ICOT Technical Report: TR-003

TR-003

A Subset of Concurrent Prolog and Its Interpreter

by

Ehud Y. Shapiro
(Weizmann Institute of Science)

February, 1983

A Subset of Concurrent Prolog and Its Interpreter

February, 1983

Ehud Y. Shapiro
The Weizmann Institute of Science
Rehovot 76100, ISRAEL

This is a revised version of
Technical Report TR-003
ICOT — Institute for New Generation Computer Technology
Mita Kokusai Bldg. 21F, 4-28 Mita 1-Chome
Minato-ku, Tokyo 108, JAPAN

Table of Contents

Programs

Figures

A Subset of Concurrent Prolog and Its Interpreter

Ehud Y. Shapiro
Department of Applied Mathematics
The Weizmann Institute of Science
Rehovot 76100, ISRAEL

Abstract

Concurrent Prolog is a variant of the programming language Prolog, which is intended to support concurrent programming and parallel execution. The language incorporates guarded-command indeterminacy, dataflow-like synchronization, and a commitment mechanism similar to nested transactions.

This paper reports on a subset of Concurrent Prolog, for which we have developed a working interpreter. It demonstrates expressive power of the language via Concurrent Prolog programs that solve benchmark concurrent programming problems. It describes in full detail an interpreter for the language, written in Prolog, which can execute these programs.

CR Categories:

Key words and phrases: logic programming, concurrent programming, object-or$\overset{e}{\text{in}}$ted programming, operating systems.

1. Introduction

Due to its expressive power, simple semantics, and amenability to efficient implementation, Prolog is a promising language for a large class of applications. Prolog also has another, as yet unexploited, aspect: it is a sequential simulation of a parallel computation model.

There are two reasons for exploiting Prolog's underlying parallelism. One is to improve the performance of Prolog in some of its current applications, perhaps using novel computer architectures. The other is to incorporate, in the range of Prolog, applications that require concurrency. Concurrent Prolog is concerned with both.

Concurrent Prolog is currently under design by the author. This paper reports on a subset of the language for which we have developed a working interpreter. This subset is a variant of sequential Prolog, but does not contain sequential Prolog properly.

When Japan's Fifth Generation Computers Project chose logic programming as its basic framework, there was some concern it would have difficulties integrating ideas and techniques of object-oriented programming. Our example programs show that Concurrent Prolog is an object-oriented programming language *par excellence*.

The synchronization mechanism of Concurrent Prolog — read-only variables — can be viewed as a generalization of dataflow synchronization [1] from functional to relational languages. It is possible that Concurrent Prolog may be a suitable programming language for dataflow computers [43, 48].

One need not wait, however, until dataflow computers become available. Our experience with implementing Concurrent Prolog, and the example programs below, suggest that this language may be a practical programming language for implementing operating system functions even on today's computers.

Two controversial features of sequential Prolog, namely the *cut* and side-effects, are cleaned-up in Concurrent Prolog. Concurrent Prolog's *commit* operator [11] achieves an effect similar to *cut* in increasing the efficiency of the program, but has a cleaner semantics due to its symmetry, much the same way as Dijkstra's guarded-command [15] has a cleaner semantics than the conventional if-then-else construct. Concurrent Prolog eschews the use of side-effects such as *assert* and *retract* to implement global data structures, since they can be implemented by perpetual processes, executing side-effect free programs.

The paper is organized as follows. Section 2 introduces logic programs and sequential Prolog. Section 3 defines the subset of Concurrent Prolog used in this paper, henceforth referred to as Concurrent Prolog, and sketches a distributed interpretation algorithm for it.

Since the programming style and concepts of Concurrent Prolog are rather different — both from sequential Prolog and from other concurrent programming languages — we suspect that the only way to gain a true understanding of the language is to study, or to develop, Concurrent Prolog solutions to nontrivial concurrent programming problems. Section 4 includes in full detail several Concurrent Prolog solutions to many such benchmark problems, including:

- a concurrent implementation of the quicksort algorithm,
- a recursive concurrent program for numbering the leaves of a tree,
- an airline reservation program,
- programs for merging streams using various scheduling strategies,

- a simple message-sending operating system,
- fragments of a Unix-like shell,
- shared FIFO and priority queues,
- a simulator of a multiprocessor Concurrent Prolog machine,
- an priority queue based multiple-printers spooler,
- an implementation of the SCAN disk-arm scheduling algorithm,
- an implementation of a concurrent algorithm for finding the connected components of a graph.

The programs are concise, elegant, and, once the Concurrent Prolog programming style is grasped, are also easy to understand.

Instead of studying someone else's programs, one is better off trying to develop some of his or her own. To facilitate such an endeavor, Section 5 describes a centralized Concurrent Prolog machine, and its implementation in Prolog-10. The implementation is 44 lines of code long, and performs about 135 process reductions per CPU second (LIPS) on a DEC-2060. The initiated reader, who has an access to a Prolog implementation, is invited to type this interpreter in and gain some first-hand experience in programming in Concurrent Prolog.

Section 6 compares Concurrent Prolog with other concurrent programming languages, including the relational language of Clark and Gregory [11], by which we were strongly influenced.

Section 7 compares Concurrent Prolog with sequential Prolog, discusses deficiencies in the subset of Concurrent Prolog described in the paper, and explore possible extensions to it.

Section 8 explores future research directions relating to concurrent logic programming.

The listings of the full Concurrent Prolog interpreter, which contains a trace and statistics package, together with some utility programs, are included as an appendix.

2. Logic programs and sequential Prolog

2.1. The logic programs computation model

Both sequential and concurrent Prolog are approximations to the computation model called logic programs. A *logic program* is a set of universally quantified first-order axioms of the form

$$A \leftarrow B_1, B_2, ..., B_n$$

where the A and the B's are atomic formulae, also called *atomic goals.* Such a clause reads "A if B_1 and B_2 and ... and B_n". A is called the clause's *head* and the B's are called its *body*.

A computation of a logic program amounts to the construction of a proof of an existentially quantified conjunctive goal from the axioms. It can have two results: success or failure. If the computation succeeds, then the values found for the variables in the initial goal constitute the output of the computation. A goal can have several successful computations, each resulting in a different output.

The computation progresses via nondeterministic goal reduction: at each step it has a current goal $A_1, A_2, ..., A_n$; it arbitrarily chooses a goal A_i, for some $1 \leq i \leq n$; it then nondeterministically chooses a clause $A' \leftarrow B_1, B_2, ..., B_k$, $k \geq 0$, for which A and A' are unifiable via a substitution θ, and uses this clause to reduce the goal. The reduced goal is $(A_1,..., A_{i-1}, B_1,..., B_k, A_{i+1},..., A_n)\theta$. The computation terminates when the current goal is empty.

This description readily suggests two forms of parallel execution: the reduction of several goals in parallel, also called *and-parallelism*, and a concurrent search of the computation paths resulting from different nondeterministic choices of the unifiable clause, also called *or-parallelism*.

Different orderings of the goals to be reduced need not be investigated since they are immaterial to the result of the computation [2]. However, the chosen ordering can greatly affect the degree of nondeterminism in the computation. Hence the major concern of a practical logic programming language it to provide the programmer with control facilities with which he can reduce this degree of nondeterminism. Typically, such facilities enable the programmer to influence both the order in which goals are reduced, and the clauses they are reduced with.

2.2. An example of a logic program

An example of a logic program that implements a variant of the quicksort algorithm is shown in Figure 1.

The program is adapted from the Prolog-10 manual [4], and so are our notational conventions: Variables begin with an upper-case letter, all other symbols with lower-case letters. The binary term [X|Y] (read "X *cons* Y") denotes the list whose head (*car*) is X and tail (*cdr*) is Y. The term [X, Y|Z] is a shorthand for [X|[Y|Z]] — the list whose *car* is X, *cadr* is Y, and *cddr* is Z. The constant [] (read "nil") denotes the empty list.

```
quicksort(Unsorted, Sorted) :-
    qsort(Unsorted, Sorted-[]).

qsort([X|Unsorted], Sorted-Rest) :-
    partition(Unsorted, X, Smaller, Larger),
    qsort(Smaller, Sorted-[X|Sorted1]),
    qsort(Larger, Sorted1-Rest).
qsort([], Rest-Rest).

partition([X|Xs], A, Smaller, [X|Larger]) :-
    A<X, partition(Xs, A, Smaller, Larger).
partition([X|Xs], A, [X|Smaller], Larger) :-
    A≥X, partition(Xs, A, Smaller, Larger).
partition([], _, [], []).
```

Figure 1: A logic program for quicksort

Underscore "_" stands for an anonymous variable that occurs only once, and hence does not deserve a name.

The best way to understand — and to document — a logic program is to state the relations it computes. The procedure *quicksort(X, Y)* computes the relation "sorting X gives Y" (or, "Y is an ordered permutation of X"). The procedure *qsort(X, Y)* computes the relation "the difference list Y is an ordered permutation of the list X". *partition(X, Y, Z, W)* computes the relation "the list Z contains all elements of the list X less than or equal to Y, in the order they appear in Y, and W contains all elements of X greater than Y, in the order they appear in X".

The first clause of *partition* reads, using Lisp jargon, "partitioning a list whose *car* is X and *cdr* is Xs according to element A gives the lists *Smaller* and X *cons Larger*, if A is less than X and partitioning Xs according to A gives the lists *Smaller* and *Larger*". Other clauses are read similarly.

2.3. Difference-lists: an example of a logic programming technique

The quicksort program illustrates an important logic programming technique, used throughout this paper, called *difference-lists* [12]. A difference-list represents a list L as the difference between two lists X and Y. As a notational convention, the term $X{-}Y$ is used. But since logic programs do not evaluate logical terms, only unify them, the name of the binary functor representing a difference-list can be arbitrary, as long as it is used consistently.

Difference-lists increase both the efficiency and brevity of logic programs. They increase their efficiency since, in some cases, two difference lists can be concatenated in constant time, and without copying data structures ("consing"). A difference list $X1-X2$ is *compatible* with $Y1-Y2$ if $X2=Y1$. Compatible difference lists can be concatenated using the following single-clause logic program:

concatenate$(X-Y, Y-Z, X-Z)$.

concatenate(X, Y, Z) computes the relation "concatenating the difference list X to the difference list Y is the difference list Z if X and Y are compatible".

One way to ensure that the two difference lists $X1-X2$ and $Y1-Y2$ are compatible is to keep the value of X2 undetermined. In such a case the *concatenate* call is executed in constant time, independent of the length of the lists. For example, the result of the call *concatenate*$([a, b, c|X]-X, [1, 2]-[], Y)$ is $X=[1, 2]$ and $Y=[a, b, c, 1, 2]-[]$.

Difference-lists increase the brevity of logic programs by eliminating the need to call *concatenate* explicitly, as in the quicksort example. The unification of the tail of the first list with the head of the second is done implicitly, by calling them with the same name, *Sorted*1, and the construction of the concatenated list is then immediate. Hence one would typically find neither the code of nor a call to the concatenate procedure in a logic program that uses difference lists.

Difference-lists are the logic-programming counterpart of Lisp's *rplcd* [36], which is also used to concatenate lists in constant time and save "consing". There is a difference between the two: the former are side-effect free, and can be discussed in terms of the abstract computation model, whereas *rplcd* is a destructive operation, which can be described only by reference to the machine representation of S-expressions.

2.4. An example of a computation

A successful computation of the *quicksort* program on the goal *quicksort*([2, 1, 3], *X*) may proceed as follows. First the goal is reduced to *quicksort*([2, 1, 3], *X*−[]) using the only clause of *quicksort*. Then the recursive clause of *qsort* is invoked; this is the only applicable clause, since the head of the second clause does not unify with the goal. The reduce goal is:

partition([1, 3], 2, *Smaller*, *Larger*),
quicksort(*Smaller*, *Sorted*−[2|*Sorted*1]),
quicksort(*Larger*, *Sorted*1−[]).

To solve *partition*([1, 3], 2, *Smaller*, *Larger*) we nondeterministically choose *partitions*'s second clause, and reduce it to $2 \geq 1$, *partition*([3], 2, *Smaller*1, *Larger*), while unifying *Smaller* with [1|*Smaller*1]. The $\geq$ test is solved immediately, assuming that arithmetic is represented as a large table (simulated by the machine arithmetic operations, or by a suitable logic program). The recursive call to *partition* is solved by two more reductions, at the end of which *Smaller*1 is unified with [] and *Larger* with [3]. We next turn to the two recursive calls to *qsort*1, which are now instantiated to

qsort([1], *Sorted*—[2|*Sorted*1]), *qsort*([3], *Sorted*1—[]).

The correct nondeterministic choices result in unifying *Sorted* with [1, 2, 3] and *Sorted*1 with [3], and the computation succeeds, with the variable *Sorted* in the initial goal unified with [1, 2, 3].

2.5. Controlling nondeterminism at the expense of completeness

In the computation of a logic program the order in which subgoals are solved is immaterial, as long as the correct nondeterministic choices are made. However, a careless ordering in the example computation above may require choosing the values of X and Y when solving $X \geq Y$, a choice that has a rather large degree of nondeterminism.

To avoid making such hopeless choices, practical logic programming languages enable the programer to control the computation, usually at the expense of the completeness of the resulting proof procedure. This incompleteness is not a "bug" in the design of a logic programming language, but a conscious design decision, whose motivation follows.

A logic program can be executed in several ways, many of which are curious, but of little practical value. For example, the *quicksort* logic program can be run backwards, and generate all permutations of a list; when invoked with its two arguments undetermined, this program can be used to generate all pairs of lists of integers such that one is an ordered permutation of the other.

For the sake of efficiency, practical logic programming languages give up the ability to make such obscure uses of logic programs. Hence we refer to them as approximations to the logic programming computation model. Unfortunately, there are not yet any mathematically elegant characterizations of these approximations; all that can be said currently is that a logic program, using an incomplete proof procedure, computes only a subset of its logical consequences. The precise subset of computable consequences can be determined only by reference to the operational semantics of the programming language.

This is in contrast with the abstract logic programs computation model, which has al least two other independent characterizations. Van Emden and Kowalski [17] show that the smallest interpretation in which a logic program is true equals the subset of its Herbrand universe on which it succeeds. They also associate a transformation with any logic program, and show that its least fixpoint is equals its smallest interpretation. Hence we refer to the smallest interpretation in which a logic program is true as *the interpretation* of the program. The price a practical logic programming language pays for controlling non-determinism is that its programs typically compute only a subset of their associated interpretations.

2.6. Sequential Prolog

Sequential Prolog is an example of an approximation to the logic programs computation model, especially designed for efficient execution on a von Neumann machine. Sequential Prolog uses the order of goals in a clause and the order of clauses in the program to control the search for a proof. In sequential Prolog the chosen goal is always the leftmost goal, and the nondeterministic choice of the unifiable clause is simulated by sequential search and backtracking. Given a goal A_1, A_2, ..., A_n and a program P, Prolog sequentially searches for the first clause in P whose head unifies with A_1, and reduces the goal using this clause. It then tries, in order from left to right, to solve the reduced goal, accumulating the bindings of variables as it goes along. If the Prolog interpreter ever fails to solve a goal then it backtracks to the last choice of a clause made, resets the bindings made since that choice, and tries the next unifiable clause. If no choices left the computation fails.

In addition to text order, sequential Prolog uses the *cut* symbol "!" to control its execution. A cut is inserted in a clause as a goal, and when used decently can be ignored in the declarative reading of a clause. Operationally, a cut commits the interpreter to the current execution path and to all choices made since, and including, the choice of a clause in which the cut occurs.

Even though it originated in a rather peculiar computation model, sequential Prolog exhibits some resemblance to conventional sequential programming languages, as summarized in Figure 1. This resemblance is partially responsible for our understanding of how to implement Prolog efficiently on a von Neumann machine [45].

Procedure:	List of definite clauses with the same head predicate
Procedure call:	Goal
Binding mechanism data selection and construction:	Unification
Execution mechanism:	Nondeterministic goal reduction, simulated by sequential search and backtracking

Figure 2: Concepts of Sequential Prolog

3. Concurrent Prolog

3.1. Basic concepts and syntax

While comparing several concurrent programming languages, Bryant and Dennis wrote [9]:

"Several issues must be considered when designing programming languages to support concurrent computation. Of primary importance is expressive power. The expressive power of a language, in the context of concurrent systems, means the form of concurrent operations, the type of communication, synchronization, and nondeterminacy which can be expressed in the language. A programming language which lacks expressive power will force the programmer to rely on a suitable set of operating system routines to implement desired behavior. A properly designed language, on the other hand, should have sufficient richness to express these functions directly."

