Specification and Initialization of a Logic Computer System

Anthony J. Kusalik

Technical Report 85-10

July 1985

Specification and Initialization of a Logic Computer System

Anthony J. Kusalik

Computer Science Department
University of British Columbia
Vancouver, B.C., Canada V6T 1W5

Technical Report 85-10
July 1985

ABSTRACT

A logic computer system consists of an inference machine and a compatible logic operating system. This paper describes prospective models for a logic computer system, and its hardware and software components. The language Concurrent Prolog serves as the single implementation, specification, and machine language. The computer system is represented as a logic programming goal *logic_computer_system*. Specification of the system corresponds to resolution of this goal. Clauses used to solve the goal – and ensuing subgoals – progressively refine the machine, operating system, and computer system designs. In addition, the accumulation of all clauses describing the logic operating system constitute its implementation. Logic computer systems with vastly different fundamental characteristics can be concisely specified in this manner. Two contrasting examples are given and discussed. An important characteristic of both peripheral devices and the overall computer system, whether they are restartable or perpetual, is examined. As well, a method for operational initialization of the logic computer system is presented. The same clauses which incrementally specify characteristics of the computer system also describe the manner in which this initialization takes place.

1. Introduction

In general, an operating system can be regarded as enhancing the set of services and capabilities provided by underlying computer hardware. A "logic operating system" fulfills this role for a logic inference machine and is implemented in a logic programming language. An inference machine and a compatible logic operating system constitute a "logic computer system".

An important aspect of logic programs is that axioms (the program statements) can be read two ways: declaratively or operationally. A single logic programming language can be used for both specification and implementation. The duality permits the same axioms which describe a model to be used in implementing a program conforming to the model.

The expressive power of logic programs allows many conventional notions to be adapted to logic inference machines and their operating systems. For example, principles of object-oriented programming work well in a logic programming environment [Kahn82, ShTa83, Zani84]. However, the properties of logic programming languages may suggest new techniques and ideals. Also, not all traditional concepts may be accommodated well within a "logical" context.

This paper presents models for a logic computer system and its two constituents, a logic operating system and a logic inference machine. Possible designs for the logic computer system are explored. The underlying hardware is assumed to be a Concurrent Prolog machine. Two alternate views of the computer system and peripheral devices are examined. The paper suggests that the operating system can be implemented by accumulating clauses from progressive model refinement. Further, a method is described for the operational initialization ("bootstrapping") of the entire logic computer system. The same language, Concurrent Prolog, is used as implementation, specification, and machine language.

Language characteristics utilized within the work – guards, dataflow control, etc. – are not peculiar to Concurrent Prolog, but are present in a number of other concurrent logic programming languages. Consequently, the ideas developed in this paper are applicable, for the most part, to related languages such as PARLOG [ClGr84a] and GHC [Ueda85].

The remainder of this section introduces the language Concurrent Prolog and surveys related work. Section 2 describes the abstract Concurrent Prolog machine and general characteristics of the operating system. Section 3 concerns specification of the logic computer system and discusses contrasting designs. The initialization of the logic computer system is the subject of Section 4. Section 5 concludes the paper and suggests areas of further study.

1.1. Language

The best-known logic programming language, Prolog [Rous75, Warr77], is a poor candidate as a specification, systems programming, and machine language: it does not allow the expression of concurrent computations without resort to side-effects. In this work, Concurrent Prolog [Shap83a], hereafter denoted "CP", is used because it is powerful, concise, and supports concurrent computation. Many effective programming constructs and techniques, such as objects, class hierarchies, stream communications, and message-passing can be cleanly realized using the language [Shap83a, Shap83c, ShTa83, TaFu83, HiCF84]. CP has also been employed in a wide variety of applications [FuTK83, Hira83, ShSh83, HeSh84, Kusa84a, ShMi84]. On the whole, results favor its

use for systems programming, and for expression of high-level or complex concepts. The kernel language, KL1, of ICOT's parallel inference machine is a superset of CP [FKTU84]. A prospective architecture for a CP machine has also appeared [Shap83b].

It is assumed that the reader is familiar with CP. An in-depth description, including a computational model, is provided by Shapiro [Shap83a]. Papers by Shapiro [Shap83c] and Shapiro and Takeuchi [ShTa83] provide summaries. A condensed introduction is also provided in Appendix A. The following table summarizes the manner in which CP embodies familiar computational concepts.

Concept	Concurrent Prolog Construct
Process	Unit goal
System	Conjunctive goal
Process state	Value of arguments
Process computation	Goal reduction
Process communication	Unification of shared variables
Process synchronization	Suspending unification of read-only variables

1.2. Related Work

Shapiro [Shap83c] has demonstrated the feasibility of CP as an operating system kernel language. In the paper, a number of common, representative operating system functions are implemented in CP. A high-level specification of an operating system with a "reboot" capability is given by way of a concise CP program. The operation of a peripheral device is described as a CP process, with the device content regarded as an argument in the process state. It is proposed that the cleanest way to achieve communication with peripheral devices in a CP machine is to have devices consume or generate CP streams.

The use of formal logic, particularly temporal logic, in the specification and verification of computer components is not uncommon. Prolog has also been employed in this capacity [UeKa83]. Suzuki [Suzu83] uses CP as a specification and verification tool. His work, however, concerns only hardware components, and not software or high-level system characteristics.

