::= <argument> | <argument> <arg list>

<predicate name>     ::= <atom>

<term>               ::= ( <predicate name> [ <arg list> ] )

<term list>          ::= <term> | <term> <term list>

<guard>              ::= ( GUARD <term list > )

<head>               ::= <atom>

<body>               ::= <term list>

<axiom>              ::= ( :- <head> [ <guard> ] <body> ). |
                         ( <head> ).

<query>              ::= ( :- <head> ).

examples

(append nil *12 *12).
( :- (append ( cons *h *t ) *12 ( cons *h *rest ) )
     (append *t              *12 *rest              )
)
```

Figure 18. BNF description of Syntax Accepted by Current Interpreter

Database of Axioms

```
Merge ( *xs, nil, *xs ).

Merge ( nil, *ys, *ys ).

Merge ( *x.*xs, *ys, *x.*zs ) <- Merge ( xs?, *ys, *zs ).

Merge ( *xs, *y.*ys, *y.*zs ) <- Merge ( *xs, ys?, *zs ).
```

Top-Level Goal

```
<- Merge ( a1.b1.c1.nil, a2.b2.c2.nil, *z ).
```

Performance Measurements

Fifteen invocations of the top-level goal required about

430 resolutions

27 seconds of real time

under light-load conditions.

This translates roughly to 23 resolutions per second.

A trace of the execution of the program later showed that over fifty percent of the CPU time was spent in message passing.

Figure 19. Performance Measurement on a Bounded-Wait Merge

6.1.1 Possible Optimizations

Noting that over fifty percent of the execution time is spent in process communication,[9] the following optimization is suggested:

> In the current system, a process that commits to a path must broadcast (via its parent process and a broadcast server) to all sibling processes which will, in turn, broadcast to all their descendants. If we examine this procedure, we find that only the parent need be informed; the remaining processes need only be informed if the committing process has changed a variable which will influence their outcome. Recalling cross-layer reference loops were removed due to mutual-exclusion considerations, the optimization simplifies to the following rule:
>
> 1. When a process commits, it always informs its parent process.
> 2. The parent process tries to unify the new variables with its copy of the axiom.
> 3. If the unification fails, the committing process is deleted as before.
> 4. If the unification succeeds and one or more variables of the parent process was instantiated then the parent process creates a broadcast server as before. If, however, none of the variables changed then broadcasting is complete.

---

[9] One should keep in mind that portions of this time are used to invoke various servers of the operating system (eg. process creation and destruction, memory allocation, file system services).

[10] The current version of the Verex Operating System supports only about 30 processes.

We can apply a similar optimization to relieve the problem of a limited number of available processes.[10] Recall that when a process resides on the solution path, it cannot be destroyed since it may contain variables in its address space which contribute to the solution. Destruction of the process will result in the destruction of its local memory space. We are free to destroy processes which do not contribute to the solution. So when a parent process successfully unifies the axiom from a reporting child process with its own axiom and determines that no new information was gained, it may release that subtree of processes to be used by some other part of the solution tree. This solution, however, is only a temporary measure since the described situation is not too common. What is needed is an increase in the number of allowable processes in the operating system.

6.1.2 Summary

Despite the interpreter's current limitations, it has allowed us to examine the implications of the constructs of Concurrent Prolog in terms of system requirements and has demonstrated, to a certain extent, the feasibility of implementing a distributed interpreter for the language. From this work, several points came to light:

- The semantics of the language can be made cleaner with a minor modification to the behaviour of the interpreter when attempting to unify an unbound read-only variable with an unbound non-read-only variable.

- New bindings should be propagated throughout the tree as soon as they become available instead of delaying them. Delayed propagation [Levy84] is possible only if read-only variables are treated outside of the normal scheme.