We claim that the computational model of logic programs embodies all the mechanisms necessary for a concurrent programming language — *concurrency, communication, synchronization, and indeterminacy*. All one needs to do is uncover them.

A system of processes corresponds to a conjunctive goal, and a unit goal to a process. The state of a system is the union of the states of its processes, where the state of a process is the value of its arguments. *And*-parallelism — solving several goals simultaneously — provides the system with concurrency. *Or*-parallelism — attempting to solve a goal in several ways simultaneously — provides each process with the ability to perform indeterminate actions. Variables shared between goals serve as the process communication mechanism; and the synchronization of processes in a system is done by denoting which processes can "write" on a shared variable, i.e. unify it with a nonvariable term, and which processes can only "read" the content of a shared variable

X, i.e. can unify X with a nonvariable term T only after X's principal functor is determined, possibly by another process. This analogy is incorporated in Concurrent Prolog, and is summarized in Figure 3.

System:	Conjunctive goal
Process:	Unit goal
Process state:	Values of arguments
Process computation:	Indeterminate process reduction
Process communication:	Unification of shared variables
Process synchronization:	Suspending instantiation of undetermined "read-only" variables
Process failure:	Goal finite-failure

Figure 3: Concepts of Concurrent Prolog

Concurrent Prolog adds two syntactic constructs to logic programs. *Read-only* annotation of variables, X?, and the *commit* operator "|". Both are used to control the computation, i.e the construction of a proof, by restricting the order in which goals can be reduced, and restricting the choice of clauses that can be used to reduce them.

A *Concurrent Prolog program* is a finite set of guarded clauses. A *guarded clause* is a universally quantified axiom of the form

$$A :- G_1, G_1, ..., G_m \mid B_1, B_2, ..., B_m, \quad m, n \geq 0$$

where the G's and the B's are atomic goals. The G's are called the *guard* of the clause and B's are called its *body*. When the guard is empty the commit operator is omitted. The clause may contain variables marked "read-only".

The commit operator generalizes and cleans sequential Prolog's cut. Declaratively, it reads like a conjunction: A is implied by the the G's *and* the B's. Operationally, a guarded clause functions similarly to an alternative in a guarded-command [15]. It can be used to reduce process $A1$ to a system B if A is unifiable with $A1$ and, following the unification, the system G is invoked and terminates successfully.

Program 1 below is a concurrent Prolog implementation of quicksort. It differs from the logic program in Figure 1 in the read-only annotations that occur in the recursive calls to *qsort* and *partition*, and in the commit operator in the recursive clauses of *partition*.

```
(0) quicksort(Unsorted, Sorted) :-
        qsort(Unsorted, Sorted-[]).

(1) qsort([X|Unsorted], Sorted-Rest) :-
        partition(Unsorted?, X, Smaller, Larger),
        qsort(Smaller?, Sorted-[X|Sorted1]),
        qsort(Larger?, Sorted1-Rest).
(2) qsort([], Rest-Rest).

(1) partition([X|Xs], A, Smaller, [X|Larger]) :-
        A<X | partition(Xs?, A, Smaller, Larger).
(2) partition([X|Xs], A, [X|Smaller], Larger) :-
        A≥X | partition(Xs?, A, Smaller, Larger).
(3) partition([], _, [], []).
```

Program 1: A Concurrent Prolog implementation of quicksort

The unification of terms containing read-only variables is an extention to normal unification [39]. The unification of a read-only term X? with a term Y is defined as follows. If Y is non-variable then the unification succeeds only if X is non-variable, and X and Y are recursively unifiable. If Y is a variable then the unification of X? and Y succeeds, and the result is a read-only variable. The symmetric algorithm applies to X and Y?.

In implementation terms, we represent the read-only annotation as a unary functor written in postfix notation, and augment Prolog's unification algorithm to handle this term specially. The code that implements this extended unification algorithm is shown in Program 19, Section 5.

This definition of unification implies that being "read-only" is not an inherited property, i.e. variables that occur in a read-only term are not necessarily read-only. Stating it differently, the scope of a read-only annotation is only the principal functor of a term, but not its arguments. This design decision provides Concurrent Prolog with a unique and powerful object-oriented programming technique, called *partially determined messages*, used and explained in several of the example programs below.

The definition of unification also implies that the success of a unification may be time-dependant: a unification that fails now, due to violation of a read-only constraint, may succeed later, after the principal functor of a shared read-only variable is determined by another process, in which this variable does not occur as read-only.

3.2. A sketch of a distributed Concurrent Prolog interpreter

The execution of a Concurrent Prolog system S, running a program P, can be described informally as follows. Each process A in S tries asynchronously to reduce itself to other processes, using the clauses in P. A process A can reduce itself by finding a clause $A1$:– $G \mid B$ whose head $A1$ unifies with A and whose guard system G terminates following that unification. The system S terminates when it is empty. It may become empty only if some of the clauses in P have empty bodies.

The computation of a Concurrent Prolog program gives rise to a hierarchy of systems. Each process may invoke several guard systems, in an attempt to find a reducing clause, and the computation of these guard systems in turn may invoke other systems. The communication between these systems is governed by the commitment mechanism. Subsystems spawned by a process A have access only to variables that occur in A. As long as a process A does not commit to a reducing clause, these subsystems can access only read-only variables in A, and all bindings they compute to variables in A which are not read-only are recorded on privately stored copies of these variables, which is not accessible outside of that subsystem. Upon commitment to a clause $A1$:– $G \mid B$, the private copies of variables associated with this clause are unified against their public counterparts, and if the unification succeeds the body system B of the chosen clause replaces A.

A more detailed description of a distributed Concurrent Prolog interpreter uses three kinds of processes: an and-dispatcher, an or-dispatcher, and a unifier; these processes should not be confused with the Concurrent Prolog processes themselves, which are unit goals.

The computation begins with a system S of Concurrent Prolog processes, and progresses via indeterminate process reduction. After an and-dispatcher is invoked with S, the computation proceeds as follows:

- An *and-dispatcher*, invoked with a system S, spawns an or-dispatcher for every Concurrent Prolog process A in S, and waits for all its child or-dispatchers to report success. When they do, it reports success and terminates.
- An *or-dispatcher*, invoked with a Concurrent Prolog process A, operates as follows. For every clause $A1$:– $G \mid B$, whose head is potentially unifiable with A, it invokes a unifier with A and the clause $A1$:– $G \mid B$. Following that the or-dispatcher waits for any of the unifiers to report success. When one such report arrives, the or-dispatcher reports success to its parent and-dispatcher and terminates.
- A *unifier*, invoked with a Concurrent Prolog process A and a guarded-clause

> $A1$:- G | B, operates as follows. It attempts to unify A with $A1$, storing bindings made to non read-only variables in A on private storage. If and when successful, it invokes an and-dispatcher with G, and waits for it to report success. When this report arrives, the unifier attempts to commit, as explained below. If the commitment completed successfully it reports success, but in either case it terminates.

At most one unifier spawned by an or-dispatcher may commit. This mutual-exclusion can be achieved by standard techniques, e.g. by using a test-and-set bit for each or-dispatcher in a shared memory model.

To commit, a unifier first has to gain a permission to do so. The mutual-exclusion algorithm must guarantee that if at least one unifier wants to commit, than exactly one unifier will be given permission to do so. After gaining such a permission, the unifier attempts to unify the local copies of its variables against their corresponding global copies. If successful, then the commitment completes successfully.

Some ommissions in the above description are as significant as its content. The abstract computation model of Concurrent Prolog does not deal explicitly with process failure, and identify it with nontermination. The early detection and deletion of failing processes can be introduced as an optimization to the basic model, without affecting its semantics. Another useful optimization is the deletion of brother unifiers, once the first such process is ready to commit.

When committing, the unifier is not required to perform the unification of the public and private copies of variable as an "atomic action". The only requirement is that the unification be "correct", in the sense that it should not modify already instantiated variables, which can be achieved in a shared memory model with a test-and-set primitive.

The intuitive justification for this freedom is the associativity of unification. If several processes attempt to unify several terms simultaneously, then their success or failure is independent of the order in which the bits of unification are attempted. This is the rationale for Warren's parallel unification algorithm [47]. If the unification is bound to fail, it does not matter which process discovers the failure, since processes that unify into the same environment must belong to the same conjunctive system, and if one of them fails then the whole conjunctive systems fails anyway.

Since a unification that currently fails may succeed later, the phrase "attempts to unify" in the description of a unifier should be interpreted as a continuous activity, which terminates only upon success. This can be implemented using a busy-waiting strategy, but several optimizations can be incorporated. First, the reason for the failure can be diagnosed. If the failure is not caused by violation of read-only constraints, then it cannot be cured in the future, and the process attempting this unification can be deleted. Secondly, more sophisticated waiting techniques can be developed if the unification fails due to read-only constraints, for example the technique invented by Alain Colmerauer, described in Section 5.

Our description of the distributed operational semantics of Concurrent Prolog is rather informal, and we are investigating ways to make it more precise. Two issues must be solved in developing a precise semantics to Concurrent Prolog. One is the meaning of infinite computations. Notions from infinitary logic, as proposed by van Emden, de Lucena [16], and Smyth [42], may be utilized. Another is capturing in the formalism the time dependant behavior of Concurrent Prolog. It is conceivable that one can time-stamp logical terms to achieve this.

3.3. An example of a computation

For example, we trace the execution of the process *quicksort*([2, 1, 3], *X*). When invoked, this process can reduce itself with Clause (1) as follows:

quicksort([2, 1, 3], *X*) :− *qsort*([2, 1, 3], *X*−[])

qsort([2, 1, 3], *X*−[]) in turn has two clauses, but only Clause (1) unifies, resulting in the reduction:

qsort([2, 1, 3], *X*−[]) :−
    partition([1, 3]?, 2, *Y*, *Z*), *qsort*(*Y*?, *X*−[2|*W*]), *qsort*(*Z*?, *W*−[])

The system now contains three processes. The two *qsort* processes are suspended, since they can proceed only by instantiating their first argument, which are read-only, either to [_|_] or to []. The *partition* process, on the other hand, has two unifiable clauses — (1) and (2). It invokes two systems, for the two guards in these clauses, but only the first one, 1<3, terminates successfully, and Clause (1) is used:

partition([1, 3]?, 2, [1|*X*], *Y*):−*partition*([3]?, 2, *X*, *Y*)

During this reduction, the first argument of the first *qsort* process is instantiated to [1|X], so it can proceed:

qsort([1|*X*]?, *Y*−[2|*Z*]):−
    partition(*X*?, 1, *V*, *W*), *qsort*(*V*?, *Y*−[1|*Z*1]), *qsort*(*W*?, *Z*1−[2|*Z*])

However, all three new processes are suspended. The only process that can proceed is *partition*([3]?, 2, *X*, [3|*Y*]):

partition([3]?, 2, *X*, [3|*Y*]):−*partition*([]?, 2, *X*, *Y*)

This reduction instantiated the next element on *qsort*'s input stream, so it can proceed:

```
qsort([3|X]?, Y-[]):-
        partition(X?, 3, U, V), qsort(U?, Y-[3|Y1]), qsort(V?, Y1-[])
```

But, again, all new processes are suspended. All reductions left use unit clauses,

```
partition([]?, 2, [], []):-true
partition([]?, 1, [], []):-true
qsort([]?, [1|X]-[1|X]):-true
qsort([]?, [2|X]-[2|X]):-true
partition([]?, 3, [], []):-true
qsort([]?, [3|X]-[3|X]):-true
qsort([]?, []-[]):-true
```

and the computation successfully terminates, with output substitution X=[1, 2, 3].

The computation of the *quicksort* program resembles the computation of the corresponding Lisp program, provided that it uses *rplcd* and Lenient-*cons* [18].

4. Programming Examples

The power of Concurrent Prolog comes from the rich set of elegant programming techniques it supports. The example programs below intend to demonstrate some of them. The tool needed to understand and develop programs written in a high-level language is a mastery of the programming idioms and techniques the programming language lends itself to. Hence, when describing a program, we also attempt to identify the programming idioms and paradigms it exemplifies.

All programs numbered "Program *n*:" have been debugged and tested using the interpreter described in Section 5, whose code is included in the appendix. Their code shown is readily executable without any modifications or additions, unless stated otherwise, with the mini-interpreter in Program 18, which lacks debugging and statistics functions. To run interactively, some of these programs need additional terminal interface software, which is included in the appendix as well.

We tried to follow several notational conventions, demonstrated by the following Concurrent Prolog program for summing a stream of integers.

To denote a stream of elements we use the variable name *S* or the suffix *s*. For

```
sum(S, Total) :- sum(S?, 0, Total).

sum([X|Xs], N, Total) :-
   plus(X, N, N1), sum(Xs?, N1, Total).
sum([], N, N).
```

Program 2: Summing the elements of a stream

example, a stream of *X*'s is usually denotes by *Xs*, as in [*X*|*Xs*]. If a tail recursive (==iterative) program modifies some of its local arguments, then the variable denoting the modified value has the suffix 1, such as in *N*1 (a prime *N*′ may be better, but is not supported by the current Prolog-10 syntax).

We use two procedures with the same name but different arities to initialize the local variables of a process, and to hide their internal representation from the caller. Typically, the process with less arguments is invoked, and in turn invokes the other process with its local variables initializes, as in Program 2.

The read-only annotation in the initialization call *sum*(*S*?, 0, *N*) is not strictly necessary, if *sum*(*S*, *N*) is always called with *S* marked as read-only, but serves as an extra precaution, in case the caller forgets.

4.1. Divide and conquer with communication

The concurrent quicksort program above combines divide-and-conquer with process communication. In quicksort, however, the "divider" process is communicating with the two "conquering" processes, but the latter two do not communicate with each other.

Another algorithm that combines divide-and-conquer with process communication was suggested by Leslie Lamport [34]. In Lamport's algorithm the "conquering" processes do communicate. The problem is to number the leaves of a tree; its solution reads as follows: "The *count* algorithm is a recursive concurrent algorithm for numbering leaves from "left to right". When called on a node, it does the following:

```
If the node is a leaf
   then Obtain the number of leaves to the left.
        Add one to obtain this leaf's number.
        Send my number to the leaf on the right.
   else Call count for each of the sons of this node.
```

The obvious modifications are made when a node is at the left hand or
the right hand edge of the tree..."

We implement the algorithm for binary trees with labeled leaves, constructed from the terms *leaf*(*X*) and *tree*(*L*, *R*). The procedure *count*(*T*) is invoked with the tree *T* whose leaves are to be numbered. It then calls an an auxiliary procedure *count*(*T*, *N*1, *N*2), whose semantics is "*T* is a binary tree with leaves numbered sequentially from *N*1 to *N*2-1".

```
count(T) :- count(T, 0, N).

count(leaf(N), N, N1) :- plus(N, 1, N1).
count(tree(L, R), N, N2) :- count(L, N, N1), count(R, N1, N2).
```

Program 3: Numbering the leaves of a tree

The process *plus*(*X*, *Y*, *Z*) computes the relation "X plus Y is Z". It waits untill at least two of its arguments are determined, then unifies the third with the appropriate number and terminates. Its implementation is shown in the appendix.

Note that no modifications are needed when the node is at the edge of a tree.

An implementation of Lamport's algorithm in Concurrent And/Or Programs is shown in [23].

4.2. Perpetual processes with internal states

Traditionally, logic-programming researchers have emphasized the stateless, side-effect free, declarative style of programming in logic. This emphasis is justified when the problems to be solved can be stated without reference to the state of the computation, and it helps to show the difference between logic programming and conventional programming. However, when discussing concurrent computations, sometimes the very nature of the problem contains reference to the state of the computation. Hence a slightly different "ideology" has to be adopted to understand the role of logic programming in concurrent computations.

In sequential logic programming, the axiom *A* :- *B* has, in addition to the declarative, model-theoretic reading: "*A* is true if *B* is true", also the operational, problem-reduction reading: "to solve *A*, solve *B*". In concurrent logic programming, a third kind of reading is necessary, the behavioral reading: "process *A* can reduce itself to system *B*". Under the behavioral reading, a concurrent logic program simultaneously provides an axiomatic definition of the possible behaviors of a process, and the "code" the process is executing.

As demonstrated below, a logic program that implement a concurrent system is stateless, side-effect free, and provides an axiomatic description of a set of legal behaviors of a system of processes, which have a state. Hence it is meaningful to talk about the implementation of processes with states by pure logic programs.

As mentioned earlier, the state of a process is the value of its arguments. According to the abstract computation model, a process cannot actively change its state, but only reduce itself to other processes. Hence, theoretically, Concurrent Prolog supports only ephemeral processes whose state is not self-modifiable. However, both from an intuitive and an implementation point of view, a process that calls itself recursively with different arguments can be viewed as a perpetual process that can change its state. If the implementation incorporates tail-recursion optimization, then the same process frame may actually be used for the different incarnations of such a process.