Presentations of ICOT's prototype inference machine PSI [UYYT83, YYTN83] and its operating system, SIMPOS [HaYo83, TYUK84], do not make use of logic programs in their high-level descriptions. Furthermore, the design of PSI is not suited to the proliferation of small-sized processes characteristic of CP. The machine language, KL0, and the implementation language, ESP, are forms of Prolog with depth-first search, backtracking, and "cut" [Chik83]. Hence, some fundamental aspects of SIMPOS are incompatible with a CP environment.

2. Hardware and Operating System Characteristics

2.1. Abstract Machine

The logic computer system is targetted for a CP machine. Since such hardware does not exist, it is necessary to define and assume hardware properties and capabilities. However, logic inference machines are still in the very early stages of their evolution. Many architectural proposals exist in the literature with little consensus on their relative

merits. Hence, the hardware description is general in nature and independent of implementational details.

As demonstrated by the ICOT's PSI [YYTN83], a logic inference machine can display conventional features, such as interrupts, sequential execution, and reliance on side-effects. However, current trends toward more "intelligent" hardware often place functionality previously assigned to lower levels of an operating system within hardware. This again is demonstrated by PSI which has firmware instructions to handle process switching, creation, deletion, and synchronization [UYYT83].

It is only natural for a logic inference machine to have characteristics of a higher level than those of conventional computers. Logic programming languages are high-level languages. A machine which executes such a language would certainly have greater capabilities and complexity.

The hardware model is as follows. A CP machine has a multi-processor configuration, consisting of an arbitrary number of individual processing elements. Each process of a conjunctive goal system can be thought of as executing on an individual processor. "Generic processors" can undertake the reduction of arbitrary goals. The machine is responsible for mapping processes to available processors[1]. Hardware supports the efficient access and propagation of shared variable bindings. In this discussion a "processor" is a generic processor, unless stated otherwise.

Each physical peripheral device has associated with it a special "device processor". This processor provides an interface between the remainder of the CP machine and the device. Viewed by other processing elements, a device processor supports a single, characteristic, perpetual process called a "device process" (DP). A DP is logically indistinguishable from other CP processes, and describes the operation of a device processor and peripheral device without recourse to side-effects. The state or contents of the device is represented by process arguments. Software access to a peripheral device (through its device processor) is achieved by communicating with the corresponding DP using CP streams. Device processes may vary in specific protocol details. A device process exists independently of the operating system processes; it exists whenever its device processor is active.

A device processor may be any type of machine. It must, however, support an interface consistent with the remainder of the CP machine. Its execution as a perpetual process must be describable by a CP program. The actual program is dependent on the specification of the entire logic computer system (as discussed in Section 3). For instance, a terminal display may have associated with it a device process described by

```
tty_display( [Char|CharStrm], DisplayedChars ) :-
    tty_display( CharStrm?, [Char|DisplayedChars] ).
```

Program (a): Terminal Display Device Process

Here, the first argument is an input stream of characters. The second argument, the local state, represents the characters that appear on the physical display.

The machine language of the CP machine may be CP, in which case its operation is described by a meta-interpreter. Alternatively, the hardware may execute a logic-based language in which higher-level logic-based systems programming languages can be

---

1. Shapiro [Shap83b] presents an alternate view in which CP programs are augmented with process-to-processor mapping notations. The concepts presented in this paper would not be adversely affected by such a change.

specified (cf. ESP and KL0 [Chik83]). A CP interpreter is then written in this low-level language, but viewed as part of the machine. In either case, unification and goal reduction are provided by the machine model.

2.2. Operating System

The operating system design follows principles of multi-process structuring (program structuring using multiple concurrent processes). Several characteristics of CP make this approach attractive: large numbers of small processes, easily attained interprocess communication, dynamic process creation and destruction, and the ability to share data structures among processes.

The logic operating system is composed of small, complementary, and cooperating servers. Each server provides a compact set of related services to other processes. Servers are constructed as CP objects [ShTa83], or as object hierarchies. They may dynamically create and destroy constituent processes. Servers communicate via object-based protocols using message-passing over streams. They may call on the utilities of devices and other servers in their operation. Progressively more substantive services are generated in this manner. Clients normally communicate directly with the server responsible for the utility being sought. Servers may be transient (dynamically created to fill a temporary need, then removed) or permanent (created at system initialization for the duration of system execution).

The operating system does not include a kernel. A process abstraction (process creation, execution, and destruction) is already provided by goal reduction. Unification provides communication, data transfer, and synchronization mechanisms. Hence, capabilities traditionally ascribed to a kernel are captured by the metalanguage and incorporated into the CP machine model.

In a conventional operating system, the bulk of the software cannot access physical I/O hardware directly. A device driver is introduced to provide an interface. Here, directly accessible devices are provided by the machine model. Clients may access a peripheral device by communicating with its corresponding device process. The operating system need only assist in identifying the appropriate stream. Servers are typically present to provide an alternate interface or additional functionality. During system initialization (discussed in Section 4), the operating system obtains a channel to each device process. These channels are preserved for the duration of system execution.

The logic operating system is not designed for a particular proposed or prototype inference machine. Rather, it only presumes the previous basic hardware model. Any CP machine, or emulator, conforming to the model should be capable of executing the operating system program.

3. Specification of the Logic Computer System

The logic computer system model has two components, a hardware (machine) model and an operating system model. The latter builds upon the machine model. The hardware model, in turn, is an extension of the computational model of the chosen logic-based machine language, in this case CP.

A logic computer system can be represented by a goal

logic_computer_system.