- Due to the above desire to instantiate read-only variables at the earliest stage, reference loops were introduced in preference over the normal simply-linked list implementation.

- To minimize the possibility of deadlock, a broadcast server should be created to propagate new bindings to child-processes. The use of an extra server will server the dependency loop between parent and child processes.

- In order to avoid child processes from affecting each other, each child clause process must make a scratch-pad copy of the axiom. This procedure will necessarily entail recursively copying the subexpressions bound to variables. A mapping between the original subexpressions to the scratch-pad copies will aid in the copying process.

- To avoid deadlock and mutual exclusion problems, interprocess pointers (or tags [Lee84]) should not be used to form reference-loops. But in eliminating these references, one must be sure to duplicate intraprocess references. That is, if a child process creates a reference between two variables in its local copy and later commits, this reference must be duplicated in the parent's copy. The mapping from the copying mechanism will be useful in determining whether the reference is interprocess or intraprocess.

Future work on this interpreter should include a re-implementation using separate address-spaces for each process (i.e. different teams) and optimizations such as a Copy-on-demand [Levy84] scheme to reduce the number of subexpressions that are duplicated. Hopefully further examination of the implementation obstacles will open up the full potential of the language and clarify its desired semantics.

7.0 REFERENCES

[Aho77] Aho, A.V., and Ullman, J.D. Principles of Compiler Design Addison-Wesley, 1977

[Boyle82] Boyle, P.D. The Design of a Distributed Kernel for a Multiprocessor System M.Sc. Thesis, University of British Columbia, Vancouver, 1982

[Boyer,Moore72] Boyer, R.S. and Moore, J.S. The Sharing of Structure in Theorem Proving Programs, Machine Intelligence 7 (ed. Meltzer & Michie), Edinburgh UP. 1972.

[Brinch Hansen75] Brinch Hansen, P. The Programming Language Concurrent Pascal IEEE Transactions on Software Engineering SE-1(2):199-207, 1975

[Cheriton79.1] Cheriton. D.R. Multi-process Structuring and the Thoth Operating System Technical Report 79-5, University of British Columbia, Vancouver, 1979 (Reprint of the author's Ph.D. thesis from the University of Waterloo)

[Cheriton79.2] Cheriton. D.R. Interactive Verex Technical Report 79-1, University of British Columbia, Vancouver, 1979

[Cheriton79.3] Cheriton, D.R., and Steeves, P.J. The Zed Reference Manual Technical Report 79-2, University of British Columbia, Vancouver, 1979

[Cheriton81] Cheriton, D.R. Distributed I/O Using an Object-Based Protocol Technical Report 81-1, University of British Columbia, Vancouver, 1981

[Deering82] Deering, S.E. Multi-Process Structuring of X.25 Software Technical Report 82-11, University of British Columbia, Vancouver, 1982

[Dijkstra76] Dijkstra, E.W. A Discipline of Programming Prentice-Hall, 1976

[Goebel80] Goebel, R. PROLOG/MTS Users' Manual Technical Manual 80-25, University of British Columbia, Vancouver, 1980

[Kowalski74] Kowalski, R.A. Logic For Problem Solving DCL Memo 75, Dept of A.I., Edinburgh, 1974

[Kusalik84] Kusalik, A.J. Bounded-Wait Merge in Shapiro's Concurrent Prolog New Generation Computing, 2 (1984), Ohmsha and Springer-Verlag, 1984

[Lee84] Lee, R.K.S. Concurrent Prolog in a Multi-Process Environment Institute for Computer Research Report 24,

University of Waterloo, September 1984 (Reprint of the author's M.Sc. thesis at the University of Waterloo)

[Levy84] Levy, J. A Unification Algorithm for Concurrent Prolog Proceedings of the Second International Logic Programming Conference, Uppsola, Sweden, July 1984

[Lockhart79] Lockhart, T.W. The Design of A Verifiable Operating System Kernel Technical Report 79-15, University of British Columbia, Vancouver, 1979

[Nilsson80] Nilsson, N.J. Principles of Artificial Intelligence Tioga, 1980

[Reiter79] Reiter, R., On Close World DataBases, (1979) Readings in Artificial Intelligence (ed. Webb & Nilsson), Tioga, 1981

[Robinson65] Robinson, J.A. A Machine Oriented Logic Based on the Resolution Principle JACM 12, 1965.

[Shapiro83] Shapiro, E.Y. A Subset of Concurrent Prolog and Its Interpreter Technical Report TR-003, ICOT-Institute for New Generation Computer Technology, 1983.

[Warren77] Warren, D.H.D. Implementing Prolog - Compiling Predicate Logic Programs DAI Research Reports 39,40, University of Edinburgh, 1977