Under the behavioral interpretation, the arguments of the process not shared by other processes can be viewed as its internal state, since they can neither be accessed nor modified by other processes. The situation can be summarized with the following "equation":

Tail recursion + local variables = perpetual processes with internal states

The ability to implement multiple perpetual processes is one reason for the increased power of Concurrent Prolog over sequential Prolog. Sequential Prolog can implement one perpetual process without side-effects, for example a text-editor [46]. However, when multiple independent global objects have to be manipulated, most programmers in sequential Prolog resort to side-effects to implement them. In Concurrent Prolog, on the other hand, global data-structures are implemented by multiple perpetual processes in a side-effect free way. Instead of accessing global data using "read" and "write" operations, messages are sent to the process holding the data, which in turn informs the sender of the content of the data and/or modifies it, according to the message, similar to

This approach is similar to the Actors [25] and SmallTalk approach in spirit [29]. However, in contrast with the pure operational character of other object-oriented formalisms, the logic programs that describe such processes enjoy the declarative/operational duality that singles out the logic programming solutions to other computing problems. In particular, the Concurrent Prolog implementation of a perpetual processes with an internal state resembles the axiomatic definition of an abstract data type. For example, consider the implementation of a stack process in Program 4.

A stack process has two arguments: the first is an input stream; the second stores

```
(0) stack(S) :- stack(S?, []).

(1) stack([pop(X)|S], [X|Xs]) :- stack(S?, Xs).
(2) stack([push(X)|S], Xs) :- stack(S?, [X|Xs]).
(3) stack([], []).
```

Program 4: A stack process.

the stack content, represented as a list of stack elements.

A stack process is invoked with the call *stack*(*S*?), where *S* is the input stream. Using the first clause in Program 4, it initializes itself with an empty stack. It then iterates, processing the messages on its input stream. If no message is available, the process suspends, due to the read-only annotation *S*? in the recursive calls to *stack*. Otherwise, one of the following three cases, which correspond the last three clauses in the program, must apply:

- Clause (1) applies if the next message on the input stream is *pop*(*X*), and the stack is nonempty. It unifies X with the top of the stack, and iterates with the rest of the input stream and the rest of the stack.
- Clause (2) applies if the next message is *push*(*X*). It adds X to the top of the stack, and iterates with the rest of the input stream and the new stack.
- Clause (3) applies if the end of the input stream is reached *and* the stack is empty. It simply terminates.

If none of these cases apply, the process fails. In particular, the process fails if the next message is *pop*(*X*) and the stack is empty, or if the end of the input stream is reached and the stack is nonempty.

The model-theoretic semantics of the stack program is simple. The interpretation of the stack program contains all goals *stack*(*S*, []) in which *S* is a balanced list over the alphabet *pop*(*X*) and *push*(*X*), where *X* ranges over the elements of the Herbrand universe of Program 4 and *pop*(*X*) is considered to be the matching right parenthesis to *push*(*X*), for any term *X*. The interpretation also contains all goals *stack*(*S*, [*X*1, *X*2, ..., *Xn*]), where *S* is a list that can be balanced by prefixing it with *push*(*Xn*), *push*(*Xn*−1), ..., *push*(*X*2), *push*(*X*1).

It is easy to augment or modify the stack program. For example, if we want the stack process to terminate successfully even if the stack is not empty, then the third clause can be modified to be *stack*([], _). The interpretation of the program then grows to include all goals *stack*(*S*, []) in which *S* is a prefix of some balanced list over *push*(*X*) and *pop*(*X*).

To make the stack process understand the message *is_empty*(X), by unifying X with *true* if the stack is currently empty and with *false* otherwise, we add the following two clauses to Program 4:

```
stack([is_empty(true)|S], []) :- stack(S?, []).
stack([is_empty(false)|S], [X|Xs]) :- stack(S?, [X|Xs]).
```

Typically, a process sends *pop*(X) with X undetermined, and waits for X to be instantiated when the message is processed by the stack. This habit can be made the rule: we can enforce the sender of a *pop*(X) message to leave X undetermined, by replacing the first clause of Program 4 by the clause:

```
stack([pop(X?)|S], [X|Xs]) :- stack(S?, Xs).
```

If *pop*(X) is sent with X determined to a stack executing the modified program, then the stack process fails.

It is not, however, always desirable to enforce this restriction. For example, Program 5, which tests whether a list (stream) is a balanced list over the alphabet '(', ')', '{', '}', would become more cumbersome if this restriction was in effect.

```
balanced(X) :- balanced(X, Y), stack(Y?).

balanced(['('|X], [push('(')|Y]) :- balanced(X?, Y).
balanced(['{'|X], [push('{')|Y]) :- balanced(X?, Y).
balanced([')'|X], [pop('(')|Y]) :- balanced(X?, Y).
balanced(['}'|X], [pop('{')|Y]) :- balanced(X?, Y).
balanced([], []).
```

Program 5: Testing balanced lists

balanced is invoked with an input stream and then spawns two processes: one translates the input stream into a stream of stack messages; the other is a stack, which executes Program 4.

The *balanced* program can be explained using Actor's jargon [25]. When the *balanced* process receives an '('} message, it sends a *push*('(') message to the stack; it operates similarly on an '{' message. When it receives an ')' message, it sends a *pop*(X) message to the stack, and verifies that X='('; similarly with '}'. When the end of the message stream is reached, it terminates, and terminates its communication stream with the stack. If both processes terminate successfully, then the whole computation terminates, and the stream is balanced.

The *balanced* program fails if the list is not balanced or contains illegal elements. It is easy to modify *balanced* so it returns *true* if the list is balanced, *false* if it is imbalanced, and fails if it contains illegal elements. This can be done by adding another argument to the two *balanced* predicates, replacing the last clause with *balanced*([], [*is_empty*(*Response*)], *Response*), and adding to Program 4 the clauses for *is_empty*.

This example is a bit contrived, since *balanced* can be implemented with a local stack, rather than with a stack process. We leave this as an exercise to the reader.

4.3. The readers and writers problem

Many problems in which several processes share some resource can be modeled after the readers and writers problem [27]. A logic program solution for a specific readers and writers problem can be obtained by instantiating the following program scheme, based on an idea by Bowen and Kowalski [5].

```
process([Transaction|S], Data) :-
    respond(Transaction, Data, NewData) | process(S?, NewData).
process([], _).
```

The process *process* has two arguments: one is a stream of transactions from several processes, serialized by *merge* processes (explained below); the other contains the data. On each transaction, the process *respond* is invoked with the transactions and the data. *respond* returns the modified data, and possibly instantiates undetermined variables in the message.

The stack program above is an instance of this scheme: *push* and *pop* messages "write" on the shared data, i.e. cause it to be modified, and the *is_empty* message only "reads" the data, without modifying it. The stack process responds to the "read" transaction *is_empty*(X) by instantiating X to *true* or *false*; it responds to the "write" transaction *push*(X) by adding X to the top of the stack; and it responds to the "read/write" transaction *pop*(X) by removing the top element from the stack, and unifying it with X.

This scheme can be improved by distinguishing between "read" transactions that only query the data without modifying it, and "write" transactions, which also modify the data. By doing so, a contiguous sequence of "reads" can be served in parallel, since the tail recursive call to *process* can be performed with *Data* rather than with *NewData*, and hence need not wait for the completion of *respond*. Once a "write" is received, however, this optimization cannot be done, since *process* must iterate with *NewData*.

Figure 4 contains a schematic implementation of this solution, assuming that transactions are either *read(Args)* or *write(Args)*.

```
process([read(Args)|S], Data) :-
    respond(read(Args), Data, _), process(S?, Data).
process([write(Args)|S], Data) :-
    respond(write(Args), Data, NewData) | process(S?, NewData).
process([], _).
```

Figure 4: A schematic solution to the readers and writers problem

4.4. An airline reservation system

A classical instance of the readers and writers problem is the airline reservation system problem. Bryant and Dennis [9] used this system as a benchmark problem for comparing different approaches to concurrent programming. We contribute our own version to the contest.

Their description of the problem reads as follows:

"The process for the airline reservation system contains information about the flights of a single airline. Initially, each flight has 100 seats available. The system can accept two kinds of commands. To reserve seats on a flight an agent given the command (*'reserve'*, *f*, *n*). If at least *n* seats are available on flight *f*, the seats will be reserved, and the system will respond with the message *true*. If that many seats are not available, the system will respond with the message *false*. To find out how many seats are available on flight *f*, a system user given the command (*'info'*, *f*). The system will respond with the number of seats which are available on the flight at the time the command is processed." (from [9] p.430)

Program 6 implements the system, using two primitives procedures: *value(A, N, V)*, which computes the relation: "the value of the N^{th} element of *A* is *V*", and *modify(A, N, V, A1)*, which computes the relation "changing the N^{th} element of *A* to *V* gives *A1*".

database is the main process. It has two arguments: an input stream of transactions, and flight-availability data. Clause (1) handles requests for availability information. On a message *info(Flight, Seats)*, it unifies *Seats* with the number of free seats available on flight *Flight*, and iterates with the rest of the input stream and the unmodified database. Clause (2) handles reservation requests. It uses a procedure *reserve(Flight, Seats, DB, Response, DB1)*, which unifies *Response* with *true* if the number of available seats in flight *Flight* is less than or equal to *Seats*, otherwise it

```
(1) database([info(Flight, Seats)|S], DB) :-
        value(DB, Flight, Seats),
        database(S?, DB).
(2) database([reserve(Flight, Seats, Response)|S], DB) :-
        reserve(Flight, Seats, DB, Response, DB1) |
        database(S?, DB1).
(3) database([], _).

(1) reserve(Flight, Seats, DB, Response, DB1) :-
        value(DB, Flight, FreeSeats),
        plus(Seats, LeftSeats, FreeSeats),
        respond(DB, LeftSeats, Flight, Response, DB1).

(1) respond(DB, Seats, Flight, true, DB1) :-
        le(0, Seats) | modify(DB, Flight, Seats, DB1).
(2) respond(DB, Seats, _, false, DB) :-
        lt(Seats, 0) | true.
```

Program 6: An airline reservation system.

unifies *Response* with *false*. It returns in *DB*1 the resulting database. Clause (3) terminates the process when the end of the input stream is reached.

The *reserve* process invokes three concurrent processes. The first one finds the number of available seats, the second computes the number of free seats left if the request is granted, and the third responds to the request accordingly. The three processes are synchronized by the availability of input data. *plus* waits for *value* to determine the value of *FreeSeats*, and the guards of *respond* wait for *plus* to compute *LeftSeats*.

The *respond* procedure has two clauses. Clause (1) returns *true* and modifies the database if the number of seats left after the reservation is granted is greater or equal to 0. Otherwise, Clause (2) returns *false*, without modifying the database.

The process *le*(X, Y) suspends untill both of its arguments are determined, and then succeeds or fails according to their values. It is implemented via a call to Prolog's arithmetic predicate $X \leq Y$:

le(X, Y) :- *wait*(X), *wait*(Y) | $X \leq Y$.

where *wait*(X) is a built-in Concurrent Prolog predicate that waits until X's main functor is determined, then terminates. Its main use is to interface between Concurrent Prolog and the underlying Prolog-10, since the latter does not know currently about read-only variables. A Prolog implementation of *wait* is shown in the appendix.

The airline reservation system is not very interesting unless it can serve many users concurrently. This is achieved by merging all streams of queries of the clients into one, as described in the following section.

4.5. Merging streams

Many concurrent programming languages use streams to support process communication. Streams are typically introduced as a new data-type, to which specialized "read" and "write" operations are defined [1, 31]. The main difference between streams and lists is that the former are only partially determined at each point of the computation. Since partially determined data-structures are supported by Prolog — both sequential and concurrent — there is no need to introduce a new data-structure into Prolog in order to implement streams, and the usual list constructors will do. Unification is used to "read" or "write" the next stream element, and read-only annotations distinguish between the "readers" and the "writers" of a stream.

The use of streams was already demonstrated in the *quicksort* and *balanced* programs above.

A process can have several input and/or output streams, and use them to communicate with several other processes; but the number of these stream is fixed for any given process. It is sometimes convenient to determine or change at runtime the number of processes communicating with another process; this can be achieved by merging communication streams.

In some languages *merge* is a built in operator [3]. Logic programs, on the other hand, can express this function directly, as shown by Clark and Gregory [11]. Program 7 adapts their implementation to Concurrent Prolog.

```
merge([X|Xs], Ys, [X|Zs]) :- merge(Xs?, Ys?, Zs).
merge(Xs, [Y|Ys], [Y|Zs]) :- merge(Xs?, Ys?, Zs).
merge(Xs, [], Xs).
merge([], Ys, Ys).
```

Program 7: Merging two streams

It implements the process *merge*(*X*, *Y*, *Z*), which computes the relation "*Z* contains the elements of *X* and *Y*, preserving the relative order of their elements".

The read-only annotation on *Ys* in the first clause and on *Xs* in the second clause are superfluous, provided that *merge* is initially invoked with its two input streams annotated read-only.

There is an ongoing discussion concerning the desired properties of a *merge* operator, and how it can be specified [37]. Smyth [42], apparently unaware of the work of Clark and Gregory, has suggested the axioms in Program 7 as a *specification* of a fair merge operator, and has shown that this specification has desirable mathematical properties, such as commutativity and associativity.

We do not find the properties shown by Smyth sufficient, since they do not guarantee bounded-waiting. In other words, given the positions of two elements in an input stream, we cannot bound the difference between their positions in the output stream on the basis of this information alone. In implementation terms, if two elements are ready in both input stream, then, without any additional information on how the logic program is executed, we cannot bound the number of merge process reductions needed for the two elements to appear in the output stream.

However, since for us the logic program is also an implementation of the *merge* operator, not only a specification, we can employ information concerning the behavior of the underlying Concurrent Prolog machine (e.g. fairness of the scheduler) to determine whether the program achieves the desired effect or not.

Assume that the clauses in the program are ordered (say, by text order), and assume that a process A has several clauses A_i :− B_i, $1 \leq i \leq n$, with empty guard whose head unifies with a process A, at the time A is created. We say that a Concurrent Prolog machine (interpreter) is *stable* if it always chooses the first clause A_1 :− B_1 to reduce A.

A Concurrent Prolog machine for which this condition holds if the number of steps required to unify A with A_1 is less than or equal to those required to unify A with A_i, for $1 < i \leq n$, is called *weakly stable*. Note that a stable machine is also weakly stable.

Any reasonable centralized Concurrent Prolog interpreter should be stable, unless one makes a special effort to eliminate this property, as suggested by Dijkstra [15]. For example, the interpreter in Program 18 is stable. We suspect that any distributed Concurrent Prolog machine should be weakly stable, or at least weakly stable with high probability.

If the implementation is stable, then the *merge* program above clearly does not achieve bounded waiting. If both streams have elements ready, then the first stream will always be chosen. When *merge*(X, Y, Z) is invoked with X and Y finite and determined, Program 7 running on a stable machine simply concatenates X to Y. In extreme cases, when infinite computations are involved, this behavior can cause elements of the second stream to wait indefinitely before they appear in the output stream.

Nevertheless, this behavior is desirable in some cases. Since the first stream has "higher-priority" than the second, this program, or a similar one, can implement interrupts, where the second stream carries the normal communication, and the first one carries exceptional communication, which should interrupt the normal execution. Several such merge processes can be composed to implement interrupts with different relative priorities.

However, if a bounded-waiting merge is desirable, it can be implemented in at least two ways, provided the Concurrent Prolog machine is at the least weakly stable. One is to alternate priorities between the two streams, an idea suggested by Johnson [30] in the context of functional concurrent programming, and implemented by the following program:

```
merge([X|Xs], Ys, [X|Zs]) :- merge(Ys, Xs?, Zs).
merge(Xs, [Y|Ys], [Y|Zs]) :- merge(Ys?, Xs, Zs).
```

which has base clauses as before. This program alternates priorities between its two streams. In every reduction the first stream becomes the second, and vice versa. Note that the priority makes a difference only if both streams have elements ready; if only one stream is ready, then its elements are moved to the output stream regardless of priorities.

To fairly merge more than two streams, one can either compose this merge program, such as in the system

```
merge(X1, X2, X), merge(Y1, Y2, Y), merge(X, Y, Z)
```

or use an n-ary merge. The same priority-based technique generalizes to merging n streams, for any fixed n, and the resulting program is a kind of a round-robin stream-scheduler. Its i^{th} recursive clause is:

```
merge(X1, X2, ..., [X|Xi], ..., Xn, [X|Ys]) :-
        merge(X2, X3, ..., Xi?, ..., Xn, X1, Ys).
```

This program rotates priorities between its streams. In each reduction the highest priority stream becomes the one with the least priority, and all other streams increase their relative priority by one.

