ICOT Technical Report: TR-034

TR-034

Systems Programming in Concurrent Prolog

by
Ehud Shapiro
The Weizmann Institute of Science
Rehovot, ISRAEL

November, 1983

ICOT

Mita Kokusai Bldg. 21F
4-28 Mita 1-Chome
Minato-ku Tokyo 108 Japan

(03) 456-3191 ~ 5
Telex ICOT J32964

Institute for New Generation Computer Technology

Systems Programming in Concurrent Prolog

by

Ehud Shapiro

Department of Applied Mathematics
The Weizmann Institute of Science
Rehovot 76100, ISRAEL

November, 1983

Abstract

Concurrent Prolog [23] combines the logic programming computation model with guarded-command indeterminacy and dataflow synchronization. It will form the basis of the Kernel Language [21] of the Parallel Inference Machine [30], planned by Japan's Fifth Generation Computers Project. This paper explores the feasibility of programming such a machine solely in Concurrent Prolog (in the absence of a lower-level programming language), by implementing in it a representative collection of systems programming problems.

This paper subsumes an earlier draft entitled "Concurrent Prolog as a Multiprocessor's Kernel Language."

To appear in Proceedings of the 11th ACM POPL Symposium, January, 1984.

1. Introduction

The process of turning a bare von Neumann machine into a usable computer is well understood. One of the more elegant techniques to do so is to implement a cross-compiler for a systems programming language (say C) on a usable computer. Then implement in that language an operating system kernel (say Unix), device drivers, a file system, and a programming environment. Then boot the operating system on the target computer. From that stage on the computer is usable, and application programs, compilers and interpreters for higher-level languages (say Franz Lisp and CProlog) can be developed on it.

This paper addresses the question of turning a bare computer into a usable one, but for a machine of a different type, namely, a parallel logic programming machine. In particular, it explores the suitability of Concurrent Prolog [28] as the kernel programming language[1] of such a computer, by asking the question:

1. Will a machine that implements Concurrent Prolog in hardware or firmware be usable as a general-purpose, multi-user, interactive computer?

or, stated slightly differently,

2. Is Concurrent Prolog expressive enough to be a kernel language of a general-purpose computer?

We investigate these questions only from the software, not from the hardware, side. Our concern with efficiency is limited by the assumption that the machine will execute at least as many Megalips[2] as today's computers execute Mips.

These questions are far reaching, but not purely speculative, given the Fifth Generation Computers Project's plans to design and build a parallel logic programming machine, and to use Concurrent Prolog as the basis for the machine's kernel language [21].

The paper attempts to give some evidence towards affirmative answers to these questions. To do so, we assume a computer that behaves like a virtual Concurrent Prolog machine, but has no other (lower-level or otherwise) programming constructs, special instructions, hardware interrupts, etc. We also assume that basic drivers for I/O

[1]The term *Kernel Language* denotes a hybrid between a machine language and a systems programming language, implemented by hardware or firmware. As far as I know it was introduced by the Fifth Generation Project [23].

[2]LIPS: Logical Inferences Per Second. In the context of Concurrent Prolog, it means process reductions per second.

devices are provided, which make each device understand Concurrent Prolog streams. This assumption is elaborated further below.

We then develop a collection of Concurrent Prolog programs that can run on such a machine, including:

- A Concurrent Prolog interpreter/debugger.
- A top-level crash/reboot loop, that reboots the operating system automatically upon a software (and perhaps also a hardware) crash.
- A Unix-like shell, that handles foreground and background processes, pipelining, and an abort ("Control-C") interrupt for foreground processes.
- A multiple-process manager *a la* MUF [6] , that manages the creation of, and communication with, multiple interactive processes.
- Programs for merging streams, using various scheduling strategies.
- A solution to the readers-and-writers problem, based on the concept of monitors [16].
- Shared queues, and their application to implementing managers of shared resources, such as a disk scheduler.

The programs have been developed and tested using a Concurrent Prolog interpreter, written in Prolog [28]. They show Concurrent Prolog's ability to express process creation, termination, communication, synchronization, and indeterminacy. They suggest that a "pure" Concurrent Prolog machine is self-contained, and that it will be be usable "as is", without too many extraneous features.

Even if such a machine is usable, it is not necessarily useful. For Concurrent Prolog to be a general purpose programming language, it has to solve conveniently a broad range of "real-life" problems. However, determining what is a real-life problem depends upon one's point of view. One may suggest implementing the algorithms in Aho, Hopcroft and Ullman's book as such a problem. Concurrent Prolog will exhibit a grand failure if such an implementation is attempted, and for a reason. The algorithms in that book (and, most other sequential algorithms) are deeply rooted in the von Neumann computer. One of the basic operations they use is destructive assignment of values to variables (including destructive pointer manipulation). This operation is cheap on a von Neumann machine, but is not available directly in the logic programming computation model, is prohibitively expensive to simulate, and thinking in terms of it results in an awkward programming style.

On the other hand, the operations that are cheap in Concurrent Prolog—process creation and communication—are not used in conventional sequential algorithms, almost by definition. Hence Concurrent Prolog (or any other logic programming language) is not adequate for implementing most von Neumann algorithms. One may ask : Is there anything else to implement (besides payroll programs)? Or: What is Concurrent Prolog good for, then?

Our experience to date suggests that, in contrast to von Neumann languages and algorithms, Concurrent Prolog exhibits strong affiliation with four other "trends" in computer science: Object-oriented programming, dataflow and graph-reduction languages, distributed algorithms, and systolic algorithms.

In [30] A. Takeuchi and the author show that Concurrent Prolog lends itself very naturally to the programming style and idioms of object-oriented programming languages such as Smalltalk [18] and Actors [14] . Many applications are easier to implement in this framework.