Specification of the system can be viewed as the process of resolving this goal. The resulting proof tree represents progressive refinements in the operating system and hardware models. To illustrate, the following clauses could describe the overall system:

```
logic_computer_system :-
   disk( DiskStrm? ),
   tty_keyboard( TtyKeyStrm? ),
   tty_display( TtyDispStrm? ),
   operating_system( [DiskStrm,TtyKeyStrm,TtyDispStrm] ) |
   true.
logic_computer_system :-
   otherwise | logic_computer_system.
```

Program (b): Logic Computer System Specification

This concise program specifies the components of the system, and the existence and style of communication channels between peripheral devices and the operating system. It also describes operational characteristics of system initialization (to be discussed in Section 4). Clauses for the subgoals *disk(DiskStrm?)*, *tty_keyboard(TtyKeyStrm?)*, and *tty_display(TtyDispStrm?)* provide more detail regarding these components of the hardware model. The operating system model is further developed by clauses for the subgoal

operating_system(DeviceStrmList)

(see Program (e) of Section 4, for example). The same language, CP, is used throughout. The accumulation of clauses from recursive refinement of the operating system model is a program which implements the operating system.

Execution of the computer system also corresponds to construction of the proof tree rooted with goal *logic_computer_system*. By nature of the application, the proof (execution) never terminates successfully; a computer system is intended to be always executing (cf. perpetual processes[Warr82]).

Program (b) succinctly specifies many properties of a logic computer system:

a) The components of the system are a disk (file storage device), terminal keyboard, terminal display, and operating system all functioning simultaneously.

b) Communication between the operating system and each device is over a single stream. Message transfer is initiated by the operating system. Given the most intuitive producer / consumer assignments for the relationships among the system components, individual exchanges are eager [TaFu83, HiCF84] between operating system and disk, and operating system and display. They are lazy between operating system and keyboard. Despite the one-way nature of the communication channels, message replies can be realized easily using incomplete messages [Shap83c, ShTa83].

c) Subgoals describing devices and the operating system are placed within the guard of the first clause. This guard system represents the computation normally being executed. Failure of one of these subgoals causes the resolution of the entire guard to be abandoned, and computation to proceed using the second clause (the semantics of *otherwise*) [ShTa83]. This alternate clause, however, simply re-invokes the goal *logic_computer_system*, restarting the previous computation, and hence the entire computer system. The system is thus said to "restart (reboot) on failure".

d) Since the subgoals *disk(DiskStrm?)*, etc. are within a guard, failure of any of them causes the abandonment of the entire computation represented by the clause. Prior to failure, changes to the state (contents) of devices are represented in the histories (streams) bound to *DiskStrm*, etc. Upon failure, results of the attempt to resolve the guard, including the bindings of these streams, are all abandoned. When the second

clause succeeds, the resolution of *logic_operating_system* begins again, but as if the computation preceding the error had never taken place. To correctly reflect the correspondence between procedural and declarative semantics, devices must therefore be "restarted". This means operationally resetting the device to the state it was in (or an equivalent) at the start of the failing guard computation. Hence, devices are "restartable".

e) A grave software error which results in failure of the subgoal *operating_system(. . .)* – an operating system "crash" – causes the system to be reinitialized as described in c) and d) [Shap83c].

f) Serious hardware errors, which would be expected to operationally require reinitialization of the system, can cause exactly that: they can be treated as failure of the goal representing the malfunctioning device. The effect on the system is demonstrated in c) and d) above. Thus hardware errors can be handled cleanly within the logical framework.

g) Power failure, whether deliberate or unforeseen, can be treated as serious hardware error. Another subgoal, *power_up*, could be added to the first guard system in Program (b). Resolution of this predicate suspends while adequate power levels are sustained, but fails if they decline. Subsequent reduction, that using the second clause, is seen to suspend until power is again available.

h) A manual restart capability (for control by humans) could be implemented as temporary cessation of power, or as a separate signal taken as indicating goal failure.

Alternate specifications of the computer system are possible. For example, each subgoal representing a device could have as an extra argument the initial state of the device. A more contrasting example is:

```
logic_computer_system :-
   disk( DiskIn, DiskOut? ),
   tty_keyboard( TtyKeyIn, TtyKeyOut? ),
   tty_display( TtyDispIn, TtyDispOut? ),
   operating_system( [DiskIn?, DiskOut, TtyKeyIn?, TtyKeyOut, TtyDispIn?, TtyDispOut] ).
```

Program (c): Alternate Logic Computer System Specification

Though it may not appear so at first glance, the computer system specified by this program is much different in character from the previous one. In particular:

i) Communication between operating system and devices is still over CP streams. However, separate input and output streams are used. More care in synchronization of messages is therefore necessary. Further, the placement of read-only annotations (assuming significance to the variable names) implies that the generation of messages is lazy [TaFu83, ClGr84a, HiCF84]. For instance, a request cannot be sent to *disk* until the device process partially instantiates *DiskIn*. Similarly, *disk* cannot generate an output message until *operating_system* or one of its subprocesses partially instantiates *DiskOut*.

j) There is no provision for reinitialization; failure of a goal means failure and termination of the entire system. Once initiated, the computer system is "perpetual".

k) Devices are perpetual; that is, the result of the computation is never "undone" as in the case of a restartable device (see item d)). Certain devices, such as file storage, are naturally conceptualized as perpetual. A logic computer system which has any perpetual component must itself be perpetual.

l) The subgoal representing the operating system cannot be allowed to fail. Therefore, the operating system must be very robust and able to always intercept subgoal failure. Techniques for this are known (cf. failure within a user shell program [Shap83c, ClGr84b]).

m) As the system is perpetual, hardware errors cannot be treated as high-level goal failure. They can, however, be represented by suspension[2]. For example, if a hardware error occurs in a device, it may be treated as suspension of the goal reduction representing the device. Operationally, it is the responsibility of the offending physical hardware to re-establish its state to that immediately preceding the error before the device computation can be seen to continue.