Another strategy to implement a fair merge is to decrease the priority of a stream that has been "read". The i^{th} recursive clause in this type of n-ary merge is:

```
merge(X1, ..., [X|Xi], ..., Xn, [X|Ys]) :-
        merge(X1, ..., Xi-1, Xi+1, ..., Xn, Xi?, Ys).
```

It is easy to show that if the Concurrent Prolog machine is weakly stable then both n-ary merge programs guarantee n-bounded waiting, which means that if the first element of an input stream is determined, then after at most n reductions of *merge* this element will appear in the output stream. The choice between the two strategies is application dependant, but we conjecture that in any reasonable Concurrent Prolog machine the round-robin scheduler would be more efficient, since the other strategy inspects most often the less busy streams before it finds a stream with an element ready.

We expect these scheduling strategies to be effective even in an implementation that is weakly stable with high probability only.

4.6. The MSG message-sending system

A simple application of the *merge* program is shown by Johnson [30]:

"MSG is a full duplex message sending system for two computer terminals, A and B. Input from A's (respectively B's) keyboard, K1 (K2) is echoed on A's (B's) screen, S1 (S2). However, when K1 (K2) issues a "send", the following form should be displayed *in a timely fashion* on S2 (S1)" (from [30], p.15).

The the MSG system is invoked by the call *msg*(($K1?$, $S1$), ($K2?$, $S2$)), and is implemented by Program 8 which invokes a system whose configuration is shown in Figure 5.

The *select* procedure selects the output streams on which the keyboard input is echoed. It uses a a built-in Concurrent Prolog procedure, *dif*(X, Y) which succeeds if and when it is established that X and Y are different, i.e. will not unify. Concurrent Prolog's *dif* is a variant of a built-in procedure of Prolog-II [13], which has the same name; a Prolog implementation of *dif* is included in the appendix. The code needed to test the MSG system on a single terminal is included in the appendix as well.

4.7. A Unix-like shell

The following program fragments implement part of the functions of the Unix shell [14].

The shell in Figure 6 receives a stream of commands and executes them. If the stream contains the abort command control-C, then it aborts the current execution, flushes the input stream until after the control-C, and resumes execution from there. The shell achieves this by racing the two guards. The first tried to solve the command X; the second to find a control-C in the input stream. The first to succeed commits the shell to the appropriate action.

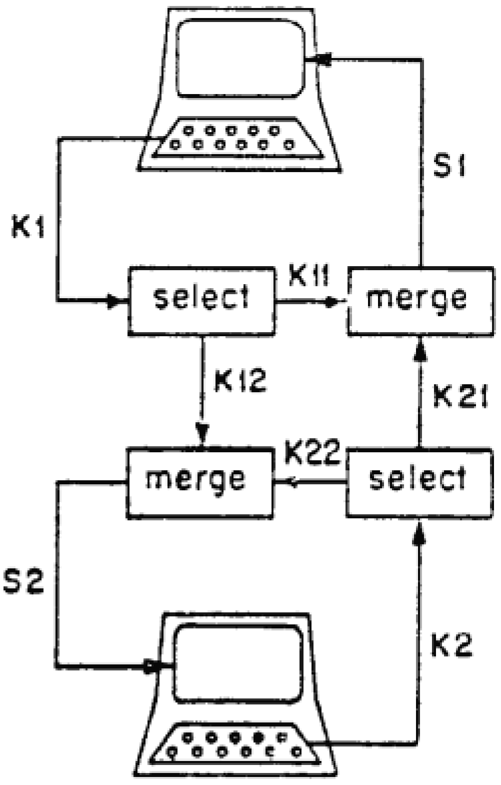

Figure 5: The MSG system

```
msg((K1, S1), (K2, S2)) :-
    select(K1?, K11, K12), select(K2?, K22, K21),
    merge(K21?, K11?, S1), merge(K12?, K22?, S2).

select([send(X)|Xs], [send(X)|Ys], [X|Zs]) :-
        select(Xs?, Ys, Zs).
select([X|Xs], [X|Ys], Zs) :-
        dif(X, send(_)) | select(Xs?, Ys, Zs).
select([], [], []).

merge(X,Y,Z) :-
        See Program 7.
```

Program 8: The MSG system

Assuming that the input stream is generated by a user sitting at a terminal, then such a user can delay typing control-C until he thinks that the program X may be

```
shell([X|Xs]) :- command(X) | shell(Xs?).
shell(Xs) :- control_c(Xs, Ys) | shell(Ys?).

control_c([X|Xs], Ys) :-
        diff(X, '↑C') | control_c(Xs?, Ys).
control_c(['↑C'|Xs], Xs).
```

Figure 6: A shell that handles an *abort* interrupt

looping, or discovers that X is not really the program he wanted to run, and type control-C then.

This program achieves the desired effect only if both guards are executed in parallel, and hence does not run correctly on our interpreter, as explained in Section 5. Note that the program works correctly even if one types ahead of the execution speed of the shell.

Another convenient feature of the Unix's shell is the ability of the user to specify that a process should run in the background. This increases the responsiveness of the system, and enables the user to do other things while non-interactive programs such as a compiler are running. This behavior is easily achieved in Concurrent Prolog, as shown by the program fragment in Figure 7.

```
shell([fg(X)|Xs]) :- command(X) | shell(Xs?).
shell([bg(X)|Xs]) :-command(X), shell(Xs?).
shell([]).
```

Figure 7: A shell that handles
background and foreground processes

The shell assumes that the user's commands are tagged either *fg* or *bg*. If the command is intended to be executed in the "foreground", then the shell executes it as a guard, and only after it terminates it iterates, ready to receive the next command on the input stream. On the other hand, if the command is to be executed in the background, the shell spawns it as a brother process, and is immediately available to execute the next command.

Note that these code fragments do not handle terminal output. Presumably, if a process is spawned in the background then a third *merge* process should be invoked to allow both the shell and the background process to communicate with the user.

4.8. Queues

Merged streams allow many client processes to share one resource; but when several client processes want to share several resources effectively, a more complex buffering strategy is needed. Such buffering can be obtained with a simple FIFO queue: a client who requires the service of a resource enqueues its request. When a resource becomes available it dequeues the next request from the queue and serves it.

Note that if there is only one resource but many clients one can obtain the effect of a FIFO queue by fairly merging all the requests into one stream, and letting the resource serve the requests in the order in which they arrive.

Program 9 is a Concurrent Prolog implementation of a FIFO queue. It handles two types of messages: *enqueue(X)*, on which it adds X to the tail of the queue, and *dequeue(X)* on which it removes the first element from the head of the queue and unifies it with X. It represents the queue using two streams — the Head stream that contains the dequeued elements, and the Tail stream that contains the enqueued elements. The content of the queue is defined to be the difference between the Head stream and the Tail stream.

```
(1) queue(S) :-
        queue(S, X, X).

(2) queue([dequeue(X)|S], [X|NewHead], Tail) :-
        queue(S?, NewHead, Tail).
(3) queue([enqueue(X)|S], Head, [X|NewTail]) :-
        queue(S?, Head, NewTail).
(4) queue([], _, _).
```

Program 9: A queue

Clause (1) initializes the queue with the Head and Tail streams equal and undetermined. Clause (2) unifies dequeued elements with elements of the Head stream, and Clause (3) unifies enqueued elements with elements of the Tail stream. Clause (4) terminates the queue process when the end of the input stream is reached. The read-only annotation of the stream S in the recursive calls ensures that the queue process waits for the next message to be determined, and does not decide to enqueue or dequeue an element on its own.

This program is very concise, and has a simple model theoretic semantics. Its interpretation contains all goals *queue(S, H, T)* such that S is a list of terms *enqueue(X)* and *dequeue*(X), T equals the list of X's for which *enqueue*(X) is in S and H equals the list of X's for which *dequeue*(X) appears in S, where elements in both lists preserve the relative order of their corresponding terms in S. If we restrict the interpretation to goals in which H=T, as done by the Clause (1), which initializes the queue, then the list of enqueued elements in S is identical to the list of dequeued elements in S, as the intuitive definition of a queue requires.

Operationally, things are a bit more tricky. Under the expected use of a queue, *enqueue*(X) messages are sent with X determined, and *dequeue*(X) with X undetermined; typically, the sender of *dequeue*(X) waits for X to be determined. Since the Head and Tail streams are initially undetermined and equal, then as long as more elements are enqueued then dequeued, the Tail stream runs ahead of the Head, and the difference between the two are exactly the elements that were enqueued but not dequeued yet. However, if the number of *dequeue* messages received exceeds that of enqueue messages, then an interesting thing happens — the content of the queue becomes "negative". The Head runs ahead of the Tail, resulting in the queue containing a negative sequence of undetermined elements, one for each excessive *dequeue* message. Although the queue process serves each *dequeue*(X) message as it comes, if the queue is empty it does not unify X with a concrete element, but only generates another undetermined stream element. When enough *enqueue* messages are received, the Tail will reach this element, and unify it with the next enqueued element.

In Lisp implementation jargon, Head and Tail are pointers to *cons* cells of the same list. An *enqueue* message advances the Tail pointer by one, and a *dequeue* message advances the Head pointer by one. When the Head pointer overtakes the Tail, it starts allocating *cons* cells, and unify their *car* with the undetermined variables X in the messages *dequeue*(X). When the Tail pointer is advances it unifies the *car* fileds of these cells with the enqueued elements.

It is interesting to observe that this behavior is compatible with common properties of queues, such as the associativity of queue concatenation. The concatenation of the two difference lists $X-Y$ and $Y-Z$ is defined to be $X-Z$. If we concatenate a queue $X-[X1,\ X2,\ X3|X]$ which contains minus three undetermined elements with a queue $[a,\ b,\ c,\ d,\ e|Y]-Y$ which contains five elements, than the result will be the queue $[d,\ e|Y]-Y$ with two elements, where the "negative" elements $[X1,\ X2,\ X3]$ are unified with $[a,\ b,\ c]$.

All this behavior is transparent to the user of a queue. A sender of a *dequeue*(X) messages does not know whether X becomes determined when the queue process has actually received this message or a bit later, when enough *enqueue* messages have arrived.

4.9. A simulator of a multiprocessor Concurrent Prolog machine

The simulator of a multiprocessor concurrent Prolog machine in Program 10 is an exercise in utilizing the queue and merge programs. The simulator is invoked with a number N and a process A. It constructs a system of one queue, N processor-simulators, and a balanced-tree shaped network of $2N-1$ merge processes that support the communication between the processors and the queue.

```
processors(N, X) :-
        queue(S?, [X|Xs], Xs), processors(1, N, S).

processors(N, N, Q) :-
        processor(N, true, Q).
processors(N1, N4, Q) :-
        N2 is (N1+N4)/2,  N3 is N2+1 |
        processors(N1, N2, Q1),
        processors(N3, N4, Q2),
        merge(Q1?, Q2?, Q).

processor(N, true, [dequeue(X)|Q]) :-
        processor(N, X?, Q).
processor(N, (A, B), [enqueue(A)|Q]) :-
        processor(N, B, Q).
processor(N, suspended(A), [dequeue(B), enqueue(A)|Q]) :-
        processor(N, B?, Q).
processor(N, A, Q) :-
        preduce(A, B) | processor(N, B, Q).

preduce(A, B) :-
        wait(A, A1) | call(reduce(A1, B)).
```

Program 10: A simulator of a multiprocessor Concurrent Prolog machine

Each processor is invoked with an identifier N, a goal *true*, and communication stream Q to a queue. A processor implements the following algorithm:

- If its goal is *true* it sends a *dequeue(X)* message to the queue and iterates with X.

- If its goal is (B, C) then it enqueues B and iterates with C.
- If its goal is *suspended*(A) is it sends an *enqueue*(A) message and a *dequeue*(B) message to the queue and iterates with B.
- If its goal is reducible to B then it iterates with B.

Program 10 abstracts away the management of the binding environment, and deals only with the flow of control. The Prolog procedure *reduce* belongs to the underlying concurrent Prolog interpreter, explained in Section 5. *wait*$(X, X1)$ is like *wait*(X), except that it returns in $X1$ the result of peeling-off all read-only annotations from the main functor of X.

A window-manager system, written in Concurrent Prolog, was used to animate the behavior of the simulator. We have implemented a program that runs each processor in a separate window, and shows the progress of the computation. It creates a recursive structure of windows, depending on the number of processors in the network, as shown in Figure 8, and shows the progress of each processor, and the content of the communication streams and the queue.

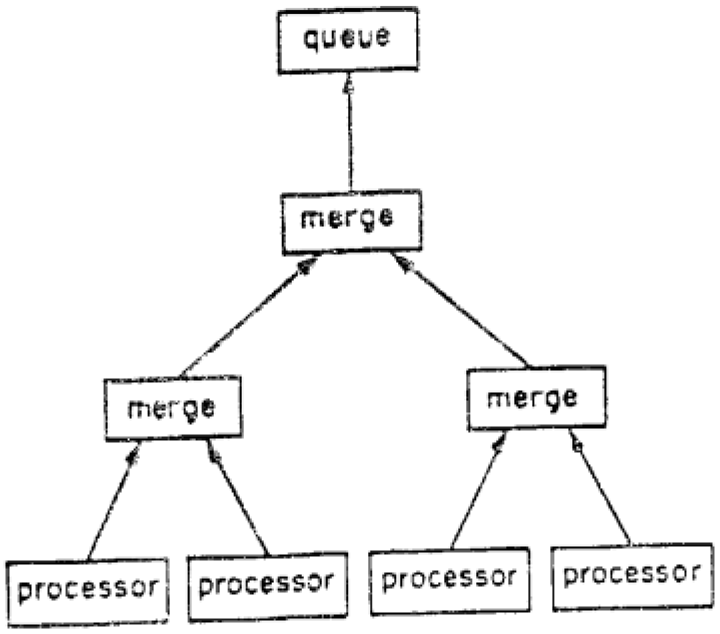

Figure 8: A multiprocessor Concurrent Prolog machine

The window system will be described in a subsequent paper. Unfortunately, current publication techniques do not support such animations. Hence we settle for a more static description of a computation of the simulator.

In the trace below the messages from the N^{th} processor are:

dequeue(N): A	The processor requests a new process from the queue since its current process A cannot be reduced any further, either because A is *true* or because it is suspended.
enqueue(N): A	The processor enqueued process A, since it has more than one process.
reduction(N): A:$-B$	The processor reduced process A to system B.