The synchronization mechanism of Concurrent Prolog—read-only variables—is a natural generalization of dataflow synchronization from functional to relational languages. The basic operation of a Concurrent Prolog program—process reduction—is basically a graph reduction operation, since a process is a DAG, and a clause can be viewed as specifying how to replace one DAG by a (possibly empty) collection of other DAGS. It is interesting to observe that the synthesis of dataflow and graph reduction mechanisms has been attempted by hardware researchers, independently of logic programming [19] .

We have some experience with implementing distributed algorithms in Concurrent Prolog. The implementation of the "Lord of the Ring" algorithm [24] in Concurrent Prolog, described in [26], exhibits a striking similarity to the English description of the algorithm, where every rule of process behavior corresponds to one Concurrent Prolog clause. That paper also reports on an implementation of a complicated distributed minimum spanning tree algorithm.

An implementation of Shiloach and Vishkin's MAXFLOW algorithm [31] demonstrates the ability of Concurrent Prolog to implement complex parallel algorithms without loss of efficiency [15] .

Numerical computations are quite remote from Artificial Intelligence, the original ecological niche of logic programming. Nevertheless, we find that the natural Concurrent Prolog solutions to numerical problems have a "systolic touch" to them, and vice-versa: that implementing systolic algorithms [22] in Concurrent Prolog is easy. A forthcoming paper will include Concurrent Prolog implementations of several systolic algorithms, including the hexagonal band-matrix multiplication algorithm [22].

Other recent applications of Concurrent Prolog include the implementation of a parallel parsing algorithm [12], an or-parallel Prolog interpreter [13], a hardware specification and debugging system [33], and a LOOPS-like [2] object-oriented knowledge representationlanguage [9].

2. Concurrent Prolog

Concurrent Prolog is a logic programming language, in that a program is a collection of universally quantified Horn-clause axioms, and a computation is an attempt to prove a goal—an existentially quantified conjunctive statement—from the axioms in the program. The goal statement describes an input/output relation for which the input is known; a successful (constructive) proof provides a corresponding output.

The difference between Concurrent Prolog and other logic programming languages (e.g. pure Prolog) is in the mechanism they provide for controlling the construction of the proof. Prolog uses the order of clauses in the program and the order of goals in a clause to guide a sequential search for a proof, and uses the cut operator to prune undesired portions of the search space. Concurrent Prolog searches for a proof in parallel. To control the search, Concurrent Prolog embodies two familiar concepts: guarded-command indeterminacy, and dataflow synchronization. They are implemented using two constructs: the commit operator "|" and the read-only annotation "?".

The commit operator is similar to Dijkstra's guarded command [7], and was first introduced to logic programming by Clark and Gregory [4]. It allows a process to make preliminary computations (specified in the guard of a clause), before choosing which action to take, i.e. which clause to use for reduction. Read-only annotations on occurrences of variables are the basic (and only) mechanism for process synchronization. Roughly speaking, a process that attempts to instantiate a variable through a read-only occurrence of it suspends until the variable is instantiated by another process. The other components of concurrent programming: process creation, termination, and communication, are already available in the abstract computation model of logic programming. A unit goal corresponds to a process, and a conjunctive goal to a system of processes. A process is created via goal reduction, and terminated by being reduced to the empty (true) goal. Conjunctive goals may share variables, which are used as communication channels between processes.

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

$$A \leftarrow G_1, G_2, \ldots, G_m \mid B_1, B_2, \ldots, B_n. \quad m, n \geq 0.$$

where the G's and the B's are atomic formulas, also called unit goals. A is called the clause's head, the G's are called its guard, and the B's its body. When the guard is empty the commit operator "|" is omitted. Clauses may contain variables marked read-only, such as "X?". The Edinburgh Prolog syntactic conventions are followed: constants begin with a lower-case letter, and variables with an upper-case letter. The special binary term $[X \mid Y]$ is used to denote the list whose head (car) is X and tail (cdr) is Y. The constant $[]$ denotes the empty list.

Concerning the declarative semantics of a guarded clause, the commit operator reads like a conjunction: A is implied by the G's and the B's. The read-only annotations can be ignored in the declarative reading.

Procedurally, a guarded-clause specifies a behavior similar to an alternative in a guarded-command. To reduce a process A using a clause

$$A1 \leftarrow G \mid B,$$

unify A with $A1$, and, if successful, recursively reduce G to the empty system, and, if successful, commit to that clause, and, if successful, reduce A to B.

The unification of a process against the head of a clause serves several functions: passing parameters, assigning values to variables, selecting and constructing data-structures, and sending and receiving messages. The example programs below demonstrate all these uses of unification.

The reduction of a process may suspend or fail during almost any of the steps described above. The unification of the process against the head of a clause suspends if it requires the instantiation of variables occurring as read-only in A or $A1$. It fails if A and $A1$ are not unifiable. The computation of the guard system G suspends if any of the processes in it suspends, and fails if any of them fails. As in guarded-commands, at most one of the process's or-parallel guard-systems may commit.

Prior to commitment, partial results computed by the first two steps of the reduction—unifying the process against the head of the clause and solving the guard—are not accessible to other processes in A's system. This prevents interference between brother or-parallel computations, and eliminates the need for distributed backtracking.

This completes the informal description of Concurrent Prolog. The simplicity of the language is an asset when attempting a hardware or firmware implementation of it.

3. A Meta-Interpreter for Concurrent Prolog

One of the simpler ways to implement a programming environment for a programming language L is augmenting L's interpreter. Among the program development tools that can be implemented in this way are sophisticated debuggers [27] , runtime-statistics packages, extensions to the language, and new embedded languages. The difficulty of implementing these tools grows with the complexity of that interpreter.

For reasons of bootstrapping and elegance, the preferred implementation language for L's programming environment is L itself, as argued eloquently by Sandewall[25]. *Hence the case in which an L interpreter can be implemented in L is of clear practical importance, as well as a useful criteria for evaluating the expressiveness and completeness of the language*, as argued by Sussman and Steele in [32] .