As demonstrated, different characteristics are possible for the logic computer system. These characteristics are concisely specified by CP programs. The two logic computer system examples are consistent with the hardware and operating system descriptions given in Section 2.

Certain combinations of properties of the previous two programs are problematic. For example, a naive mixture of "restartable" and "perpetual" devices is not viable because of the commit operator's effect on the propagation of variable bindings. That is, in the program

```
logic_computer_system :-
   restartable_part( CommonStrms ),
   perpetual_part( CommonStrms ).

restartable_part( [DiskStrm] ) :-
   tty_keyboard( TtyKeyStrm? ),
   tty_display( TtyDispStrm? ),
   operating_system( [DiskStrm,TtyKeyStrm,TtyDispStrm] ) |
   true.
restartable_part( CommonStrms ) :-
   otherwise | restartable_part( CommonStrms ).

perpetual_part( [DiskStrm] ) :-
   disk( DiskStrm? ).
```

Program (d): Inconsistent Logic Computer System Specification

any bindings made to *DiskStrm* by resolution of the goal

operating_system([DiskStrm,TtyKeyStrm,TtyDispStrm])

will not be known to *disk* prior to commitment. But commitment to a clause to reduce the goal

restartable_part([DiskStrm])

(i.e. the second clause) occurs only after failure of the guard computation which generated the bindings to *DiskStrm* (the first one). Therefore, the *disk* process never receives any messages. Making

operating_system([DiskStrm,TtyKeyStrm,TtyDispStrm])

a subgoal of

2. The suspension of goal reduction is a fundamental capability in CP. In fact, Shapiro's original computational model for CP [Shap83a] treats goal failure as infinite suspension.

perpetual_part([DiskStrm])

alters the symptoms, but does not rectify the underlying problem.

3.1. Representation of Errors

Since the CP machine has a distributed architecture, representing a hardware error as suspension is easier in some respects than representing it as goal failure. With the goal-failure scheme, knowledge of an error cannot remain local and must be distributed to, and acted upon, by processors responsible for other goals of the current conjunctive system. The error-as-suspension approach allows knowledge of an error occurrence to remain restricted to a single device processor.

Error-as-suspension can also be used for restartable devices, in particular for less serious errors. For example, the computer system should not be reinitialized just because the lineprinter is suddenly out of paper. It is preferable to consider the lineprinter as suspended in its response to the message which motivated the error condition. The device process will be seen to continue after paper is added.

Errors of less gravity can be handled by message replies for both restartable and perpetual devices. For instance, output of character *Char* may be achieved by sending the message *out(Char,Reply)* to the terminal display. To indicate that there was a problem in doing this, the terminal display could bind *Reply* to the constant *error*.

3.2. Perpetual versus Restartable Devices

Certain peripheral devices are more naturally conceptualized as perpetual devices. For example, with file storage the most up-to-date state (content) should always be maintained. Restartable file storage would require that on reinitialization the entire informational content of the device be eliminated, reverting back to some initial state. However, to be useful file storage must be nonvolatile across hardware error, power failure, and other sources of reinitialization. It seems best, then, that file storage be perpetual.

Most other peripheral devices can be conceptualized as either perpetual or restartable. For example, on reinitialization the screen of a restartable terminal display can be cleared, reestablishing an initial state; a line printer can generate a page eject to ensure that any output will be at the top of clean paper. However, even though characters have disappeared from the screen, they were present at some specific point in time with certain characters preceding and following; the fact that the line printer generated a particular page of output cannot be later refuted. Therefore, in a more abstract sense, these devices can also be regarded as perpetual. With this view, the initial state of a restartable device is actually an equivalence class of states. For a terminal display, for instance, all members of the class may be represented by clear screens.

The initial states of restartable devices are not restricted to those given in the examples. Instead of a clear screen, the initial state of a terminal display could, for instance, involve having the string "wake me" displayed in the lower right corner. The initial state of a restartable file storage device could include predetermined files and their contents.

4. Initialization of the Logic Computer System

A device process exists independently of the operating system servers; it exists whenever its device processor is active. The purpose of system initialization is to initiate the permanent servers (see Section 2.2) of the operating system, and establish

communication channels to each device process. Communication via shared variables is declaratively simple (see Programs (b) and (c)). Its practical use, however, requires operational initialization.

The following is a simple but effective mechanism for establishing communication channels from the operating system to each device process. It assumes that the CP machine represents variables as pointers into memory accessible by all processors and globally addressable (not necessarily global, multi-ported memory). The logic computer system is taken to be of the style in Program (b); in particular, devices are restartable, a single stream exists to each device, and communications are initiated by the operating system. Finally, it is assumed that each device process is able to accept a message of the form *init(DeviceType)* and respond by unifying *DeviceType* with a ground term identifying its type.

On initialization, each device processor has a separate, predetermined variable that it tries to access, waiting for it to be instantiated. N I/O devices, *device1* through *deviceN*, are assumed to exist. The "number" of each device is set by physical manipulation[3]. The first N variable addresses are used by the N devices. Device i tries to access the ith variable. The operation of each device – *deviceM* is used as an example – at this point is describable as resolution of the goal

deviceM(DeviceM?).

One processor, not a device processor, is designated the "initialization processor"[4]. It begins the resolution of a predetermined (firmware) goal:

operating_system([Device1, . . . , DeviceM, . . . , DeviceN]).