We invoke the Concurrent Prolog interpreter of the appendix with the call to the simulator to solve the goal *qsort*([2, 1], X) with four processors.

| ?– *solve*(*processors*(4, *qsort*([2, 1, 3], X))).

dequeue(1): *true*
dequeue(2): *true*
dequeue(3): *true*
dequeue(4): *true*

First the four processors complete the solution of the goal *true* they where invoked with, and send a *dequeue* message to the queue.

reduce(1): *qsort*([2, 1, 3], X)? :–*qsort1*([2, 1, 3], X–[])
reduce(1): *qsort1*([2, 1, 3], X–[]):–
    partition([1, 3]?, 2, Y, Z), *qsort1*(Y?, X–[2|W]), *qsort1*(Z?, W–[])

Processor 1 received the input goal, it reduced it twice, and now it enqueues two out of three new processes.

enqueue(1): *partition*([1, 3]?, 2, X, Y)
enqueue(1): *qsort1*(X?, Y–[2|Z])

Processor 3 received the process enqueued by 1, and reduces it:

reduce(3): (*partition*([1, 3]?, 2, [1|X], Y)? :–*partition*([3]?, 2, X, Y))

As a result the second process enqueued by Processor 1 and dequeued by Processor 2 becomes reducible:

reduce(2): (*qsort1*([1|X]?, Y–[2|Z])? :–
    partition(X?, 1, V, W), *qsort1*(V?, Y–[1|$Z1$]), *qsort1*(W?, $Z1$–[2|Z]))

Processor 3 completes the reduction, while 1 enqueues its suspended process and request a new one and reduces it, and 2 enqueues two of the three processes in its system.

```
reduce(3): (partition([3]?, 2, X, [3|Y]):-partition([]?, 2, X, Y))
dequeue(1): qsort1([3|X]?, Y-[])
enqueue(2): partition(X?, 1, Y, Z)
reduce(3): (partition([]?, 2, [], []):-true)
reduce(1): (qsort1([3]?, X-[])? :-
        partition([]?, 3, Y, Z), qsort1(Y?, X-[3|W]), qsort1(Z?, W-[]))
enqueue(2): qsort1(X?, Y-[1|Z])
dequeue(3): true
```

following that all that remain are processes that reduce themselves to *true*:

```
reduce(4): (partition([]?, 1, [], [])? :-true)
enqueue(1): partition([]?, 3, X, Y)
reduce(2): (qsort1([]?, [2|X]-[2|X]):-true)
reduce(3): (qsort1([]?, [1, 2|X]-[1, 2|X])? :-true)
dequeue(4): true
enqueue(1): qsort1(X?, Y-[3|Z])
dequeue(2): true
dequeue(3): true
reduce(4): (partition([]?, 3, [], [])? :-true)
reduce(1): (qsort1([]?, []-[]):-true)
reduce(3): (qsort1([]?, [3]-[3])? :-true)
dequeue(4): true
dequeue(1): true
dequeue(3): true
```

Now all processors have sent a *dequeue* message to the queue and wait for its response, but the queue is empty. This deadlock is detected by the interpreter, which terminates and presents the locked processes and their interconnections. Note that the queue has minus four elements, one of every unsatisfied *dequeue* request from a processor.

```
*** cycles: 17
*** Deadlock detected. Locked processes:
processor(1, X?, Y)
processor(2, Z?, U)
merge(U?, Y?, X1)
processor(3, Y1?, Z1)
processor(4, U1?, V1)
merge(V1?, Z1?, Y2)
merge(X1?, Y2?, V2)
queue(V2?, X3, [Z, U1, X, Y1|X3])
```

The interpreter also provides some statistics on the execution of each process:

```
# enqueue(1): 4
# enqueue(2): 2
# dequeue(1): 3
# dequeue(2): 2
# dequeue(3): 4
# dequeue(4): 3
# reduce(1): 5
# reduce(2): 2
# reduce(3): 5
# reduce(4): 2
```

which shows that the load of work was fairly balanced between the processors, considering the small size of the example. Statistics on the behavior of the interpreter — how many reductions and suspension occured at each level of process invocation — are also provided:

```
# reduction(1): 180
# reduction(2): 37
# suspension(1): 83
# suspension(2): 20
```

The rate of suspensions to reductions measures the scheduling overhead. In this example it is close to 50%. Finally, we also get the output of the computation, which is $X = [1, 2, 3]$.

4.10. Priority queues

A priority queue require a different representation from a FIFO queue since it needs to be manipulated explicitly. In the following example a priority queue is represented as a list of pairs (X, P), where X is the element and P is its associated priority. On *enqueue*(X, P) the queue process inserts X to the list according to its priority; on *dequeue*(X) it removes X from the head of the list. Program 11 was our first attempt at implementing a priority queue.

Although this program looks benign, it has a serious bug. The reader may wish to meditate on the program, trying to find the bug (or, alternatively, prove the program correct) before proceeding.

The queue process will fail if its next message is *dequeue*(X) and the list representing the queue is empty: only Clause (2) handles a *dequeue* message, and it attempts to unify the second argument of *queue* with a nonempty list. One would like in this case to suspend processing the *dequeue* messages until an *enqueue* message

```
(1) queue(S) :- queue(S?, []).

(2) queue([dequeue(X)|S], [(X, _)|Q]) :-
        queue(S?, Q).
(3) queue([enqueue(X, P)|S], Q) :-
        insert((X, P), Q?, Q1), queue(S?, Q1).
(4) queue([], _).

(1) insert((X, P), [(X1, P1)|Q], [(X, P), (X1, P1)|Q]) :-
        P≤P1 | true.
(2) insert((X, P), [(X1, P1)|Q], [(X1, P1)|Q1]) :-
        P1<P | insert((X, P), Q?, Q1).
(3) insert((X, P), [], [(X, P)]).
```

Program 11: A priority queue (first trial)

arrives, and process the *enqueue* message first. This can be attained by splitting the requests into two stream, one for *enqueue* and one for *dequeue* messages, as done in Program 12. The new implementation serves *dequeue* messages only if its queue is nonempty; otherwise it waits for an *enqueue* message.

```
(1) queue(Es, Ds) :- queue(Es?, Ds?, []).

(2) queue(Es, [dequeue(X)|Ds], [(X, _)|Q]) :-
        queue(Es, Ds?, Q).
(3) queue([enqueue(X, P)|Es], Ds, Q) :-
        insert((X, P), Q?, Q1), queue(Es?, Ds, Q1).
(4) queue([], [], _).

insert(X, Q, Q1) :-
        See Program 11.
```

Program 12: A priority queue (second trial)

Clause (1) initializes the empty queue. Clause (2) handles the case in which the queue is nonempty and a *dequeue* message is ready. Clause (3) handles ready enqueue messages, and Clause (4) terminates the process if the end of the two streams is reached.

This type of priority queue is used in the implementation of the disk-arm scheduler described below. A more efficient priority queue can be obtained using balanced trees.

4.11. A spooler

Arvind and Brock [3] describe an implementation of a priority printer manager. It manages one printer by maintaining two queues, a fast queue for small files, and a slow queue for large files. It prints the files on the printer, giving priority to the fast queue, and sends back a confirmation when the printing is completed.

Hewitt et al. [26] describe an Actors implementation of a hard-copy server. It manages two printers, but does so with no priority considerations.

Program 13 combines the functionality of the two systems, and does so in a more concise and elegant form. Instead of two queues is manages a priority queue, which provides a more refined response. It follows the approach of Arvind and Brock [3] and treats system I/O in a side-effect free way, by identifying I/O devices with the streams they produce or consume. It is initialized with communication streams corresponding to its external I/O devices: in our example two printers and an interface to the users. The users can share the spooler by merging their streams.

The users of the system send messages *print(File, Response)* to the spooler. A simple filter wraps these messages with *enqueue*, adds to them their priority, which is the size of the file, and sends them to a priority queue. Upon termination it puts two *halt* messages in the queue, one for each printer.

A printer-controller process sends the queue a stream of *dequeue(print(File, Response))* messages, with *File* and *Response* undetermined. For each such message the printer waits for *File* to be determined, prints it, and unifies Response with *true*. The printer-controller terminates and closes its I/O streams when *File=halt*. In the above example the *halt* message is sent with priority 0, which means that the printers will halt immediately after finishing printing the current file, and that all files awaiting printing in the queue will be flushed. This behavior can be modified by changing the priority of the *halt* message; setting it to 1000 will cause the printers to halt only after all files in the queue of size<1000 have been printed.

4.12. An implementation of the SCAN disk-arm scheduling algorithm

The goal of a disk-arm scheduler is to satisfy disk I/O requests with minimal arm movements. The simplest algorithm is to serve the next I/O request which refers to the track closest to the current arm position. This algorithm may result in unbounded waiting — a disk I/O request may be postponed indefinitely. The SCAN algorithm tries to minimize the arm movement, while guaranteeing bounded waiting. The algorithm reads as follows:

```
spooler((P1, P2), U) :-
    filter(U?, U1),
    printer(D1, P1), printer(D2, P2), merge(D1?, D2?, D),
    queue(U1?, D?).

filter([print(F, R)|S], [enqueue(print(F, R), Size)|S1]) :-
    length(F, Size) | filter(S?, S1).
filter([], [enqueue(print(halt, _),0), enqueue(print(halt, _),0)]).

printer([dequeue(print(File, Response))|S], P) :-
    printer1(File?, Response, S, P).

printer1(File, true, [dequeue(print(File1, R1))|S], P) :-
    dif(File, halt), print(File, P, P1) | printer1(File1?, R1, S, P1).
printer1(halt, true, [], []).

print([], [end_of_file|P], P).
print([X|Xs], [X|P1], P) :-
        print(Xs?, P1, P).

queue(Es, Ds) :-
        See Program 12.
```

Program 13: A spooler

"while there remain requests in the current direction, the disk arm continues to move in that direction, serving the request(s) at the nearest cylinder; if there are no pending requests in that direction (possibly because an edge of the disk surface has been encountered), the arm direction changes, and the disk arm begins its sweep across the surface in the opposite direction". (from [28] p.94).

The disk scheduler has two input streams — a stream of I/O requests from the user(s) of the disk, and a stream of partially determined message from the disk itself. The scheduler has two priority queues, represented as lists: one for requests to be served at the upsweep of the arm, and one for the requests o be served at the downsweep. It represents the arm state with the pair (*Track*, *Direction*), where *Track* is the current track number, and *Direction* is *up* or *down*.

The disk scheduler is invoked with the goal:

disk_scheduler(DiskS?, UserS?)

where *UserS* is a stream of I/O requests from the user(s) of the disk, and *DiskS* is a

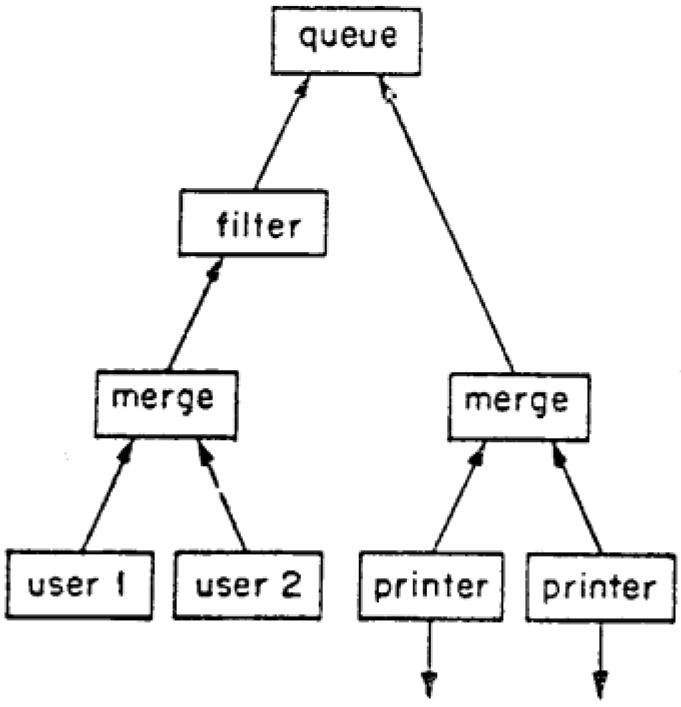

Figure 9: A spooler

stream of partially determined messages from the disk controller. I/O requests are of the form *io*(*Track*, *Args*), where *Track* is the track number and *Args* contain all other necessary information.

The first step of the scheduler is to initialize itself with two empty queues and the arm positioned on track 0, ready for an upsweep; this is done with Clause (0).

After the initialization, the scheduler proceeds using three clauses:

- Clause (1) handles requests from the disk. If such a request is ready in the disk stream, the scheduler tries to dequeue the next request from one of the queues. If successful, that request is unified with the disk request, and the scheduler iterates with the rest of the disk stream, the new queues, and the new arm state. The dequeue operation fails if both queues are empty
- Clause (2) handles requests from the user. If an I/O request is received from the user it is enqueued in one of the queues, and the scheduler iterates with the rest of the user stream and the new queues.
- Clause (3) terminates the scheduler, if the end of the user stream is reached and if both queues are empty. Upon termination, the scheduler send a 'halt' message to the disk controller.

```
(0) disk_scheduler(DiskS, UserS) :-
        disk_scheduler(DiskS?, UserS?, ([], []), (0, up)).

(1) disk_scheduler([Request|DiskS], UserS, Queues, ArmState) :-
        dequeue(Request, Queues, Queues1, ArmState, ArmState1) |
        disk_scheduler(DiskS?, UserS, Queues1, ArmState1).
(2) disk_scheduler(DiskS, [Request|UserS], Queues, ArmState) :-
        enqueue(Request, Queues, Queues1, ArmState) |
        disk_scheduler(DiskS, UserS?, Queues1, ArmState).
(3) disk_scheduler([io(0, halt)|_], [], ([], []), _).

(1) dequeue(io(T,X), ([io(T,X)|UpQ],[]), (UpQ,[]), _, (T,up)).
(2) dequeue(io(T,X), ([io(T,X)|UpQ],DownQ), (UpQ,DownQ), (_,up), (T,up)).
(3) dequeue(io(T,X), ([], [io(T,X)|DownQ]), ([],DownQ), _, (T,down)).
(4) dequeue(io(T,X), (UpQ,[io(T,X)|DownQ]), (UpQ,DownQ), (_,down), (T,down)).

(1) enqueue(io(T, Args), (UpQ, DownQ), ([io(T, Args)|UpQ], DownQ), (T, down)).
(2) enqueue(io(T, Args), (UpQ, DownQ), (UpQ, [io(T, Args)|DownQ]), (T, up)).
(3) enqueue(io(T, Args), (UpQ, DownQ), (UpQ1, DownQ), (T1, Dir)) :-
        T>T1 | insert(io(T, Args), UpQ, UpQ1, up).
(4) enqueue(io(T, Args), (UpQ, DownQ), (UpQ, DownQ1), (T1, Dir)) :-
        T<T1 | insert(io(T, Args), DownQ, DownQ1, down).

(1) insert(io(T, X), [], [io(T, X)], _).
(2) insert(io(T, X), [io(T1, X1)|Q], [io(T, X), io(T1, X1)|Q], up) :-
        T<T1 | true.
(3) insert(io(T, X), [io(T1, X1)|Q], [io(T, X), io(T1, X1)|Q], down) :-
        T≥T1 | true.
(4) insert(io(T, X), [io(T1, X1)|Q], [io(T1, X1)|Q1], up) :-
        T≥T1 | insert(io(T, X), Q, Q1, up).
(5) insert(io(T, X), [io(T1, X1)|Q], [io(T1, X1)|Q1], down) :-
        T<T1 | insert(io(T, X), Q, Q1, down).
```

Program 14: A SCAN disk-arm scheduler

The *dequeue* procedure has clauses for each of the following four cases:

- Clause (1): If *DownQ* is empty then it dequeues the first request in *UpQ*, and changes the new state is an upsweep, were the track number is the track of the I/O request.
- Clause (2): If the arm is on the upsweep and *UpQ* is nonempty then it

dequeues the first request in *UpQ*. The new state is as in the previous clause.

- Clauses (3) and (4): Are the symmetric cases for *DownQ*.

Note that no clause applies if both queues are empty, hence in such a case the *dequeue* procedure fails. Since the disk scheduler invokes *dequeue* as a guard, it must wait in this case for the next user request, and use Clause (3) to enqueue it. If such a request is received and enqueued then in the next iteration the disk request can be served.

The *enqueue* procedure also handles four cases. If the I/O request refers to the current arm track, than according to the SCAN algorithm it must be postponed to the next sweep. Clauses (1) and (2) handle this situation for the upsweep and downsweep cases. If the request refers to a track number larger than the current track, than it is inserted to *UpQ* by Clause (3), otherwise it is inserted to *DownQ*, by Clause (4).

The insertion operation is a slight augmentation to the priority queue insertion of Program 12.

To test the disk scheduler, we have implemented a simulator for a 10-track disk controller. The controller sends a stream of partially determined I/O requests, and, when the arguments of the previous request becomes determined, it serves it and sends the next request.

```
(0) disk_controller([io(Track, Args)|S]) :-
        disk_controller(Track?, Args?, S, [0, 0, 0, 0, 0, 0, 0, 0, 0, 0]).

(1) disk_controller(Track, Args, [io(Track1, Args1)|S], D) :-
        disk(Track, Args, D, D1) | disk_controller(Track1?, Args1?, S, D1).
(2) disk_controller(_, halt, [], _).

(1) disk(_, (_, false), [], []).
(2) disk(0, (read(X), true), [X|D], [X|D]).
(3) disk(0, (write(X), true), [_|D], [X|D]).
(4) disk(N1, IO, [X|D], [X|D1]) :-
        plus(N, 1, N1) | disk(N, IO, D, D1).
```

Program 15: A simulator of a 10-track disk controller

When invoked with a stream S, the controller initializes the disk content and sends the first request using Clause (0). It then iterates with Clause (1), serving the previous I/O request and sending the next partially determined request, until a *halt* message is received, upon which it closes its output stream and terminates, using Clause (2).

The disk simulator assumes that the arguments of an I/O request are pairs (*Operation*, *ResultCode*), where the operations are *read*(*X*) and *write*(*X*). On *read*(*X*) Clause (2) unifies X with the content of the requested track number. On *write*(*X*) Clause (3) replaces the requested track content with X. The *ResultCode* is unified with *true* if the operation completed successfully (Clauses (2) and (3)), and with *false* otherwise (Clause (1)). An example of an unsuccessful completion is when the requested track number exceeds the size of the disk.

An additional interface that transforms normal terminal I/O into stream I/O was developed, and is shown in the appendix. Together these programs were debugged and tested using the interpreter described in Section 5.