Designing an expressive language with a simple meta-interpreter[3] is like solving a fixpoint equation. If the language L is too weak, then L's data-structures may not be rich enough to represent L programs conveniently. If the control constructs of L are incomplete they cannot be used to simulate themselves conveniently.

On the other hand, if the control structures of L are awkward and unrestricted and the data-structures are too baroque, then its interpreter becomes very large and unintelligible (e.g. *goto* cannot be used in a simple way to simulate unrestricted *goto*, but the easiest way to simulate a *while* statement is using a *while* statement in the interpreter).

A meta interpreter for pure sequential Prolog can be written in three Prolog clauses, and, indeed, implementing software tools and embedded languages via extending this interpreter is a common activity for Prolog programmers.

A meta interpreter for Concurrent Prolog is described below. It assumes the existence of a built-in system predicate *clauses(A,Cs)*, that returns in Cs the list of all clauses in the interpreted program whose head is potentially unifiable with A.[4] The constant *true* signifies an empty guard or an empty body.

[3]Called a meta-circular interpreter in [32].

[4]In our current implementation Cs is the list of all clauses with the same head predicate as A. Better indexing mechanism can make the predicate more selective. Another possible optimization is to use the bounded-buffer technique of Takeuchi and Furukawa [34], to generate clauses on a demand-driven basis.

```
reduce(true).
reduce((A,B)) ←
        reduce(A?), reduce(B?).
reduce(A) ←
        clauses(A,Clauses) |
        resolve(A,Clauses,Body),
        reduce(Body?).

resolve(A,[(A←Guard|Body)|Cs],Body) ←
        reduce(Guard) | true.
resolve(A,[C|Clauses],Body)←
        resolve(A,Clauses,Body) | true.
```

Program 1: A Meta-interpreter for Concurrent Prolog

Like any other Concurrent Prolog program, Program 1 can be read both declaratively, i.e. as a set of axioms, and operationally, i.e. as a set of rules defining the behavior of processes. Declaratively, *reduce(A)* states that A is true (provable) with respect to the axioms defined in the predicate *clauses*. Operationally, the process *reduce(A)* attempts to reduce the system of processes A to the empty (halting) system *true*.

Declaratively, the axioms of *reduce* read: *true* is true. The conjunction *A,B* is true if A is true and B is true. The goal A is true if there are clauses Cs with the same head predicate of A, resolving A with Cs gives B, and B is true. The predicate *resolve(A,[C| Cs],B)* reads, declaratively, that resolving A with the axioms [C|Cs] gives B if the clause C has head A, guard G and body B, and the guard G is true, or if recursively resolving A with Cs gives B.

Operationally, the clauses of *reduce* say that the process *true* halts. The process *(A, B)*reduces itself to the processes A and B, and that the process A, with clauses Cs, reduces itself to B if the result of resolving A with Cs is B.

The reader not familiar with logic-programming may be puzzled by this interpreter. It seems to capture the control part of the computation, but does not seem to deal at all with unification, the data component. The answer to the puzzle is that the call to the first clause of *resolve* is doing the work, by unifying the process with the head of the clause. Their unification is achieved by calling them with the same name, *A*.

This interpreter assumes one global program, whose axioms are accessible via the system predicate *clauses*, as in conventional Prolog implementations. In a real implementation of Concurrent Prolog, programs would be objects that can be passed as arguments, and *reduce* and *clauses* would have an additional argument, the program being simulated.

This interpreter cannot execute Concurrent Prolog programs that use built-in system predicates, such as itself (it uses the predicate *clauses*). The current implementation of Concurrent Prolog contains several (13) system predicates: metalogical predicates (*clauses* and *system*), control predicates (*otherwise* and =), interface to the underlying prolog, I/O (*read* and *write*), and arithmetic predicates (the lazy evaluator := and 5 arithmetic test predicates). To handle system predicates, the interpreter can be augmented with the clause

```
reduce(A) ←
        system(A) | A.
```

system(X) is a system predicate that succeeds if X is a Concurrent Prolog system predicate, and fails otherwise. For example, the call *system(system(X))* succeeds. The clause demonstrates the use of the *meta-variable*, a facility also available in Prolog, which allows to pass processes to other processes as data-structures. It is used extensively in the shell programs below.

One may suggest that using the metavariable facility, a Concurrent Prolog meta-interpreter can be implemented via the clause

```
reduce(A) ← A.
```

This claim is true, except that it will be rather difficult to implement the software tools mentioned earlier as an extensions to this interpreter, whereas implementing a Concurrent Prolog single stepper by extending Program 1 is a trivial matter.

The interpreter in Program 1 is 10 to 20 times slower than the underlying Concurrent Prolog implementation. We feel that it is reasonable to pay a 10-fold slowdown during the program development phase for a good programming environment. Besides, a default to the underlying Concurrent Prolog can be incorporated easily, as in the case of system predicates, so that in developing large systems only the portion of the code that is under development needs the extra layer of simulation.

4. Streams

Concurrent Prolog processes communicate via shared logical-variables. Logical variables are single-assignment: they can be either uninstantiated or instantiated, but, once instantiated, their value cannot be destructively modified. Hence the Concurrent

Prolog computation model is indifferent to the distinction between the shared-memory computation model and the communication based model. A shared logical-variable can be viewed as a shared memory cell that can accept only one value, or as a communication channel that can transmit only one message.

The distinction between the "reader" and "writer" of a shared variable (or the "sender" and "receiver" of the message) is done via read-only annotations. A process *p(...X?...)* cannot instantiate X. Attempts of *p* to reduce itself to other processes using clauses that require the instantiation of X, such as

$$p(\ldots f(a)\ldots) \leftarrow \ldots$$

suspend, until X is instantiated by some other process. If X is instantiated to $f(Y)$, then the process *p* can unify with that clause, even though it instantiated Y to *a*, since the scope of a read-only annotation is only the main functor of a term, but not variables that occur inside the term. This property enables a powerful programming technique that uses *incomplete messages* [28] .