The variable addresses for *Device1* through *DeviceN* are known by the previous convention. However, the operating system does not presuppose which variable will be used for which device. Computation proceeds using the following program:

3. This setting is analogous to the vector or CSR address in the DEC PDP-11 architecture.
4. This designation and the number of devices, N, can be set in a variety of ways, from firmware memory values to hardware jumpers.

```
operating_system( DeviceStrmList ) :-
   init_server( DeviceStrmList?, DeviceResp ),
   permanent_servers( DeviceResp? ).

init_server( [DeviceStrm|DeviceStrmList], DeviceResp ) :-
   establish_comm( DeviceStrm, EstCommStrm ),
   merge( EstCommStrm?, RespStrm?, DeviceResp ),
   init_server( DeviceStrmList?, RespStrm ).
init_server( [], [] ).

establish_comm( DeviceStrm, RespStrm ) :-
   send( init( DeviceType ), DeviceStrm, NDeviceStrm ),
   establish_comm( DeviceType?, NDeviceStrm, RespStrm ).
establish_comm( DeviceType, DeviceStrm, RespStrm ) :-
   wait( DeviceType ) |
   send( reg_device( DeviceType, DeviceStrm ), RespStrm, [] ).

permanent_servers( RespStrm ) :-
   file_system_servers( FSServerStrm ),
   . . .
   user_servers( UserServerStrm ),
   merge( [ReqStrm?,FSServerStrm?, . . . , UserServerStrm?], StrmServerReq ),
   stream_server( StrmServerReq?, [] ).

stream_server( [reg_device( DeviceType, DeviceStrm )|ReqStrm], ServerDB ) :-
   stream_server( ReqStrm?, [avail(DeviceType?,DeviceStrm)|ServerDB] ).
.
.
.
```

Program (e): Operating System Initialization

At some point in this computation, the subgoal

establish_comm(DeviceStrm, RespStrm)

must be resolved for each device *i*. Consider the general case, device *M*, for which *DeviceStrm* is bound to *DeviceM*. Resolution of the goal

establish_comm(DeviceM, RespStrm)

causes *DeviceM* to be bound to *[init(DeviceType)|NDeviceM]* and a process

establish_comm(DeviceType?, NDeviceM, RespStrm)

to be invoked. This last process suspends awaiting instantiation of *DeviceType*. It has been arranged that *DeviceM* is a variable shared by the *deviceM* and *operating_system* processes. Therefore, its binding is also known by *deviceM*. The device processor for device *M* has been awaiting just such an instantiation. The program describing its operation contains a clause similar to

```
deviceM( [init( example_type )|ReqStrm] ) :-
   deviceM( ReqStrm?, initial_content ).
.
```

(*example_type* would actually be replaced by an atom identifying the type of this device; for example, *terminal_display* or *line_printer*. Likewise, *initial_content* would be the initial state of the device.) Reduction of the goal

deviceM([init(DeviceType)]¦NDeviceM]?)

succeeds, binding *DeviceType* to *example_type*. The suspended *establish_comm* process can now resume execution. It places information necessary for further communications with device *M* in an outgoing message destined for *stream_server*. Communication between device *M* and the operating system is now established. The new variable *NDeviceM* is known to both parties and will be used for the next exchange. The establishment of communication at system initialization, then, is primarily a matter of coordination.

The following points can be made regarding initialization and Program (e):

a) The clauses for predicates *permanent_servers* and *stream_server* are given in outline form for purposes of discussion. *send*, *merge*, and *wait* are assumed to be self-explanatory, though descriptions are given in Appendix B.

b) The memory locations for variables *Device1* . . . *DeviceN* have no special properties. It may even be possible, given sophisticated tail recursion optimization and garbage collection techniques, to reuse them for other variables.

c) Reduction of the subgoals in Program (e) can migrate to idle processors. The initialization processor is only required to start the computation.

d) A device process need not retain the capability to handle an *init(DeviceType)* message once communications with the operating system processes have been established.

e) Not only does Program (e) specify how system initialization takes place, it is also a refinement of the operating system model. For example, the program initiates, and the operating system is composed of, a set of permanent servers and a transient server to aid in initialization. Certain predicates, such as *establish_comm*, require no further elaboration, whereas the bulk of the operating system is described by the clause for *permanent_servers* and its subgoals.

f) The high-level specification of the operating system is independent of the number and types of peripheral devices in the computer system.

g) The stream server is an important permanent server. Its purpose is the maintenance of associations between identifiers (of objects) and communication variables to these objects. On receipt of a message *reg_device(DeviceType,DeviceStrm)*, it adds to its database the information "*DeviceStrm* is the stream to device *DeviceType*".

h) The operating system is initiated in such a way that the unexpected absence of a device processor does not create severe problems. The most significant consequence would be an *establish_comm* process suspended, awaiting a reply from a non-existent DP. The rest of the system can carry on. This also means that the operating system can be started expecting more devices than are actually present. New devices can easily be added at a later point in time without restarting operations.