4.13. Dataflow computations and cyclic communication streams

Cyclic communication streams are commonly used in dataflow languages. A classical example is a dataflow program that computes the Fibbonachi series, by cycling the elements and adding each two consecutive elements to generate the following element [44]. A similar (nonterminating) Concurrent Prolog program is shown below.

```
fib(S) :- fib1([0, 1|S]).

fib1([X1, X2, X3|Xs]) :- plus(X1, X2, X3) , fib1([X2, X3|Xs]).
```

Program 16: Generating the Fibbonachi series

The Fibbonachi program does not use explicitly cyclic communication stream. A logic program that uses cyclic communication streams was developed by van Emden and de Lucena [16], as solution to Hamming's problem [15] — generate all multiples of 2, 3, and 5 without repetition. Their solution, adapted to Concurrent Prolog, was tested using our interpreter and was found to work. Here we apply the same programming technique to another problem — finding the connected components of a graph. We associate with each node in the graph a distinct integer number. The name of a connected component is the smallest number of any node in the component. The algorithm, for a graph with vertices V is:

```
For each node n in V do
    Set Xn to n.
    Repeat |V| times:
        Send Xn to all nodes adjacent to n,
        Receive a number from every adjacent node,
        Let Xn' be the minimum of Xn and the numbers received.
        Set Xn to Xn'.
    Set n's connected component number to Xn.
```

The algorithm is not very efficient. For an n-vertex graph its computation depth is in the order of n, and its length is in the order of n^2 [40]. This means, roughly, that given n processors, the algorithm runs in linear time. Better parallel algorithms are known [41], but this algorithm — and its implementation — are certainly the simplest.

Program 17 assumes a specialized adjacency list representation of the graph. With each node N we associate a stream variable Xn. The graph is represented as a list of n triples (N, Xn, As), where N is the node number, Xn is its associated stream variable, and As is a list of the stream variables associated with the nodes adjacent to N. Given this graph, the program computes a list of pairs (N, C), where C is the name of the connected component of the node N.

```
(1) cc(Graph, CList) :-
        cc(Graph, Graph, CList).

(2) cc(Graph, [(N, [N|Xn], As)|Gs], [(N, C)|Cs]) :-
        node(Graph, N, Xn, As, C),
        cc(Graph, Gs, Cs).
(3) cc(_, [], []).

(1) node([_|G], Xn, [Xn1|Xns], As, C) :-
        min(Xn, As?, Xn1, As1), node(G, Xn1, Xns, As1, C).
(2) node([], C, [], _, C).

(1) min(Xn, [[B|Bs]|As], Xn1, [Bs|As1]) :-
        lt(Xn, B) | min(Xn, As?, Xn1, As1).
(2) min(Xn, [[B|Bs]|As], Xn1, [Bs|As1]) :-
        le(B, Xn) | min(B, As?, Xn1, As1).
(3) min(Xn, [], Xn, []).
```

Program 17: Computing the connected components of a graph

Clauses (1)-(3) spawn a node process for each node in V; note that they generates the stream of pairs (N, C) before the component name C of each node is determined.

Each node process has four arguments. The first is the graph itself, which it uses to count |V| iterations. Following are the Xn variable explained above, the list of streams of adjacent nodes, and the node's final component number. The node process iterates |V| times, using Clause (1), and on each iteration performs the operations described above.

The *min* process extracts the smallest element among all the first elements of the adjacent streams and the current node number, and returns a list of the rest of the streams.

For example, if we invoke the program on the 7-vertex graph in which 1 is connected to 2 and 3, 2 is connected to 4, 5 is connected to no one, and 6 is connected to itself and to 7, we get the following result:

```
| ?- solve(cc([(1, X1, [X2, X3]), (2, X2, [X1, X4]), (3, X3, [X1]), (4, X4, [X2]),
|          (5, X5, []), (6, X6, [X6, X7]), (7, X7, [X6])], Cs)).
|
Cs = [(1, 1), (2, 1), (3, 1), (4, 1), (5, 5), (6, 6), (7, 6)],
X1 = [1, 1, 1, 1, 1, 1, 1, 1],
X2 = [2, 1, 1, 1, 1, 1, 1, 1],
X3 = [3, 1, 1, 1, 1, 1, 1, 1],
X4 = [4, 2, 1, 1, 1, 1, 1, 1],
X5 = [5, 5, 5, 5, 5, 5, 5, 5],
X6 = [6, 6, 6, 6, 6, 6, 6, 6],
X7 = [7, 6, 6, 6, 6, 6, 6, 6]
```

which shows, in addition to the component numbers, also at which cycle each node discovered its correct component name. We see that after three cycles every node knew its final component name.

A program for distributed array relaxation can be obtained following the same technique. Adjacent array cells are connected by communication streams, and a procedure that computes some average function replaces the *min* procedure. The termination condition of an array relaxation program is more difficult: a tree-shaped network of control processes that monitor the progress of each cell is spawned. Each leaf control-process propagates *continue* or *halt* tokens up the tree, depending on whether the cell it monitors has converged. Each internal control-processes or's the *continue* tokens; if all control-processes agree that it is time to halt, a *halt* message is sent to every cell process, and the current value of the cell is its output.

5. A centralized Concurrent Prolog machine and its Prolog implementation

The centralized Concurrent Prolog interpreter in Program 18 is a simplification of the interpreter in the appendix, which we used to develop and debug the programs described in the paper. It is 44 lines of code long, and it performs about 135 process reductions per CPU second on a DEC 2060.

5.1. Control

The interpreter maintains two data structures: a queue of Concurrent Prolog processes, and a deadlock indicator. When invoked with a system of processes, the interpreter schedules the processes in the queue, sets the deadlock indicator on, and appends to the tail of the queue a cycle marker, used to detect deadlock. It then iterates, dequeueing a process, reducing it, and inserting the reduced system of processes into the queue, according to the scheduling policy. If a process cannot be reduced then it is enqueued back. If no process in the queue can be reduced then the system is deadlocked, and the interpreter fails.

Each iteration proceeds as follows: If the queue contains only the cycle marker, then the interpreter terminates. Otherwise it dequeues an element from the queue. If that element is the cycle marker and the deadlock indicator is off it enqueues the cycle marker back, sets the deadlock indicator on, and iterates. If the cycle marker is encountered when the deadlock indicator is on the computation fails.

If the dequeued element is a process then it attempts to reduce it, as explained below, and if the reduction is successful, it inserts the newly created processes into the queue, according to the scheduling policy, and iterates. If the reduction is not successful then it enqueues the process and iterates.

To reduce a process A, the interpreter sequentially searches through the clauses $A1 :- G \mid B$ in the program, trying to unify A with $A1$ and, if successful, to solve the guard by calling itself recursively with G. If it finds such a clause then the reduced system is B. Standard indexing mechanisms can be used to focus the search for a unifiable clause.

The overhead of a scheduling policy is measured by the ratio of unsuccessful vs. successful reductions. We have experimented with a variety of scheduling policies, including depth-first, breadth-first, and mixed. In depth first scheduling, the reduced system is added to the head of the queue. This policy typically has less overhead, but it is not *and*-fair, since if the computation is nonterminating then some processes may

```
solve(A) :-
        system(A), !, A.
solve(A) :-
        schedule(A, X, X, Head, [cycle|Tail]),
        solve(Head, Tail, deadlock).

solve([cycle], [], _) :- !.
solve([cycle|Head], [cycle|Tail], nodeadlock) :- !,
        solve(Head, Tail, deadlock).
solve([A|Head], Tail, DL) :-
        system(A), !, A,
        solve(Head, Tail, nodeadlock).
solve([A|Head], Tail, DL) :-
        reduce(A, B, DL, DL1),
        schedule(B, Head, Tail, NewHead, NewTail),
        !, solve(NewHead, NewTail, DL1).

reduce(A, B, _, nodeadlock) :-
        guarded_clause(A, G, B),
        solve(G), !.
reduce(A, suspended(A), DL, DL).

guarded_clause(A, G, B) :-
        guarded_clause(A, B1), find_guard(B1, G, B).

find_guard((A|B), A, B) :- !.
find_guard(A, true, A).

schedule(true, Head, Tail, Head, Tail) :- !.
schedule(suspended(A), Head, [A|Tail], Head, Tail) :- !.
schedule((A, B), Head, Tail, Head2, Tail2) :- !,
        schedule(A, Head, Tail, Head1, Tail1),
        schedule(B, Head1, Tail1,.Head2, Tail2).
schedule(A, Head, [A|Tail], Head, Tail).
```

Program 18: A centralized Concurrent Prolog interpreter

never reach the head of the queue. In breadth first scheduling, implemented in Program 18, the reduced system is added to the tail of the queue. This scheduling policy is fair, but may have a rather severe overhead. The scheduling overhead in the examples we tried ranged from 0 to 100 percent. For example, in the simulator of the multiprocessor reduction machine in Program 10 we used a breadth first scheduler for the processors, but a depth first scheduler for the communication processes -- the merge processes and the queue. With this policy we obtained a scheduling overhead of 30 percent.

One drawback of this interpretation algorithm is that it does not incorporating true or-parallelism, or, in other words, that it is not *or*-fair. If there are both nonterminating and terminating guard systems, the interpreter may fail to solve any of the guards. This interpreter, therefore, does not execute correctly the program in Figure 6, which implements a shell ("exec") program that handles an *abort* interrupt.

If loaded into Prolog-10, or any compatible Prolog implementation, Program 18, together with the implementation of unification in Program 19 and the *guarded_clause* procedure described below should execute all Concurrent Prolog programs shown in the paper. The only additions necessary are a definition of the read-only symbol '?' as a postfix unary functor,and a definition of the predicate *system*(*X*), which succeeds if X is a Prolog system predicate. A definition of the Concurrent Prolog "built-in" predicates *wait*(_, _) and *dif*(_, _) and of some useful I/O routines is included in the appendix.

The full Concurrent Prolog interpreter shown in the appendix includes also trace and statistics packages. When compiled, it performs around 100 reductions per CPU second on a DECsystem 2060.

5.2. Unification

The interpreter is so simple partly because it uses Prolog's binding environment in the implementation of the extended unification algorithm, shown in Program 19. To understand this program note that semi-colon is Prolog's or; the predicate = is defined by the unit clause X=X; *nonvar*(*X*) and *var*(*X*) are built-in Prolog meta-logical predicates, which succeed if X is instantiated or not instantiated to a non-variable term, respectively; X=..Y explodes the term X into a list Y whose *car* is the main functor of X and *cdr* is the list of arguments of X.

Clause (1) deals with the case where one of the terms is a variable, and defaults to Prolog to do the unification; Clauses (2) and (3) deal with unification of read-only terms; Clauses (4), (5), and (6) recurse on the structure of the term, if previous clauses do not apply.

Using this unification procedure and Prolog's built-in predicates *clause* and *functor* we can invoke clauses without instantiating read-only variables, with the procedure *guarded_clause*:

```
(1) unify(X, Y) :-
        ( var(X) ; var(Y) ), !, X=Y.
(2) unify(X?, Y) :- !,
        nonvar(X), unify(X, Y).
(3) unify(X, Y?) :- !,
        nonvar(Y), unify(X, Y).
(4) unify([X|Xs], [Y|Ys]) :- !,
        unify(X, Y), unify(Xs, Ys).
(5) unify([], []) :- !.
(6) unify(X, Y) :-
        X=..[F|Xs], Y=..[F|Ys], unify(Xs, Ys).
```

Program 19: Unification of terms with read-only annotations.

```
guarded_clause(A, B) :-
    functor(A, F, N), functor(A1, F, N),
    clause(A1, B),
    unify(A, A1).
```

functor(A, F, N) names the relation "A is a term whose principal functor has name F and arity N". When invoked with its first argument A instantiated to a term it unifies F and N with the name and functor of A. When invoked with F and N instantiated it unifies A with a most general term whose principal functor has name F and arity N. *clause(A, B)* unifies B with a clause whose head unifies with A; it requires A to be instantiated to a nonvariable term.

5.3. Optimizations

Several optimizations are essential to make this interpreter a more practical tool. The first is to incorporate the unification of read-only terms in the underlying unification algorithm of Prolog. This can be done easily if one has an access to the Prolog sources; I do not have access to the Prolog-10 sources, but I have access to people who have access to the Prolog-10 sources, who may be kind enough to incorporate this extension. I expect this improvement to result in at least a 5-fold speedup.

Another essential optimization is the reduction of scheduling overhead by elimination of busy waiting. This can be done using a technique invented by Alain Colmerauer, and incorporated in Prolog-II — an implementation of Prolog on the Apple-II computer — to implement the "freeze" predicate [13].

The technique is extremely simple and elegant, and eschews the need for elaborate hardware mechanisms such as associative memory to implement the dataflow-based synchronization mechanisms of Concurrent Prolog. If one or more processes are waiting for some variable's principal functor to be determined, then the memory cell representing this variable is temporarily assigned to a pointer to the list of these processes. When another process unifies this variable with a non-variable term, it first remove this pointer, instantiate the variable, and activates all the processes on the waiting list.

A third important improvement is saving the state of locked subsystems. Currently, if a subsystem is locked, then the interpreter executing it fails, its state is lost, and its execution starts from scratch the next time the process that invoked this subsystem is scheduled. It would be better to save the state of such a system; but this requires some additional bookkeeping to garbage-collect irrelevant subsystems.

6. Comparison with other concurrent programming languages

"**Occam's Razor** [William of *Ockham*]: A scientific and philosophical rule that entities should not be multiplies unnecessarily which is interpreted as requiring that the simplest of competing theories be preferred to the more complex or that explanations of unknown phenomenon be sough in terms of known quantities"

— *Webster's New Collegiate Dictionary.*

6.1. The relational language of Clark and Gregory

The roots of Concurrent Prolog can probably be traced back to the work of Kahn and MacQueen [31]. They have shown that concurrent programming can be modeled naturally by processes computing relations over streams. Van Emden and de Lucena [16] provided a translation of the model of Kahn and MacQueen into the formalism of logic programming. However, both the language of Kahn and MacQueen and its logical counterpart are deterministic.

Clark and Gregory extended the approach of van Emden and de Lucena to indeterminate computations with the concept of a guarded-clause. Their relational language can implement indeterminate stream merge, for example. The subset of Concurrent Prolog described in this paper is a result of an attempt to generalize and clean up the relational language of Clark and Gregory. Several differences can be found between the two.

The language of Clark and Gregory requires that a guard be ground before an attempt is made to solve it; hence it is immaterial whether it is solved sequentially or in parallel. Our guards may contain general systems; there is no restriction that a guard system be ground before it is invoked. This enables Concurrent Prolog to support a hierarchy of concurrent systems, which is essential for implementing operating systems. See for example the shell program on Page 29.

The language of Clark and Gregory restricts the variables shared between processes to be of type "stream", and requires that each such variable will have at most one producer; this has to be verified at compile time, based on mode declarations and variable annotations. The producer and consumer annotations are inherited, so, for example, a receiver of a message cannot respond to it by instantiating an undetermined variable in it. Hence their language does not support programming with partially determined messages, a key concept in Concurrent Prolog.

Concurrent Prolog alleviates all these restrictions since it incorporates a simpler and more basic synchronization mechanism. Instead of mode declarations and consumer and producer annotations, Concurrent Prolog has one synchronization primitive: the read-only annotation. This mechanism is expressive enough to achieve behaviors induced by the mechanisms of the relational language, and more. The generality of this mechanism enables us to share variable of any type between processes, not only streams. It also enables the use of partially determined messages. In addition, our synchronization mechanism can be implemented very efficiently, in contrast to the requirement that an invoked clause should have a ground guard.

We view the read-only annotation as the major contribution of Concurrent Prolog over the language of Clark and Gregory.

The requirement that a shared variable has only one producer is not incorporated in Concurrent Prolog, and we think it is is not essential in the relational language as well. The reason is that two producers of the same variable are conjunctive goals. A conjunctive goal with a shared variable can succeed only if all its conjuncts agree on what value this variable should take. A process instantiates a shared variable only after it is committed to that instantiation, i.e. after it solves the guard of a clause. Hence if two processes attempt to instantiate a variable to mutually non-unifiable terms then they are committed to non-unifiable solutions of the conjunctive goal; hence the conjunctive goal should fail, which is what happens in Concurrent Prolog. Since goal failure is an ordinary phenomenon in a computation of a logic program, this case deserves no special treatment.

It should be mentioned that read-only mode declarations, such as

mode merge(?, ?, _)

can be used as a shorthand for annotating the first two argument of *merge* as read-only in every call *merge* in the program. A preprocessor can use such declarations to install the appropriate annotations in the Concurrent Prolog code prior to its compilation.