Even though logical variables are single-assignment, two processes can communicate with each other via a single shared variable, by instantiating a variable into a term that contains both the message and another variable, to be used in subsequent communications. This programming techniques gives the effect of streams.

The cleanest way to implement I/O functions in a Concurrent Prolog machine is for I/O devices to generate and/or consume Concurrent Prolog streams. The current implementation of Concurrent Prolog, which is an interpreter written in Prolog [28] , supports only terminal I/O (the rest is done by the underlying Prolog). It implements the stream abstraction for the user terminal via two predicates, *instream(X)*, which generated the stream X of terms typed in by the user, and *outstream(X)*, that outputs to the screen the stream X. They are implemented using the underlying Prolog *read* and *write* predicates.

```
instream([X|Xs]) ←
        read(X) | instream(Xs).

outstream([]).
outstream([X|Xs]) ←
        write(X), outstream(Xs?).
```

Program 2: Implementing terminal I/O streams using *read* and *write*

If we want *instream* to allow the user to signify the end of the stream, the program has to be complicated a little.

Using these programs, a "device-driver" that implements a stream interface to the terminal can be specified:

```
terminal(Keyboard,Screen) ←
        instream(KeyBoard), outstream(Screen?).
```

In a virtual Concurrent Prolog machine in which interfaces to I/O device drivers are implemented as streams, there will be no need for specialized I/O primitives. One possible exception is a screen-output primitive (*write* or *bitblt*), which may be needed for convenience and efficiency.

5. Booting an Operating System

Assume that device drivers for a terminal (screen, keyboard and a mouse), disk, and a local network have been defined for a personal workstation. Then the following program can be used to boot its operating system:[5]

```
boot ←
        monitor(KeyBoard?,Mouse?,Screen,DiskIn?,DiskOut,NetIn?,Netout),
        terminal(KeyBoard,Mouse,Screen?),
        disk(DiskIn,DiskOut?),
        net(NetIn,NetOut?) |
        true.
boot ←
        otherwise |
        boot.
```

Program 3: Booting an operating system

The first clause invokes the device drivers and the monitor. The second clause automatically reboots the system upon a software crash of either the monitor or the device drivers. *otherwise* is a Concurrent Prolog system predicate that succeeds if and when all of its brother or-parallel guards fail. Declaratively, it may read as the negation of the disjunction of the guards of the brother clauses[6].

[5] We are aware of the fact that efficiency considerations may prevent the use of pure streams for devices that generate a lot of useless data, such as a mouse, and that some lower-level interface may be required.

[6] The predicate *otherwise* is not implemented correctly in the current Concurrent Prolog interpreter [28]. It may succeed when it has suspended brother or-parallel guards, instead of suspending, and succeeding only when all such guards fail. Hence programs using it are not fully debugged.

6. A Unix-like Shell

A shell is a process that receives a stream of commands from the terminal and executes them. In our context the commands are processes, and executing them means invoking them. A simple shell can be implemented using the metavariable facility,

```
shell([X|Xs]) ←
        X, shell(Xs?).
shell([]).
```

This shell is batch-oriented. It behaves like a Unix-shell that executes all commands in "background" mode, in the sense that it does not wait for the completion of the previous process before accepting the next command. As is, it achieves the effect of Unix-like pipes, using conjunctive goals with shared variables as commands. For example, the Unix command

p | q | r

can be simulated with the conjunctive system

p(X), q(X?,Y), r(Y?).

provided that the Unix command p does not read from its primary input and q does not write to its primary output. External I/O by user programs is handled below.

Note that since the process's I/O streams have explicit names, we are not confined to linear pipelining, and any desired I/O configuration of the processes can be specified.

One of this shell's drawbacks is that it will crash if the user process X crashes, since X and shell(Xs) are part of the same conjunctive system, which fails if one of its members fails. This can be remedied by calling *envelope(X)* instead of X.

```
envelope(X) ← X | write(halted(X)).
envelope(X) ← otherwise | write(failed(X)).
```

It is easy to augment the shell to distinguish between background and foreground processes, assuming that every command X is tagged *bg(X)* or *fg(X)*, as done in Program 4.

```
(1) shell([]).
(2) shell([fg(X)|Xs]) ←
        envelope(X) | shell(Xs?).
(3) shell([bg(X)|Xs]) ←
        envelope(X), shell(Xs?).
```

Program 4: A shell that handles foreground and background processes

Note that foreground processes are executed in the shell's guard. This allows a simple extension to shell so it will handle an abort ("control-C" on decent computers) interrupt for foreground processes. Upon the reception of an *abort* command the currently running foreground process (if there is one) is aborted, and the content of the input stream past the abort command is flushed. This is achieved by the clauses in Program 4a.

```
(4) shell(Xs) ←
        seek(abort,Xs,Ys) | shell(Ys?).

seek(X,[X|Xs],Xs).
seek(X,[Y|Xs],Ys) ←
        X\==Y | seek(X,Xs?,Ys).
```

Program 4a: An extension to the shell that handled an *abort* interrupt.

The program operates as follows. When an *fg(X)* command is received, the two guards, *envelope* and *seek* are spawned in parallel, and begin to race. The first to commit aborts the second, so if *envelope* terminates before *seek* found an *abort* command in the input stream (most probably because the user hasn't typed such a command yet) then the *envelope* commits, *seek* is aborted, and *shell* proceeds normally with the next command. On the other hand, if *seek* succeeds in finding an *abort* command before *envelope* terminates, then *envelope* is aborted, and *shell* proceeds with the input past the *abort* command, as returned by *seek*.