Unfortunately, Program (e) may be too idealistic and impractical, at least given conventional techniques for initializing computer systems. It is implicit that the entire operating system program is present within the machine at the start of operation. A more typical situation has the operating system program stored on a file-structured device[5]. The initially executed program – the (primary) bootstrap – is minimal and stored in ROM. Its sole purpose is to read into main memory a larger program and begin its

5. Though this is the norm, it need not be. The development of novel computer architectures allows the questioning of such forms of conventional wisdom.

execution. These operational considerations require changes to Program (e) which interfere with the correspondence between model refinement and initialization procedure. However, through a conscious effort and a language extension, the interference can be minimized. The following is an example:

```
operating_system( DeviceStrmList ) :-
  boot_server( DeviceStrmList?, DeviceResp, OSProg ),
  prove( permanent_servers( DeviceResp? ), OSProg? ).

boot_server( DeviceStrmList, DeviceResp, OSProg ) :-
  contact_devices( DeviceStrmList, RespStrm ),
  boot_from_fsd( RespStrm?, OSProg, DeviceResp ).

contact_devices( [DeviceStrm|DeviceStrmList], DeviceResp ) :-
  establish_comm( DeviceStrm, EstCommStrm ),
  merge( EstCommStrm?, RespStrm?, DeviceResp ),
  contact_devices( DeviceStrmList?, RespStrm ).
contact_devices( [], [] ).

boot_from_fsd( RespStrm, OSProg, DeviceResp ) :-
  receive( reg_device( fsd, FSDStrm ), RespStrm, NRespStrm ) |
  send( access( permanent_servers, OSProg ), FSDStrm, NFSDStrm ),
  send( reg_device( fsd, NFSDStrm ), DeviceResp, NRespStrm ).
boot_from_fsd( [Resp|RespStrm], OSProg, [Resp|DeviceResp] ) :-
  otherwise |
  boot_from_fsd( RespStrm?, OSProg, DeviceResp ).
```

Program (f): Operating System Bootstrap

The logic computer system is assumed to include a file system device (FSD) [Kusa84b] in which the remainder of the operating system program is stored. The clauses for *establish_comm* are as in Program (e). *otherwise* and *receive* are familiar CP predicates (a description is given in Appendix B). The new metalogical predicate *prove* is similar to *call* of PARLOG [ClGr84b]. Its definition is an application of the work of Bowen and Kowalski [BoKo82]. Declaratively the goal

```
prove( Goal, Prog )
```

succeeds if *Goal* is provable from program *Prog*. Its resolution suspends until both its input arguments are instantiated.

The *contact_devices* process is equivalent to *init_server* of Program (e). The role of *boot_from_fsd* is to monitor responses from *establish_comm* processes on stream *RespStrm*, watchful for the one identifying the file system device (FSD)[6]. Upon arrival of this response, a request to access the file with identifier *permanent_servers* is sent to the FSD, a replacement *reg_device* response is inserted into the output response stream, and the process terminates. All other responses on *RespStrm* are passed through unaltered. It is assumed that the file identified by *permanent_servers* contains all clauses necessary for the reduction of the goal

---

6. The FSD is a device process which provides the basic services of creation, access, removal, and stable storage of files [Kusa84b]. In response to a request *access(FName,FContent)* the device process unifies *FContent* with the current contents of the file identified by *FName*. Both *FName* and *FContent* are arbitrary terms, though *FName* must be ground.

permanent_servers(DeviceResp)

i.e. the bulk of the operating system.

5. Concluding Remarks

This paper has presented models for a logic computer system, and its hardware and software components. It has demonstrated that CP programs can be used to concisely specify a logic computer system, its operating system, and operation of peripheral devices. Examples with significantly different characteristics were given and compared.

A method for operationally initializing a logic computer system was presented. As demonstrated, the correspondence between model refinement and operating system initiation need not be adversely affected by the necessity of a bootstrap. In Program (f), most of the complications are contained within the specification of *boot_server*.

It is noteworthy that concepts such as hardware error and reinitialization do not complicate the declarative reading of Programs (b) and (c). These concepts are inherently operational and are handled within that component of the language and computer system models.

5.1. Further Study

Several areas of further study are immediately apparent:

- Errors can be represented by suspension for both perpetual and restartable devices. However, this requires that, following the error, a device continue from the precise point of preemption. Implementationally, this should not be difficult to approximate. It is not clear, however, that it can ever be precisely attained.
- Certain devices, such as file storage, are best conceptualized as perpetual. However, hardware errors cannot be represented as conjunctive goal failure for perpetual devices. The error-as-suspension scheme may also be unworkable because of the problem mentioned above. Therefore, other means of handling hardware errors should be investigated.
- It may be feasible to declaratively account for a restartable file storage device which operationally retains its contents on system reinitialization. The idea of an oracle presents one possibility.
- Because of Program (d), it may be taken that restartable and perpetual devices cannot both be present within a single logic computer system. This may not necessarily be the case. The metalogical predicate *prove* (similar to *call* of PARLOG [ClGr84b]) offers several possibilities.

Preliminary investigations suggest interesting results in these areas.

As Programs (b) and (c) suggest, computer systems with a wide variety of characteristics can be specified. As further study, systems with varying properties can be developed, explored, and compared. Techniques for operationally initializing these systems can also be investigated. Other issues which can be explored include protection and security and user-programmable error handling.

Acknowledgements

This work benefitted greatly from the encouragement, criticism, and suggestions of Harvey Abramson. Also, Mats Carlsson provided valuable comments.

Appendix A - Introduction to Concurrent Prolog

CP [Shap83a] facilitates the expression of concurrency, communication, synchronization, and indeterminacy by a minimal extension to the basic computational model of logic programs. The language is based on the Relational Language of Clark and Gregory [ClGr81]. In CP, as opposed to Prolog, the AND- and OR-parallelism of the theoretical model[CoKi81] of logic programs is retained. A conjunctive goal can be regarded as a system of processes, a unit goal being an individual process. The state of a process is the value of a goal's arguments, and the state of a system is the union of the states of its processes. Concurrency among processes is the AND-parallelism of the theoretical model. The OR-parallel trial of candidate clauses provides each process with the ability to perform indeterminate actions. Variables shared between goals serve as the process communication mechanism. Synchronization is achieved by denoting which processes can write a variable (instantiate it to a non-variable term).