Another extention of Prolog to a parallel programming language, called Epilog, was proposed by Wise [48]. Apparently, Epilog is not concerned with solving concurrent programming problems, but only with the parallel execution of logic programs. Nevertheless, it has some constructs in common with the Relational Language and Concurrent Prolog. The most apparent difference between Concurrent Prolog and Epilog is in the design methodology: we strive to find the minimal set of basic control constructs necessary to express concurrent computations, while Epilog enjoys a proliferation of these.

6.2. Concurrent functional programming languages

There is a natural mapping from any (first order) functional program to a relational program: replace every n-ary function symbol by an $n+1$-ary relation symbol, annotate the output variable of the relation as read-only in its occurence as an input, and replace function composition by process conjunction. As discussed earlier, this translation achieves the effect of programming with Lenient-*cons* [18]. The effect of *frons* [19], can also be achieved in Concurrent Prolog, using stream merge.

In comparison, the translation from relational to functional programs is not so straightforward. Since the output of a process can be named explicitly, complex communication patterns between processes can be specified, including cyclic ones. The basic computational model of functional languages supports only hierarchical communication — between child and parent processes. Functional programming languages must add extraneous features to their basic computational model to achieve flexibility in communication.

One approach to the problem was taken by Harel and Nehab in Concurrent And/Or programs [23], which extended the functional And/Or programming language with a CSP-like communication mechanism.

Another approach was taken by Johnson [30], who describe a system of processes as a system of functions computing a solution to a set of simultaneous equations. The result is quite similar to Concurrent Prolog: each equation corresponds to a goal with one output variable, and a set of simultaneous equations corresponds to a conjunctive goal.

It may be the case that by adding additional features to functional languages, such as Lenient-*cons* and simultaneous equations, one can get closer and closer to the expressiveness of a relational language; but I do not see a reason not to work directly with the more expressive formalism.

The preference of relations over functions in concurrent programming is evident also in the theoretical treatments of this issue, for example in [7, 37, 38, 42].

6.3. Dataflow languages

The synchronization mechanism of Concurrent Prolog is very similar to that of dataflow languages [1]: a process suspended on undetermined read-only variables is analogous to an operator waiting for its arguments to arrive. Concurrent Prolog, however, lends itself to more concise and elegant programming. For example, dataflow languages extensively use the "let" construct, which achieves only a subset of the effects of unification, and does so in a cruder and more verbose way.

Another difference is that dataflow languages are deterministic, and hence must introduce *merge* as a built-in operator. According to Bryant and Dennis's [9] version of Occam's Razor mentioned earlier in Section 3, this constitutes an evidence in favor of Concurrent Prolog and against the dataflow languages. One immediate implication of the determinacy of dataflow languages it that they must introduce a new *merge* operator for every scheduling policy described in Section 4; on the other hand all of these are user programmable in Concurrent Prolog.

Another important difference between Concurrent Prolog and dataflow languages is related to monitors, and is discussed in the next section.

6.4. Monitors

Dataflow languages are a cleaner computational model than conventional concurrent programming languages such as Concurrent Pascal [6], and CSP/k [28], which use monitors [27]. In spite of that, programs that implement shared resources are simpler and more readable when monitors are used.

To share a resource, dataflow languages use tagged-merge operators, that merge requests from several processes to the shared resource and tag them with the origin of the request. After the resource receives the request, it replies back to the sender using the tags on the message. Such a communication network includes both tagged-merge and tagged-split operators, and cannot be modified easily at runtime. A solution to this problem, termed *managers*, is admitted by its inventors to obstruct the initial simplicity and clarity of the dataflow model [3].

The crux of the problem is that the resource has to know who issued the request in order to respond to it. A monitor can execute a monitor call without knowing the identity of the caller explicitly, since the caller can find the response to its monitor call by inspecting the appropriate arguments in the call. In implementation terms, a monitor call contains the identity of the caller in the memory address of its result arguments. The monitor can respond to the call by placing the desired values in those memory addresses, thus avoiding the communication protocol overhead a dataflow language requires in order to ship the response back to the sender.

One may argue that the difference between the two approaches is only in the level at which the problem is solved. Monitors solve the problem of responding to requests at the implementation level, whereas dataflow languages solve it at the programming level. We agree, but find this difference crucial for the convenience and flexibility of the language.

The Concurrent Prolog solution to implementing shared resources enjoys the benefits of both worlds, due to the concept of partially determined messages. A partially determined message is sent to a shared resource via merged streams, just as in a dataflow language; but the shared resource responds to it by placing values in undetermined variables in the message, just as in a monitor call. Hence our examples use only merge, not tagged-merge, and consequently do not use tagged-split either.

In implementation terms, the recipient of a partially determined message can respond to it without knowing the explicit identity of the sender, since its identity is hidden in the address of the undetermined variables in the message.

This analogy provides an interesting distinction between procedure calls and monitor calls in Concurrent Prolog. In monitor-based programming languages a procedure call and a monitor call are two basic, mutually irreducible operations. On the other hand in Concurrent Prolog a procedure call, i.e. a process invocation, is a basic operation, executed directly by the Concurrent Prolog machine, whereas a monitor call is a data-structure, which is sent, received and processed by the Concurrent Prolog software. This may be another instance in which Bryant, Dennis, and Occam's razor applies.

6.5. Actors

The Actors model [25, 35], is also closely related to Concurrent Prolog, as our use of Actors jargon to explain Concurrent Prolog programs suggests. One similarity is the "light-weight" and dynamic nature of Concurrent Prolog processes and the actors of an

Actor system. Another is the use of pattern-matching to select and construct the response to a message.

The main difference between the two is the simplicity and clarity of the computational model. The Actor model is centered around an operational semantics, which freely mixes object-level and meta-level operations, and tends to be described in a rather loose way.

Some say: "Prolog is Planner [24] done right". If analogy was a valid inference rule, one could conclude: "Concurrent Prolog is Actors done right".

7. Relation to sequential Prolog

Initially, our goal was to extend Prolog to a concurrent programming language, and one of our design criteria was to properly contain sequential Prolog. However, as our research progressed, we have realized that this heads-on approach is not appropriate, since Prolog looks the way it does precisely because it was designed to run efficiently on a sequential von Neumann machine. As a result, it suffers from many of the illnesses that make conventional programming languages not suitable for new architectures.

For example, the state of the computation of sequential Prolog is very complex. In addition to the binding environment and the stack, also maintained by other programming languages, Prolog maintains the backtrack point for each goal on the stack, and a trail-stack that says which variable bindings should be reset upon backtracking.

Hence we currently believe that properly containing Prolog may not be a desirable goal. However, since in the meantime we would like to see Concurrent Prolog as the systems programming language of a von Neumann Prolog machine, there is a need to find some way to integrate the two. One possibility is already available in our Concurrent Prolog interpreter: to call Prolog from Concurrent Prolog we use the predicate *call*(X), which defaults to Prolog to solve X. Since both sequential and Concurrent Prolog maintain the same binding environment, there are no interface problems.

Ultimately, there should be a more direct way to incorporate in Concurrent Prolog some of the powerful properties of sequential Prolog it currently lacks. Kowalski [32] drew a distinction between two types of nondeterminism: *don't-care nondeterminism* and *don't-know nondeterminism*. In other circles the first is referred to as indeterminacy, while the second simply as nondeterminism. Concurrent Prolog incorporates don't-care nondeterminism, but not don't-know nondeterminism. The latter is simulated in sequential Prolog by sequential search and backtracking.

For example, the following sequential Prolog program for finding an element in the intersection of two lists

```
intersect(X, L1, L2) :- member(X, L1), member(X, L2)
```

would not work correctly in Concurrent Prolog, since the first process to find a value for X will commit to it and not backtrack, even if its choice does not suit the other process. This, of course, does not mean that Concurrent Prolog cannot implement list intersection; but to do so it needs to iterate explicitly on at least one of the two lists:

```
intersect(X, [X|L1], L2) :- member(X, L2) | true.
intersect(X, [_|L1], L2) :- intersect(X, L1, L2) | true.
```

The ability to talk about process failure is another important extention, essential to make Concurrent Prolog a practical systems programming language. For example, we would like the shell process to report to the user when the execution of his command has failed, and would like the operating system to reboot itself upon a software crash.

Adding the ability to talk about process failure is similar to extending sequential Prolog with negation-as-failure [10], but not quite the same. Earlier we claimed that *commit* is a cleaned-up *cut*. This is due to its symmetry: since all guards are assumed to be executing in parallel, the first to reach the *commit* kills alternative computation paths both above and below it. In contrast, *cut* kills only alternatives left below it.

The relation of sequential Prolog's cut to Concurrent Prolog's commit is similar to the relation of conventional if-then-else to Dijkstra's guarded-command. The same argument Dijkstra uses against if-then-else and in favor of the guarded-command is applicable to cut and commit: the lack of symmetry and the reliance of the default in the former, versus symmetry and explicit conditions in the latter.

One consequence of this difference is the inability to implement negation-as-failure using commit. The standard implementation of negation-as-failure in sequential Prolog

```
not(X) :- X, !, fail.
not(X).
```

is essentially an if-the-else. It reads: "if X succeeds then fail, else succeed". The corresponding concurrent Prolog program would not have the desired effect, since the second clause may succeed even when X is solvable.

Cut is a controversial component in Prolog, and for good reasons. We believe that commit captures the essence of cut, which is the ability commit the execution to the current computation path, without introducing its less desirable features, i.e. the ability to implement if-then-else and implicit negation.

In spite of what said, Concurrent Prolog is not immune to awkward programming practices. For example, if the Concurrent Prolog machine is stable (cf. Page 24), then a weak form of implicit negation can be expressed, a possibility that led Dijkstra to recommend non-stable implementations [15].

This discussion implies that the ability to talk about process failure is a proper extension to Concurrent Prolog. Recent work by Hagia [22], which introduces the notion of levels, may be applicable to the problem. The idea is to partition the procedures in a program into levels, where each procedure can talk about failure of a process only if the procedure it executes is in a lower level. This proposal coincides with the intuitive requirement that a kernel process can talk about the failure of a shell process, and that a shell process can talk about the failure of a user process, but not vice versa.

We believe that having process failure as a primitive concept, and the hierarchical structure of systems created during computation, make Concurrent Prolog a robust systems programming language. Consider the following implications of these properties:

- If a process fails, then its system fails; but this does not mean that the whole computation fails, unless the failing process is in the top-level system.
- If all processes in a system are suspended, then the system is said to be locked; but this is not necessarily a deadlock. A subsystem S1 may be locked at some point in time, and unlocked later, because a sister subsystem S2 instantiated some variable, which appears as read-only in S1. Only if the top-level system is locked, then the situation cannot be cured, and deadlock can be established.
- The scope of interference between processes is restricted to subsystems. If two brother processes commit to unify a shared variable with non-unifiable terms, they fail, and their subsystem fails, but other subsystems are not affected. This scope restriction follows from the requirement that bindings computed by a guard system are made public only after the guard system terminates successfully.

8. Future research

In some sense, both sequential and Concurrent Prolog resemble an assembly language more than a high-level language. We refer to the flat name space of procedures, and to the lack of any type or other declarations whose consistency with the program is checked statically. The roots of this deficiency, however, are sociological, not conceptual. To the contrary, we believe that logic programs lend themselves to modular programming and static analysis at least as well as other types of programs, perhaps even more. The same costumes suggested to cover Prolog's naked body can equally well fit Concurrent Prolog [8, 20, 21].

In addition to extending the language, we consider several other research directions as worth pursuing. First is implementing in Concurrent Prolog a multi-tasking operating system for a single-user computer. We hope that the efficiency of Concurrent Prolog will be sufficient for such a machine, and that any lack thereof will be compensated by the ease in which sophisticated software can be developed in it, as our experience suggests. Considering multi-user operating systems, it seems that the ability to pass streams as arguments in a message may be a sound and simple basis for a capabilities system.

For example, a simple window system was written and debugged in Concurrent Prolog by A. Takeuchi and the author in less than two days, after the definition of the basic screen I/O primitives in Prolog was completed. The system that animates the simulator of a multi-processor Concurrent Prolog machine, described in Section , was implemented on top of the window system by the author in less than one day.

Another research direction is the development of a multi-processor Concurrent Prolog machine. We expect that such a machine will require a rather different architecture from what has been proposed so far. For example, Colmerauer's strategy for implementing suspended processes may eliminate the need for associative memory, a key component in some current dataflow architectures.

Yet another research direction is related to work on systolic algorithms [33]. We find in several Concurrent Prolog programs that once the network of processes is spawned, it starts behaving like a systolic array. This is manifested most clearly in the array relaxation program (a variant of Program 17, not shown in the paper). This suggests the use of Concurrent Prolog as a specification language for systolic chips, with all the ramifications of such a point of view, including a Concurrent-Prolog-to-chip compiler.

Acknowledgements

The author acknowledges Lawrence Byrd, Shimon Cohen, David Harel, Michael Fischer, Steven Gregory, Frank McCabe, Bob Nix, Fernando Pereira, Stan Rosenschein, A. Takeuchi, and David Warren for helpful discussions.

This research began while the author was at Yale University, supported by NSF grant no. MCS80002447, and continued under the support of the DAI project at SRI International, ONR contract no. N000014-80-C-0296.

The experimental part of the research, including the definition of the subset of Concurrent Prolog described in the paper, was carried while the author was a visiting scientist at ICOT — the Institute for New Generation Computer Technology.

The author is a recipient of the Sir Charles Clore fellowship at the Weizmann Institute of Science.

This document was produced on the facilities of the Computer Science Department of Yale University.

I. A Concurrent Prolog interpreter

```
%% Interpreter for a subset of concurrent Prolog.
:- public solve/1,
    reduce/2,
    display_counters/0,
    trace/2,
    wait/2,
    wait/1,
    dif/2.

:- op(450, xf, '?').

:- call((value(initialized, true) ;
   compile([dsutil, 'system.def']),
   % dsutil contains some utilities.
   % system.def contains the definition of the predicate system1(_).
   set(smode, depth_first),
   set(smode(read(_)), breadth_first),
   set(countingset, [reduction(_), suspension(_), system(_)]),
   set(traceset, [reduction(_), suspension(_)]),
   set(initialized, true))).

solve(A) :-
    clear_counters,
    solve(A, 0),
    display_counters.

solve(true, _) :- !.
solve(A, D) :-
    system(A), !, trace(system(D), A), A;
    trace(solve(D), A),
    schedule(A, X, X, Head, [cycle(1)|Tail]),
    solve(Head, Tail, deadlock, D),
    trace(solved(D), A).

solve([cycle(N)], _, _, D) :- !,
    (D=0, writel(['*** cycles: ', N]), nl; true).
solve([cycle(N)|Head], [], deadlock, D) :- !,
    D=0, writel(['*** cycles: ', N]), nl,
    writelnl(['*** Deadlock detected.  Locked processes:'|Head]) ;
    fail.
solve([cycle(N)|Head], [cycle(N1)|Tail], nodeadlock, D) :- !,
    N1 is N+1,
    solve(Head, Tail, deadlock, D).
solve([A|Head], Tail, DL, D) :-
    system(A), !, trace(system(D), A), A,
    solve(Head, Tail, nodeadlock, D).
solve([A|Head], Tail, DL, D) :-
    D1 is D+1,
    trace(call(D1), A),
    reduce(A, B, DL, DL1, D1),
    trace(reduction(D1), (A:-B)),
    schedule(B, Head, Tail, NewHead, NewTail),
    !, solve(NewHead, NewTail, DL1, D).

reduce(A, B, _, nodeadlock, D) :-
    guarded_clause(A, G, B, D),
    trace(try_clause(D), (A:-(G|B))),
    solve(G, D), !.
reduce(A, suspended(A), DL, DL, D) :-
```

```
    trace(suspension(D), A).


reduce(A, B) :-
    guarded_clause(A, G, B, 1),
    solve(G, 1), !.
reduce(A, suspended(A)) :-
    trace(suspension, A).

schedule(true, Head, Tail, Head, Tail) :- !.
schedule(suspended(A), Head, [A|Tail], Head, Tail) :- !.
schedule((A, B), Head, Tail, Head2, Tail2) :-
    value(smode, breadth_first), !,
    schedule(A, Head, Tail, Head1, Tail1),
    schedule(B, Head1, Tail1, Head2, Tail2).
schedule((A, B), Head, Tail, Head2, Tail2) :-
    value(smode, depth_first), !,
    schedule(B, Head, Tail, Head1, Tail1),
    schedule(A, Head1, Tail1, Head2, Tail2).
schedule(A, Head, Tail, [A|Head], Tail) :-
    value(smode(A), depth_first), !.
schedule(A, Head, [A|Tail], Head, Tail) :-
    value(smode(A), breadth_first),  !.
schedule(A, Head, Tail, [A|Head], Tail) :-
    value(smode, depth_first),  !.
schedule(A, Head, [A|Tail], Head, Tail) :-
    value(smode, breadth_first),  !.


guarded_clause(A, G, B, D) :-
    ready_clause(A, B1, D), find_guard(B1, G, B).

find_guard((A|B), A, B) :- !.
find_guard(A, true, A).

ready_clause(A, B, D) :-
    functor(A, F, N), functor(A1, F, N),
    clause(A1, B),
    trace(unify(D), (A, A1)),
    unify(A, A1).
unify(X, Y) :- ( var(X) ; var(Y) ), !, X=Y.
unify(X?, Y) :- !,
    nonvar(X), unify(X, Y).
unify(X, Y?) :- !,
    nonvar(Y), unify(X, Y).
unify([X|Xs], [Y|Ys]) :- !,
    unify(X, Y), unify(Xs, Ys).
unify([], []) :- !.
unify(X, Y) :-
    X=..[F|Xs], Y=..[F|Ys], unify(Xs, Ys).

trace(_, _) :-
    value(trace, off), !.
trace(A, B) :-
    add_counter(A),
    % break(A, B), % add a break package
    value(traceset, S),
    ( member(A, S) ; S=all ),
    writel([A, ': ', B]), nl, !.
trace(_, _).

clear_counters :-
    value(counter(X), Y), Y>0, set(counter(X), 0), fail ; true.

add_counter(A) :-
    value(countingset, S), member(A, S), add1(counter(A), _) ; true.

display_counters :-
    value(countingset, S), member(X, S),
    value(counter(X), Y), Y>0, writel(['# ', X, ': ', Y]), nl, fail ;
    sum_counters.

sum_counters :-
    value(countingset, S),
    setof(Y, (X, S)^(member(X, S), value(counter(X), Y)), S1),
    sum(S1, 0, Total),
    writel(['Total: ', Total]), nl.
```