A more general interrupt, *grand_abort*, that aborts all processes spawned by *shell*, both foreground and background, can also be implemented quite easily:

```
topshell(Xs) ←
        shell(Xs) | true.
topshell(Xs) ←
        seek(grand_abort,Xs,Ys) | topshell(Ys?).
```

The distinction the shell in Program 4 makes between background and foreground processing is not of much use, however, since foreground processes are not interactive, i.e. they do not have access to the shell's input stream. One problem with the shell giving a user program its input stream is that upon termination the user program has to return the remaining stream back, so that the shell can proceed. Since we cannot expect every interactive user program to obey a certain convention for halting (cf. quit, exit, halt, stop, bye, etc.) the shell has to implement a uniform command, say *exit* to "softly" terminate an interactive session with a user program (in contrast to aborting it). A filter, called *switch* monitors the input stream to the program. Upon the reception of an *exit* command it closes the output stream to the program, returns the rest of the input stream to the shell, and terminates. A reasonable interactive user program should terminate upon encountering the end of the input stream. If it is not reasonable, an *abort* interrupt will always do the job. The following code implements this idea. Commands to interactive foreground processes are of the form *fg(P,Pi)*, where P is the process and Pi is its input stream. For example, a command to run the process *foo(X)* with input stream *X* will be given as *fg(foo(X?),X)*.

```
(5) shell([fg(X,Xi)|Xs]) ←
        envelope(X), switch(Xs?,Xi,Ys) |
        shell(Ys?).

switch([exit|Xs],[],Xs).
switch([X|Xs],[X|Ys],Zs) ←
        X\=exit | switch(Xs?,Ys,Zs).
```

Program 4b: An extension to the shell that handles interactive user programs.

7. A manager of Multiple Interactive Processes

The shell described above can handle only one interactive process at a time, like the DEC supplied TOPS-20 EXEC. MUF (Multiple User Forks) is a popular DEC-20 program, developed at Yale university [6], which overcomes this limitation. It can handle multiple interactive processes, and has a mechanism for easy context switching. It cannot compete, of course, with the convenience of a system with a bitmap display and a pointing device.

MUF associates names with processes. It has commands for creating a new process, freezing or killing a process, resuming a frozen process, and others. Program 5 achieves some of this functionality.

```
(0) muf(X) ←
        muf(X,[]).
(1) muf([create(Pname,Process,Pin,Pout)|Input],Ps) ←
        Process,
        tag(Pname,Pout),
        muf([resume(Pname)|Input?],[(Pname,Pin)|Ps]).
(2) muf([resume(Pname)|Input],Ps) ←
        find_process(Pname,Ps,Pin,Ps1)|
        distribute(Input?,Pin,Input1,Pin1),
        muf(Input1?,[(Pname,Pin1)|Ps1]).
(3) muf([exit|Input][,[(Pname,[])|Ps]) ←
        muf(Input?,Ps).
(4) muf([],Ps) ←
        close_input(Ps).
(1) find_process(Pname,[(Pname,Pin)|Ps],Pin,Ps).
(2) find_process(Pname,[Pr|Ps],Pin,[Pr|Ps1])←
        otherwise |
        find_process(Pname,Ps,Pin,Ps1).
(1) distribute([],Pin,[],Pin).
(2) distribute([X|Input],Pin,[X| Input],Pin)←
        muf_command(X)| true.
(3) distribute([X|Input],[X|Pin],Input1,Pin1)←
        otherwise |
        distribute(Input?,Pin,Input1,Pin1).
(1) close_input([]).
(2) close_input([(Pname,[])|Ps])←
        close_input(Ps).
(1) muf_command(create(_,_,_,_)).
(2) muf_command(resume(_)).
(3) muf_command(exit).
```

Program 5: mini-MUF

The *muf* process is invoked with the call *muf(X?)* where X is its input stream. It first initializes itself with the empty process list, using Clause (0), then iterates, serving user commands.

On the command *create(Pname,Process,Pi,Po)* it creates a process *Process*, and a process *tag(Pname,Po)*, that tags the process's output stream elements with the process's name, and displays them on the screen. It also adds a record with the process's name, *Pname*, and input stream, *Pi*, to its process list, and sends itself the command *resume(Pname)*.

On the command *resume(Pname)*, *muf* uses Clause (3). It searches its process list for the input stream of the process *Pname*, and puts this process record first on the list. This is done by *find_process*. If successful, it invokes *distribute(Input?,Pin,Input1,Pin1)*, which copies the elements of the stream *Input* to the stream *Pi* (Clause 3) until it reaches the end of the stream (Clause 1), or encounters a command to *muf* (Clause 2). In that event it terminates, returning the updated streams in *Pi1* and *Input1*. *muf* itself is suspended on *Input1*.

On the command *exit*, *muf* closes the input stream of the current process, and removes it from the process list (Clause 3).

When encountering the end of its input stream, *muf* closes the input streams of all the processes in its list, and terminates (Clause 4).

Some of the frills of the real MUF can be easily incorporated in our mini-implementation. For example, the *freeze* command resumes the previously resumed process, without having to name it explicitly. This is implemented by the following clause:

```
muf([freeze|Input],[Pr,(Pname,Pi)|Ps])←
        muf([resume(Pname)|Input?],[(Pname,Pi),Pr|Ps]).
```

which reverses the order of the first two process records on the process list, and sends itself a *resume* command with the name of the previously resumed process. A similar default for *exit* can be added likewise.

Note that if the length of the process list is less then two, this clause would not apply, since its head would not unify with the *muf* process. Similarly, if a *resume* command is given with a wrong argument, Clause (2) wouldn't apply, since the guard, *find_process*, would fail.

muf, as defined in Program 5, would crash upon receiving such erroneous commands. Adding the following clause would cause it to default, in such cases, to an error-message routine:

```
muf([X|Input],Ps) ←
        otherwise |
        muf_error(X,Ps),
        muf(Input?,Ps).
```

muf_error analyses the command with respect to the process list, and reports to the user the type of error it's made.

Similarly easy to implement are queries concerning the names of the processes in the process list, and the identity of the currently resumed process.