CP introduces two constructs to the model of logic programs: read-only annotations of variables and the commit operator. Read-only variable references, $X?$ where X is a variable, are used to constrain the order and pace of process reduction. Commit, denoted by '|', permits both "committed choice" and "don't care" nondeterminism.

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

$$H :- G_1, \ldots, G_m \mid B_1, \ldots, B_n \qquad m,n \geq 0$$

where the G_i's and the B_j's are atomic formulae (unit goals). H is the clause head and the G_i's form the guard. The guard may be empty, in which case the commit operator is omitted. Read-only variable references may appear within any part of a clause.

The semantics of a guarded-clause

$$H :- G \mid B$$

are as follows. Declaratively, read-only annotations are ignored and the commit operator reads as a conjunction: H is true if G and B are true. Operationally, the clause is similar to an alternative in a guarded-command [Dijk76]. To reduce a process H' using the clause above, H and H' are unified, G is recursively reduced to the empty system, commitment is made to this clause, and H' is reduced to B. The reduction may suspend or fail at any of these steps. Unification of H and H' suspends if it requires the instantiation of variables annotated as read-only. It fails if H and H' are not unifiable. The reduction of the guard system G suspends if the processes in it all suspend, and fails if any of them fails. Commitment may fail if variable bindings generated by the guard computation conflict with those generated by other (concurrent) computations.

The semantics of the commit operation require that variable bindings produced by the first two steps of reduction – unification of H and H' and reduction of G – are accessible only to processes in G, or their descendants, prior to the commitment. Also, as part of commitment, all other OR-parallel attempts to reduce H' are abandoned.

As a programming aid, CP contains the metalanguage predicate *otherwise* [ShTa83]. A single *otherwise* goal in a guard – the only manner in which it can be used – succeeds if and when all other OR-parallel guards fail.

Appendix B - Commonly Used Concurrent Prolog Predicates

The following is a list of commonly used CP predicates employed in the programming examples. A description is given for each.

B.1. System Predicates

wait(X)

waits until the principle functor of its argument, *X*, is determined, then terminates with success [Shap83a].

otherwise

this predicate may only be used as a single subgoal in a guard. It succeeds if and when all of its brother OR-parallel guards fail. Declaratively, it may be read as the negation of the disjunction of the guards of the brother clauses [ShTa83].

B.2. User-Definable Predicates

merge(In1,In2,Out)

computes the relation "*Out* contains the elements of *In1* and *In2*, preserving the relative order of their elements". The predicate may demonstrate various operational properties, depending on its precise definition and utilization of operational characteristics of the language implemenation [Shap83a, Kusa84a, ShMi84, UeCh84, ShSa85].

send(Msg,Strm,NStrm)

names the relation "the result of sending *Msg* on stream *Strm* is the stream *NStrm*" [Shap83c].

receive(Msg,Strm,NStrm)

names the relation "the result of receiving *Msg* on stream *Strm* is the stream *NStrm*" [Shap83c].

References

BoKo82.

K. A. Bowen and R. A. Kowalski, "Amalgamating Language and Metalanguage in Logic Programming," in *Logic Programming*, ed. K. L. Clark and S.-Å. Tärnlund, pp. 153-172, Academic Press, London, England, 1982.

Chik83.

T. Chikayama, "ESP - Extended Self-contained Prolog - as a Preliminary Kernel Language of Fifth Generation Computers," *New Generation Computing*, vol. 1, no. 1, pp. 11-24, Tokyo, 1983.

ClGr81.

K. L. Clark and S. Gregory, "A Relational Language for Parallel Programming," *Proceedings of the ACM Conference on Functional Programming Languages and Computer Architectures*, pp. 171-178, ACM, October 1981.

ClGr84a

K. L. Clark and S. Gregory, "PARLOG: Parallel Programming in Logic," Research Rep. DOC 84/4, Department of Computing, Imperial College, London, April 1984.

ClGr84b

K. L. Clark and S. Gregory, "Notes on Systems Programming in PARLOG," Research Rep. DOC 84/15, Department of Computing, Imperial College, London, July 1984.

CoKi81.

J. S. Conery and D. F. Kibler, "Parallel Interpretation of Logic Programs," *Proceedings of the ACM Conference on Functional Programming Languages and Computer Architecture*, pp. 163-170, ACM, Portsmouth, New Hampshire, October 18-22, 1981.

Dijk76.

E. W. Dijkstra, *A Discipline of Programming*, Prentice-Hall, Englewood Cliffs, New Jersey, 1976.

FKTU84.

K. Furukawa, S. Kunifuji, A. Takeuchi, and K. Ueda, "The Conceptual Specification of the Kernel Language Version 1," Technical Report, ICOT, Tokyo, 1984.

FuTK83.

K. Furukawa, A. Takeuchi, and S. Kunifuji, "Mandala: A Concurrent Prolog Based Knowledge Programming Language / System," TR-029, ICOT, Tokyo, November, 1983.

HaYo83.

T. Hattori and T. Yokoi, "Basic Concepts of the SIM Operating System," *New Generation Computing*, vol. 1, no. 1, pp. 81-85, Tokyo, Japan, 1983.

HeSh84.

L. Hellerstein and E. Y. Shapiro, "Implementing Parallel Algorithms in Concurrent Prolog: The MAXFLOW Experience," *1984 International Symposium on Logic Programming*, pp. 99-115, IEEE, Atlantic City, NJ, February 6-9, 1984.