II. Some Utilities

```
% "Built-in" predicates.

%wait(X, Y) :-
    % wait until X is instantiated, "peel-off" extraneous '?'
    % annotations, and return the result in Y. Useful for interfacing
    % to regular Prolog.
wait(X) :-
    wait(X, _).

wait(X, _) :- var(X), !, fail.
wait(X?, Y) :- !, wait(X, Y).
wait(X, X).


%dif(X, Y) :- X and Y are not unifiable.
dif(X, Y) :-
    (var(X) ; var(Y)), !, fail.
dif(X?, Y) :- !,
    dif(X, Y).
dif(X, Y?) :- !,
    dif(X, Y).
dif([], []) :- !,
    fail.
dif([X|Xs], [Y|Ys]) :- !,
    dif(X, Y) ; dif(Xs, Ys).
dif(X, Y) :-
    X=..[Fx|Xs], Y=..[Fy|Ys],
    ( Fx==Fy ; dif(Xs, Ys) ).
dif(X, Y) :-
    (var(X) ; var(Y)), !, fail.


system(wait(_, _)).
system(wait(_)).
system(dif(_, _)).
system(X) :- system1(X).
```

```
% Interface to tty
instream(Xs) :- % Xs is the current input stream
   read(X) | instream(X, Xs).

instream(end_of_file, []).
instream([], Xs) :-
   instream(Y?, Xs), read(Y).
instream([X|Xs], [X|Ys]) :-
   instream(Xs, Ys).
instream(X, [X|Xs]) :-
   wait(X) | instream(Y?, Xs), read(Y).

outstream([X|Xs]) :- % Xs is the current output stream
   writel(['*** oustream: ', X]), nl | outstream(Xs?).
outstream([]).

wait_write(X, Y) :- % wait for X and output Y to current output stream
   wait(X) | call((write(Y), nl)).

% wrap stream elements with an identifying tag
wrap([], _, []).
wrap([X|Xs], W, [WrappedX|Ys]) :-
   WrappedX=..[W, X] | wrap(Xs?, W, Ys).


lt(X, Y) :- wait(X, X1), wait(Y, Y1) | X1<Y1.
le(X, Y) :- wait(X, X1), wait(Y, Y1) | X1=<Y1.

% lazy evaluator of arithmetic expressions

eval(X, Y) :- wait(X, Y), integer(Y) | true .
eval(X+Y, Z) :- eval(X?, X1), eval(Y?, Y1), plus(X1, Y1, Z).
eval(X-Y, Z) :- eval(X?, X1), eval(Y?, Y1), plus(Z, Y1, X1).
eval(X*Y, Z) :- eval(X?, X1), eval(Y?, Y1), times(X1, Y1, Z).

plus(X, Y, Z) :- wait(X, X1), wait(Y, Y1) | Z is X1+Y1.
plus(X, Y, Z) :- wait(X, X1), wait(Z, Z1) | Y is Z1-X1.
plus(X, Y, Z) :- wait(Y, Y1), wait(Z, Z1) | X is Z1-Y1.
```

```
times(X, Y, Z) :- wait(X, X1), wait(Y, Y1) | Z is X1*Y1.
times(X, Y, Z) :- wait(X, X1), wait(Z, Z1) | Y is Z1/X1.
times(X, Y, Z) :- wait(Y, Y1), wait(Z, Z1) | X is Z1/Y1.

% multiples of 2, 3 and 5 without repititions.
multiples :-
    stream_multiply(2, [1|X?], X2),
    stream_multiply(3, [1|X?], X3),
    stream_multiply(5, [1|X?], X5),
    opmerge(X2?, X3?, X23),
    opmerge(X5?, X23?, X),
    outstream(X?).

stream_multiply(N, [U|X], [V|Z]) :-
    V is N*U | stream_multiply(N, X?, Z).

opmerge([U|X], [U|Y], [U|Z]) :- opmerge(X?, Y?, Z).
opmerge([U|X], [V|Y], [U|Z]) :- lt(U, V) | opmerge(X?, [V|Y], Z).
opmerge([U|X], [V|Y], [V|Z]) :- lt(V, U) | opmerge([U|X], Y?, Z).
```

III. Testing programs

```
% testing the airline reservation system
testdb :-
    solve((
    instream(X),
    usetransact(X?, Y),
    database(Y?, [200, 200, 200, 200, 200])
    )).

% user interface to airline reservation system
usetransact([], []).
usetransact([info(Flight)|Messages], [info(Flight, Seats)|Msg]) :-
    wait_write(Seats, ['Available seats : ', info(Flight, Seats)]),
    usetransact(Messages?, Msg).
usetransact([reserve(Flight, Seats)|Messages],
        [reserve(Flight, Seats, Response)|Msg]) :-
    wait_write(Response, ['Answer : ', reserve(Flight, Seats, Response)]),
    usetransact(Messages?, Msg).

value([X|_], 0, X).
value([_|R], N, V) :- N1 is N-1 | value(R, N1, V).

modify([X|Y], 0, V, [V|Y]).
modify([X|Y], N, V, [X|Y1]) :- N1 is N-1 | modify(Y, N1, V, Y1).


% testing the MSG operating system
testmsg :-
    solve((
    instream(X),
    split(X?, K1, K2),
    msg((K1?, S1), (K2?, S2)),
    wrap(S1?, screen1, S11), outstream(S11?),
    wrap(S2?, screen2, S22), outstream(S22?)
    )).

%testing the spooler
testsp :-
    solve((
    instream(X),
    spooler((P1, P2), X?),
    wrap(P1?, printer1, P11), outstream(P11?),
    wrap(P2?, printer2, P22), outstream(P22?)
    )).

% testing the disk-arm scheduler.
testds :-
    solve((
    instream(X),
    usedisk(X?, User),
    disk(Disk),
    disk_scheduler(Disk?, User?)
    ))

% terminal interface simulating a disk user
usedisk([], []).
usedisk([read(T)|X], [io(T, (read(D), OK))|Y]) :-
```

```
    usedisk(X?, Y), wait_write(OK, disk_read(T, D, OK)).
usedisk([write(T, D)|X], [io(T, (write(D), OK))|Y]) :-
    usedisk(X?, Y), wait_write(OK, disk_write(T, D, OK)).
```

References

[1] William B. Ackerman.
Data flow languages.
IEEE Computer 15(2):15-25, 1982.

[2] K. R. Apt and M. H. van Emden.
Contributions to the Theory of Logic Programming.
Journal of the ACM 29(3):841-863, July, 1982.

[3] Arvind and J. Dean Brock.
Streams and Managers.
In M. Makegawa and L. A. Belady (editors), *Operating Systems Engineering*, pages 452-465. Springer-Verlag, 1982.
Lecture notes in Computer Science no. 143.

[4] D. L. Bowen, L. Byrd, L. M. Pereira, F. C. N. Pereira and D. H. D. Warren.
PROLOG on the DECSystem−10 User's Manual.
Technical Report , Department of Artificial Intelligence, University of Edinburgh, October, 1981.

[5] Kenneth A. Bowen and Robert A. Kowalski.
Amalgamating language and metalanguage in logic programming.
Technical Report 4/81, School ofComputer and Information Science, Syracuse University, June, 1982.

[6] Per Brinch Hansen.
The programming language Concurrent Pascal.
IEEE Transactions on Software Engineering SE-1(2):199-207, 1975.

[7] J. Dean Brock and William B. Ackerman.
Scenarios: A Model of Non-determinate computations.
In Dias and Ramos (editors), *Formalization of Programming Concepts*, pages 252-259. Springer-Verlag, 1981.
Lecture notes in Computer Science no. 107.

[8] M. Bruynooghe.
Adding redundancy to obtain more reliable and readable Prolog programs.
In *Proceedings of the First International Logic Programming Conference*, pages 129-134. ADDP-GIA, Faculte des Sciences de Luminy, Marseille, France, September, 1982.

[9] Randal E. Bryant and Jack B. Dennis.
Concurrent Programming.
In M. Makegawa and L. A. Belady (editors), *Operating Systems Engineering*, pages 426-452. Springer-Verlag, 1982.
Lecture notes in Computer Science no. 143.

[10] Keith L. Clark.
Negation as failure.
In H. Gallaire and J. Minker (editors), *Logic and Data Bases*, . Plenum, 1978.

[11] K. L. Clark and S. Gregory.
A relational language for parallel programming.
In *Proceedings of the ACM Conference on Functional Programming Languages and Computer Architecture*. ACM, October, 1981.

[12] Keith Clark and Stan-Ake Tarnlund.
A first-order theory of data and programs.
In B. Gilchrist (editor), *Information Processing* 77, pages pp.939-944. North-Holland, 1977.

[13] A. Colmerauer, H. Kanui, and M. van Kanegham.
Last steps towards an ultimate Prolog.
In *Proceedings of the Seventh International Joint Conference on Artificial Intelligence*, pages 947-948. IJCAI, 1981.

[14] Dennis M. Richie and Ken Thompson.
The Unix time-sharing system.
Communications of the ACM 17(7):365-375, 1974.

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

[16] M. H. van Emden and G. J. de Lucena.
Predicate logic as a programming language for parallel programming.
In K. L. Clark and S. A. Tarnlund (editors), *Logic Programming*, . Academic Press, 1982.

[17] M. H. van Emden and R. A. Kowalski.
The semantics of predicate logic as a programming language.
Journal of the ACM 23:733-742, October, 1976.

[18] D.P. Friedman and D.S. Wise.
The Impact of Applicative Programming on Multiprocessing.
In *Proceedings of the 1976 International Conference on Parallel Processing*. 1976.

[19] D.P. Friedman and D.S. Wise.
An approach to fair applicative multiprogramming.
In G. Kahn (editors), *Semantics of Concurrent Computations*, . Springer-Verlag, 1979.
Lecture notes in Computer Science no. 70.

[20] K. Furukawa, R. Nakajima, and A. Yonezawa.
Modularization and abstraction in logic programming.
1983.
In preparation.

[21] I. Futo, J. Szeredi.
T–Prolog: A Very High Level Simulation System
SZKI, Budapest, 1981.

[22] Masami Hagia.
Logic programming and inductive definitions.
November, 1982.
Unpublished manuscript, RIMS, Kyoto University.

[23] David Harel and Smadar Nehab.
Concurrent and/or programs: recursion with communication.
Technical Report CS82-09, Weizmann Institute of Science, Department of Applied Mathematics, June, 1982.

[24] Carl Hewitt.
Description and Theoretical Analysis (Using Schemata) of Planner: a Language for Proving Theorems and Manipulating Models in a Robot.
Technical Report TR-258, MIT Artificial Intelligence Lab, 1972.

[25] Carl Hewitt.
A Universal, Modular Actor Formalism for Artificial Intelligence.
In *IJCAI3*. IJCAI, 1973.

[26] Carl Hewitt, Giuseppe Atardi, and Henry Lieberman.
Specfying and proving properties of guardians for distributed systems.
In G. Kahn (editor), *Semantics of Concurrent Computations*, . Springer-Verlag, 1979.
Lecture notes in Computer Science no. 70.

[27] C. A. R. Hoare.
Monitors: an operating systems structuring concept.
Communications of the ACM 17(10):549-557, 1974.

[28] R. C. Holt, G. S. Graham, E. D. Lazowska and M. A. Scott.
Structured Programming with Operating Systems Applications.
Addison Wesley, 1978.

[29] Ingalls, Daniel H.
The smalltalk-76 programming system: design and implementation.
In *Conference Record of the Fifth Annual ACM Symposium on Principles of Programming Languages*, pages 9-16. Association for Computing Machinery, January, 1978.

[30] Steven D. Johnson.
Circuits and Systems: Implementing Communications with Streams.
Technical Report 116, Indiana University, Computer Science Department, October, 1981.

[31] Gilles Kahn and David B. MacQueen.
Coroutines and networks of parallel processes.
In B. Gilchrist (editor), *Information Processing* 77, pages pp.993-998. North-Holland, 1977.

[32] Robert A. Kowalski.
Logic for Problem Solving.
Elsevier North Holland Inc., 1979.

[33] H. T. Kung.
Why systolic architechtures?
IEEE Computer 15(1):37-46, 1982.

[34] Leslie Lamport.
A Recursive Concurrent Algorithm.
January, 1982.
Unpublished note.

[35] Henry Lieberman.
A Preview of Act 1.
Technical Report AIM-625, MIT, Artificial Intelligence Laboratory, June, 1981.

[36] John McCarthy, Paul W. Abrahams, Daniel J. Edwards, Timothy P. Hart, and Michael I. Levin.
LISP 1.5 *Programmer's Manual.*
The M.I.T. Press, Cambridge, Massachusetts, 1965.

[37] David Park.
On the Semantics of fair parallelism.
In D. Bjorner (editor), *Lecture Notes in Computer Science*, pages 504-526. Springer-Verlag, 1980.
Lecture notes in Computer Science no. 86.

[38] V. R. Pratt.
On the composition of processes.
In *Proceedings of the Ninth ACM Symposium on Principles of Programming Languages*, pages 213-223. ACM, January, 1982.

[39] J. A. Robinson.
A machine oriented logic based on the resolution principle.
Journal of the ACM 12:23-41, January, 1965.

[40] Ehud Y. Shapiro.
Alternation and the computational complexity of logic programs.
In *Proceedings of the First International Logic Programming Conference*. ADDP-GIA, Faculte des Sciences de Luminy, Marseille, France, September, 1982.

[41] Yosi Shiloach and Uzi Vishkin.
An $O(log\ n)$ parallel connectivity algorithm.
Journal of Algorithms 3:57-67, 1982.

[42] M. B. Smyth.
Finitary relations and their fair merge.
Internal Report CSR-107-82, University of Edniburgh, Computer Science Department, March, 1982.

[43] Sunichi Uchida.
Towards a New Generation Computer Architechture: Research and Development Plan for Computer Architechture in the Fifth Generation Computer Project.
Technical Report TR-001, ICOT — Institue for New Generation Computer Technology, July, 1982.

[44] William W. Wadge.
An extensional treatment of dataflow deadlock.
In G. Kahn (editor), *Semantics of Concurrent Computations*, pages 283-299. Springer-Verlag, 1979.
Lecture notes in Computer Science no. 70.

[45] David H. D. Warren.
Implementing Prolog – Compiling Predicate Logic Programs.
Technical Report 39 & 40, Department of Artificial Intelligence, University of Edinburgh, 1977.

[46] David H. D. Warren.
Perpetual processes: an unexploited Prolog programming technique.
In *Proceedings of the Prolog Programming Environments Workshop.* Datalogi, Linkoping, Sweden, March, 1982.

[47] Davi d Warren.
MegaLips now!
Unpublished note, 1982.

[48] Michael J. Wise.
A Parallel Prolog: the Contruction of a Data Driven Model.
In *Symposium on Lisp and Functional Programming*, pages 56-67. Association for Computing Machinery, 1982.