8. Merging streams

A Concurrent Prolog process can have several input and/or output streams, and use them to communicate with several other processes; but the number of these streams 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 functional and dataflow languages *merge* is a built in operator [1,11]. Logic programs, on the other hand, can express it directly, as shown by Clark and Gregory [4]. Program 6 adapts their implementation to Concurrent Prolog. It implements the process *merge(X?, Y?, Z)*, which computes the relation "Z is the interleaving of X and Y".

```
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 6: Merging two streams

Using stream-merge as the basic method of many-to-one communication poses three major problems:

1. How to provide a fair access to the shared process?

2. How to minimize communication delay?

3. How to route a response back to the sender?

The building-block of a fair communication network is a fair merge operator. The abstract computation model of Concurrent Prolog is under-specified, and does not determine which of the first two clauses of Program 6 would be chosen for reduction if both input streams have elements ready. Dijkstra ([7], p. 204) has considered this under-specification a desirable property of the guarded-command, and recommended simulating a totally erratic demon when choosing between two applicable guards. Nevertheless, for reasons of efficiency and expressiveness, we prefer to work in a more stable environment, and allow the programmer to control the chosen clause in the special case in which there are applicable clauses with empty guards.

A stable Concurrent Prolog machine always reduces a process using the first unifiable clause with an empty guard, if such a clause exists.

A stable implementation is a natural consequence of having a sequential dispatcher for the guards of a process. Such a dispatcher would perform the unification of the process against the clauses' heads sequentially, and dispatch the guard of a clause if the unification with its head succeeds. It would commit as soon as it succeeds in unifying the process with the head of a clause whose guard is empty.

The definition of a stable machine assumes that some order (say, text order) is imposed on the clauses of each procedure in the program. Note that a stable implementation guarantees nothing about the selection of clauses with non-empty guards.

On a stable machine, Program 6 above will always prefer the first stream over the second, if both streams have elements ready. Hence it does not guarantee bounded waiting (however, it may be used to implement a notion of interrupts with different relative priorities).

To achieve fairness, this program is modified slightly, so it switches the positions of the two streams on each reduction, as specified in Program 7. On a stable machine, this would ensure 2-bounded-waiting.

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

Program 7: Fairly merging two streams

This program is a satisfactory solution to the problem of merging two streams. More than two streams can be merged by constructing a tree of merge operators. It is not difficult to see (cf. [29])that a balanced merge tree composed of fair binary merge operators ensures linear bounded-waiting, and has a logarithmic communication delay.

The construction of a static balanced merge tree is easy. To allow a dynamically changing set of processes a fair and efficient access to a shared resource, a more innovative solution is required. In [29]we define self-balancing binary and ternary merge operators. These operators compose dynamically into a balanced merge-tree, using algorithms similar to 2-3-tree insertion and deletion. The algorithms require sending messages that contain communication channels, in order to reshape the tree. In other words, the algorithm uses incomplete messages. 2-3 merge trees also ensure linear bounded-waiting and logarithmic communication delay, hence we believe they provide an acceptable solution to the problem of dynamic many-to-one communication.

The problem of routing back the response to a message is solved, at the programming level, using incomplete messages. A message that requires a response typically contains an uninstantiated variable; the sender of the message suspends, looking at the variable in read-only mode. The recipient of the message responds to it by instantiating that variable. This technique is used in the monitor, queue, and disk scheduling programs below.

8.1 A note on abstract stream operations

A more abstract (but also longer and less efficient) implementation of *merge* can be obtained using the *send(X,S,S1)* and *receive(X,S,S1)* operations on streams. They define the relation "the result of sending (receiving) X on stream S is the stream S1" as follows:

```
send(X,[X|Xs],Xs).
receive(X,[X|Xs],Xs?).
```

Such an implementation hides the internal representation of the stream, and eliminates the need to use the read-only annotation almost entirely in the calling program, since the resulting stream of *receive* is already annotated as read-only. We find the use of *send* and *receive* explicitly, instead of achieving this effect implicitly via unification, essential for the readability of programs with complex communication patterns, such as the ones described in [15,26].

The *send* and *receive* calls can be eliminated for the sake of efficiency using standard partial-evaluation and program-transformation techniques [20,35].

9. Monitors and the readers-and-writers problem

The Concurrent Prolog solution to the readers and writers problem uses this method of many-to-one communication. It is very similar, in spirit, to the idea of monitors [16]. A designated process (a 'monitor') holds the shared data in a local argument, and serves the merged input stream of 'read' and 'write' requests ('monitor calls'). It responds to a 'read' request through the uninstantiated response variable in it ('result argument').

A schematic implementation of a monitor is shown in Program 8. Note that it serves a sequence of *read* requests in parallel, since the recursive invocation of *monitor* in Clause (2) is not suspended on the result of *serve*, in contrast to Clause (1).

```
(1) monitor([write(Args)|S], Data) ←
        serve(write(Args), Data, NewData), monitor(S?, NewData?).
(2) monitor([read(Args)|S], Data) ←
        serve(read(Args), Data, _), monitor(S?, Data).
(3) monitor([],_).
```

Program 8: A schematic implementation of a monitor

In monitor-based programming languages, a procedure call and a monitor call are two basic, mutually irreducible operations. In Concurrent Prolog, on the other hand, there is one basic construct, a process invocation, whereas a monitor call is a secondary concept, or, rather, a programming technique.

Concurrent Prolog monitors and merge operators can implement operating systems in a functional style without side-effects, using techniques similar to Henderson's [11].

10. 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.

The following implementation of shared queues is a canonical example of Concurrent Prolog programming style. It exploits two powerful logic programming techniques: incomplete messages, and difference-lists.

A shared queue manager is an instance of a monitor. ***enqueue*** is a "write' operation, and "dequeue" involves both "read" and "write". An abstract implementation of a queue monitor is shown in Program 9.