HiCF84.

H. Hirakawa, T. Chikayama, and K. Furukawa, "Eager and Lazy Enumerations in Concurrent Prolog," *Proceedings of the Second International Logic Programming Conference*, pp. 89-100, Uppsala, Sweden, July 2-6, 1984.

Hira83.
H. Hirakawa, "Chart Parsing in Concurrent Prolog," TR-008, ICOT, Tokyo, Japan, May 1983.

Kahn82.
K. M. Kahn, "Intermission - Actors in Prolog," in *Logic Programming*, ed. K. L. Clark and S.-Å. Tärnlund, pp. 213-228, Academic Press, London, England, 1982.

Kusa84a.
A. J. Kusalik, "Bounded-Wait Merge in Shapiro's Concurrent Prolog," *New Generation Computing*, vol. 2, no. 2, Springer-Verlag, Tokyo, Japan, 1984.

Kusa84b
A. J. Kusalik, "The File System of a Logic Operating System," Technical Report 84-21, Computer Science Department, University of British Columbia, Vancouver, B.C., Canada, November 1984.

Rous75.
P. Roussel, "Prolog: Manuel de Reference et d'Utilisation," Technical Report, Groupe d'Intelligence Artificielle, Université d'Aix Mareille, 1975.

Shap83.a
E. Y. Shapiro, "A Subset of Concurrent Prolog and Its Interpreter," TR-003, ICOT, Tokyo, Japan, January 1983. Also as CS83-06, Department of Applied Mathematics, Weizmann Institute of Science, Rehovot, Israel.

Shap83b
E. Y. Shapiro, "(Lecture Notes on) The Bagel: a Systolic Concurrent Prolog Machine," TM-0031, ICOT, Tokyo, Japan, November 1983.

Shap83c
E. Y. Shapiro, "Systems Programming in Concurrent Prolog," TR-034, ICOT, Tokyo, Japan, November 1983.

ShMi84.
E. Y. Shapiro and C. Mierowsky, "Fair, Biased, and Self-Balancing Merge Operators: Their Specification and Implementation in Concurrent Prolog," *1984 International Symposium on Logic Programming*, pp. 83-90, IEEE, Atlantic City, NJ, February 6-9, 1984.

ShSa85.
E. Y. Shapiro and M. Safra, "Fast Multiway Merge Using Destructive Operations," CS85-01, Department of Applied Mathematics, Weizmann Institute of Science, Rehovot, Israel, January 1985.

ShSh83.
A. Shafrir and E. Y. Shapiro, "Distributed Programming in Concurrent Prolog," CS83-12, Department of Applied Mathematics, Weizmann Institute of Science, Rehovot, Israel, August 1983.

ShTa83.
E. Y. Shapiro and A. Takeuchi, "Object Oriented Programming in Concurrent Prolog," *New Generation Computing*, vol. 1, no. 1, pp. 25-48, Tokyo, 1983.

Suzu83.
N. Suzuki, "Experience with Specification and Verification of Complex Computer Using Concurrent Prolog," University of Tokyo, July 1983.

TaFu83.
A. Takeuchi and K. Furukawa, "Interprocess Communication in Concurrent

Prolog," *Proceedings Logic Programming Workshop '83*, pp. 171-185, Algarve, Portugal, June 26 - July 1, 1983. Also as ICOT Technical Report TR-006, 1983.

TYUK84.

S. Takagi, T. Yokoi, S. Uchida, T. Kurokawa, T. Hattori, T. Chikayama, K. Sakai, and J. Tsuji, "Overall Design of SIMPOS," *Proceedings of the Second International Logic Programming Conference*, pp. 1-12, Uppsala, Sweden, July 2-6, 1984.

UeCh84.

K. Ueda and T. Chikayama, "Efficient Stream / Array Processing in Logic Programming Languages," *Proceedings of the International Conference on Fifth Generation Computer Systems 1984*, pp. 317-326, ICOT, Tokyo, Japan, November 6-9, 1984.

Ueda85.

K. Ueda, "Guarded Horn Clauses," TR-103, ICOT, Tokyo, Japan, June 1985.

UeKa83.

T. Uehara and N. Kawato, "Logic Circuit Synthesis using Prolog," *New Generation Computing*, vol. 1, no. 2, pp. 187-193, 1983.

UYYT83.

S. Uchida, M. Yokota, A. Yamamoto, K. Taki, and H. Nishikawa, "Outline of the Personal Sequential Inference Machine: PSI," *New Generation Computing*, vol. 1, no. 1, pp. 75-79, Tokyo, 1983.

Warr77.

D. H. D. Warren, "Implementing Prolog – Compiling Predicate Logic Programs," Technical Reports 39 and 40, Department of Artificial Intelligence, University of Edinburgh, Edinburgh, Scotland, May 1977.

Warr82.

D. H. D. Warren, "Perpetual Processes – An Unexploited Prolog Technique," *Proceedings of the Prolog Programming Environments Workshop*, Linkoping University, Sweden, March 1982.

YYTN83.

M. Yokota, A. Yamamoto, K. Taki, H. Nishikawa, and S. Uchida, "The Design and Implementation of a Personal Inference Machine: PSI," *New Generation Computing*, vol. 1, no. 2, pp. 125-144, Tokyo, 1983.

Zani84.

C. Zaniolo, "Object-Oriented Programming in Prolog," *1984 International Symposium on Logic Programming*, pp. 265-270, IEEE, Atlantic City, New Jersey, February 6-9, 1984.