```
(0) queue_monitor(S) ←create_queue(Q),
        queue_monitor(S,Q).
(1) queue_monitor([Request|S],Q) ←
        serve(Request,Q,Q1),
        queue_monitor(S?,Q1?).
(2) queue_monitor([],Q).
```

Program 9: A queue monitor

Clause (0) creates an empty queue; Clause (1) iterates, serving queue requests; and Clause (2) halts the queue monitor upon reaching the end of the requests stream.

The implementation of the queue operations employs difference-lists [3]. A difference-list represents a list of elements (in this context, the queue's content) as the difference between two lists, or streams. For example, the difference between [1,2,3,4|X] and X is the list [1,2,3,4]. As a notational convention, we use the binary term X\Y (read "the difference between X and Y"), to denote the list that is the difference between the list X and the list Y. Note that this term has no special properties predefiend, and any binary term will do, as long as it is used consistently.

```
create_queue(X\X).
serve(enqueue(X),Head\[X|NewTail],Head\NewTail).
serve(dequeue(X),[X|NewHead]\Tail,NewHead\Tail).
```

Program 9a: Queue operations

create_queue(Q) states that Q is an empty difference-list. The clauses for *serve* define the relation between the operation, the old queue, and the new queue. On an *enqueue(X)* message, X is unified with the first element of the Tail stream, and in the new queue NewTail is the rest of the stream. On a *dequeue(X)* message, X is unified with the first element of the Head stream, and in the new queue NewHead is the rest of the old Head stream.

Operationally, the program mimics the pointer twiddling of a conventional queue program. One difference is the simplicity and uniformity of the way in which variables are transmitted into and from the queue, using unification, compared to any other method of parameter passing and message routing.

Another is the behavior of the program when more *dequeue* messages have arrived then *enqueue* messages. In this case the content of the difference-list becomes "negative". The Head runs ahead of the Tail, and the negative difference between them is a list of uninstantiated variables, each for an excessive dequeue message. Presumably, a process who sends such a message then suspends on its variables in a read-only mode. One consequence is that excessive dequeue requests are served exactly in the order in which they arrived.

The program can be condensed and simplified, using program transformation techniques [20,35]. The resulting program is more efficient, and reveals more clearly the declarative semantics of the queue monitor. Its operational semantics, however, seems to become a bit more obscure, and its does not hide the internal representation of the queue, as Program 10 does.

```
(1) queue_monitor(S) ←
        queue_monitor(S?, X\X).

(1) queue_monitor([dequeue(X)|S] , [X|NewHead]\Tail) ←
        queue_monitor(S?, NewHead\Tail).
(2) queue_monitor([enqueue(X)|S],  Head\[X|NewTail]) ←
        queue_monitor(S?, Head\NewTail).
(3) queue_monitor([],_).
```

Program 10: A simplified queue monitor

Declaratively, the queue_monitor program computes the relation *queue(S)*, which says that S is a legal stream of queue operations. It uses an auxiliary relation *queue_monitor(S,Dequeue\Enqueue)*, which says that *Dequeue* is the list of all elements X such that *dequeue(X)* occurs in S (Clause 1), and that *Enqueue* is the list of all elements X such that *enqueue(X)* occurs in S (Clause 2). The interface between these two relations (Clause 0), constrains the list of enqueued elements to be identical to the list of dequeued elements, by calling them with the same name.

11. Bounded-buffer communication

Bounded buffers were introduced into logic programming by Clark and Gregory [4]as a primitive construct. Their principal use in logic-programming is not to utilize a fixed memory-area for communication, but rather to enforce tighter synchronization between the producer and the consumer of a stream.

Takeuchi and Furukawa [34]have shown how to implement bounded-buffers Concurrent Prolog, hence it need not be considered a primitive. Their implementation represents the buffer using a difference-list, and uses incomplete messages to synchronize the producer and the consumer of the stream.

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:

"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 [17]p.94).

```
(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 *: The SCAN disk-arm scheduler

The disk scheduler has two input streams—a stream of I/O requests from the user(s) of the disk, and a stream of incomplete messages 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 to 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 stream of partially determined (incomplete) 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 by Clause (0). Following 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 sends a 'halt' message to the disk controller.

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 to be upsweep, where 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 Clause (1).
- Clauses (3) and (4): Are the symmetric clauses 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, then it is inserted to *UpQ* by Clause (3), otherwise it is inserted to *DownQ*, by Clause (4).

The insertion operation is a straightforward ordered-list insertion. More efficient data-structures, such as 2-3-trees, can be used if necessary.

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 become 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(N, IO, [X|D], [X|D1]) ←
        N>0 | N1:=N-1, disk(N1?, IO, D, D1).
```

Program *: 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.

13. Conclusion

We have provided some evidence that a machine that implements Concurrent Prolog in hardware or firmware will be self-contained, usable, and useful, without much need to resort to reactionary concepts and techniques.

The next logical step is to build it.

Acknowledgements

This research is supported in part by IBM Poughkeepsie, Data Systems Division. Part of it was carried while the author was visiting ICOT, the Institute for New Generation Computer Technology, Tokyo.

The paper benefited from a critical survey of an earlier paper of mine, written by David Gelenter [10].

References

[1] Arvind and J. Dean Brock,
Streams and Managers,
in *Semantics of Concurrent Computations*, G. Kahn (ed.) pp.452-465, LNCS 70, Springer-Verlag, 1979.

[2] Daniel G. Bobrow and Mark Stefik
The LOOPS Manual (preliminary version),
Memo KB-VLSI-81-13, Xerox PARC. 1983.

[3] Keith L. Clark and Stan-Ake Tarnlund,
A first-order theory of data and programs,
in *Information Processing 77*, B. Gilechrist (ed.), pp.939-944, North-Holland, 1977.

[4] Keith L. Clark and Steve Gregory
A relational language for parallel programming,
in *Proceedings of the the ACM Conference on Functional Languages and Computer Architecture*, October, 1981.

[5] Keith L. Clark and Steve Gregory
PARLOG: A Parallel Logic Programming Language
Research report DOC 83/5, Department of Computing, Imperial College of Science and Technology, May 1983.

[6] J.R. Ellis, N. Mishkin, and S.R. Wood
Tools: an Environment for Timeshared Computing and Programming,
Research Report 232, Department of Computer Science, Yale University, 1982.

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

[8] Daniel P. Friedman and David S. Wise

An Indeterminate Constructor for Applicative Programming
in *Conference Record of the Seventh Annual ACM Symposium on Principle of Programming Languages*, pp. 245-250, 1980.

[9] K. Furukawa, A. Takeuchi, and S. Kunifuji
Mandala: A Knowledge Programming Language on Concurrent Prolog,
ICOT Technical Memorandum TM-0028 (in Japanese), 1983.

[10] David Gelenter
A Note on Systems Programming in Concurrent Prolog,
Unpublished manuscript, Yale University, 1983.

[11] Peter Henderson
Purely Functional Operating Systems
in *Functional Programming and its Applications*, P. Henderson and D.A. Turner (eds.), Cambridge University Press, 1982.

[12] Hideki Hirakawa
Chart Parsing in Concurrent Prolog,
ICOT Technical Report TR-008, 1983.

[13] Hideki Hirakawa et al.
Implementing an Or-Parallel Optimizing Prolog System (POPS) in Concurrent Prolog,
ICOT Technical Report TR-020, 1983.

[14] Carl C. Hewitt
A universal modular Actor formalism for artificial intelligence.
In *Proceedings of the Third International Joint Conference on Artificial Intelligence*, IJCAI, 1973.

[15] Lisa Hellerstein and Ehud Shapiro
Algorithmic Programming in Concurrent Prolog: the MAXFLOW experience.
Technical Report CS83-12, Department of Applied Mathematics, The Weizmann Institute of of Science, 1983.

[16] C.A.R. Hoare
Monitors: an operating systems structuring concept
Communications of the ACM, 17(10), pp.4549-557, 1974.

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

[18] Daniel H. Ingalls

The SmallTalk-76 programming system: design and implementation.
In *Conference records of the Fifth Annual ACM Symposium on Principles of Programming Languages*, pages 9-16, ACM, January 1976.

[19] Robert M. Keller, Gary Lindstrom, and Elliot I. Organic
Rediflow: a multiprocessing architecture combining reduction with data-flow.
Unpublished manuscript, Department of Computer Science, University of Utah, 1983.

[20] H.J. Komorowski
Partial evaluation as a means for inferencing data-structures in an applicative language: a theory and implementation in the case of Prolog.
In *Conference Record of the Ninth Annual ACM Symposium on Principles of Programming Languages*, pp.255-268, ACM, 1982.

[21] S. Kunifuji et al.
Conceptual Specification of the Fifth Generation Kernel Language Version 1 (preliminary draft)
ICOT Technical Memorandum TM-0028, 1983.

[22] H.T. Kung
Let's Design Algorithms for VLSI Systems,
Technical Report CMU-CS-79-151, Department of Computer Science, Carnegie-Mellon University, 1979.

[23] T. Moto-Oka et al.
Challenge for knowledge information processing systems (preliminary report on fifth generation computer systems)
In *Proceedings of International Conference on Fifth Generation Computer Systems*, pages 1-85, JIPDEC, 1981.

[24] Danny Dolev, Maria Klawe and Michael Rodeh
An $0(n \log n)$ Uni-directional distributed algorithm for extrema finding in a circle,
Journal of Algorithm 3, pages 245–260, 1982.

[25] Erik Sandewall
Programming in an interactive environment: the Lisp experience.
Computing Surveys, ACM, March 1978.

[26] Avner Shafrir and Ehud Shapiro
Distributed Programming in Concurrent Prolog,
Technical Report CS83-12, Department of Applied Mathematics, The Weizmann Institute of of Science, 1983.

[27] Ehud Shapiro
Algorithmic Program Debugging,
ACM Distinguished Dissertation Series, MIT Press, 1983.

[28] Ehud Shapiro
A Subset of Concurrent Prolog and its Interpreter,
Technical Report TR-003, ICOT—Institute for New Generation Computer Technology, 1983. Also available as Technical Report CS83-06, Department of Applied Mathematics, The Weizmann Institute of of Science.

[29] Ehud Shapiro,
Fair, Biased, and Self-Balancing Merge Operators: Their Specification and Implementation in Concurrent Prolog,
Technical Report CS83-12, Department of Applied Mathematics, The Weizmann Institute of of Science, 1983.

[30] Ehud Shapiro and Akikazu Takeuchi
Object Oriented Programming In Concurrent Prolog,
Journal of New Generation Computing Volume 1, Number 1, 1983.

[31] Yossi Shiloach and Uzi Vishkin
An $O(n2 \log n)$ parallel MAX-FLOW algorithm,
J. of Algorithms, Vol. 3, #2, June 1982. pp. 128-147.

[32] Guy L. Steele Jr. and Gerald J. Sussman
The Art of the Interpreter or, The Modularity, Complex
Technical Memorandum AIM-453, Artificial Intelligence Laboratory, MIT, May 1978.

[33] Nohirisa Suzuki
Experience with specification and verification of complex computer using Concurrent Prolog
in *Logic Programming and its Applications,* D.H.D. Warren and M. van Caneghem (eds), Lawrence Erlbaum Press, To appear.

[34] A. Takeuchi and K. Furukawa
Interprocess Communication in Concurrent Prolog,
in *Proc. Logic Programming Workshop 83,* pp. 171-185, Albufeira, Portugal, June 1983. Also ICOT Technical Report TR-006, 1983.

[35] Hisao Tamaki and Taisuke Sato
A Transformation System for Logic Programs which Preserves Equivalence,
ICOT Technical Report TR-018, 1983.

[36] Sunichi Uchida
Inference machine: from sequential to parallel.
In *Proceedings of the 10th Annual International Symposium on Computer Architecture*, pages 410-416, Stockholm, 1983. Also ICOT Technical Report TR-011.