A Declarative Environment for Concurrent Logic Programming*

Keith L. Clark and Ian T. Foster
Dept of Computing, Imperial College
180 Queen's Gate, South Kensington
London SW7 2BZ
janet: klc/itf@uk.ac.ic.doc
uucp: ...!mcvax!ukc!icdoc!klc/itf

Abstract

A logic programming environment should provide users with declarative control of program development and execution and resource access and allocation. It is argued that the concurrent logic language PARLOG is well suited to the implementation of such environments. The essential features of the PARLOG Programming System (PPS) are presented. The PPS is a multiprocessing programming environment that supports PARLOG (and is intended to support Prolog). Users interact with the PPS by querying and updating collections of logic clauses termed data bases. The PPS understands certain clauses as describing system configuration, the status of user deductions and the rules determining access to resources. Other clauses are understood as describing meta relationships such as inheritance between data bases. The paper introduces the facilities of the PPS and explains the essential structure of its implementation in PARLOG by a top down development of a PARLOG program which reads as a specification of a multiprocessing operating system.

1. Introduction

Language-based operating systems and programming environments have demonstrated the advantages of defining systems in a high-level language [Joseph et al, 1978], [Wulf, 1981], [Sandewall, 1978]. Such systems are most effective when the high-level language's computational model is capable of representing operational aspects of the underlying computer system and at the same time is close in spirit to that of the languages to be supported by the system. It must also be efficiently implementable.

This paper describes a logic programming environment implemented in a concurrent logic language. This is the PARLOG Programming System, or PPS [Foster, 1986a]. This programming environment aims to provide low-level support for declarative programming in logic languages by providing simple and expressive environment structures that are accessible as logic clauses with a declarative reading.

The PPS is implemented in PARLOG [Clark and Gregory, 1984a]. This language has been found to be particularly well-suited to the implementation of logic progamming environments. Of particular importance is its powerful control meta call, which permits a PARLOG program to initiate and control execution of other programs. A second aim of this paper is therefore to introduce and illustrate the use of PARLOG and its meta call for implementing programming environments. In this respect the paper is a sequel to [Clark and Gregory, 1984b]

The user views the PPS as a set of data bases. This set includes data bases that he defines himself and others defining system and program structure or containing libraries of useful program components. The PPS also maintains data bases representing more volatile information such as the history of user interaction, data structures constructed by query evaluations and currently active program evaluations. All information is available in declarative form and can be accessed both by the user and by other logic programs.

The user interacts with this set of data bases using a single mechanism: the *query*. The user initiates evaluation of a query in a PPS data base (in other words, inititates execution of a program) using the following syntax:

<data base> : <query>

This requests that <query> be evaluated in <data base>. For example:

utilities: analyse(my_db, A)

The user can use this mechanism to run any program defined in a PPS data base.

The PPS, which runs on Sun workstations, provides a window interface to facilitate user interaction. The user enters queries in the *console window*. Each valid query that is entered is allocated a unique identifier and a *query interface window*. The user is subsequently able to interact with a query using its interface window.

The paper introduces the facilities of the PPS and explains the essential structure of its implementation in PARLOG by a top down development of a PARLOG program which reads as a specification of a multiprocessing operating system. Even so, the PARLOG program presented is only a slight simplification of the actual PPS implementation.

The paper assumes familiarity with the general concepts of logic programming and Prolog, but it does briefly introduce PARLOG in section 2. Section 3 describes the basic structure of the PPS and presents the core of its implementation. Sections 4 and 5 introduce the PPS's meta programming and program structuring tools, presenting the extensions to the core implementation that they require. Section 6 describes how the PPS handles resource access. Finally, related work is surveyed and conclusions presented.

2. PARLOG

PARLOG differs from Prolog in three important respects: concurrent evaluation, 'don't-care non-determinism' and its use of mode declarations to specify communication constraints on shared variables. Each relation call in a PARLOG conjunction can be evaluated concurrently as a separate process. Shared variables act as communication channels along which messages are sent.

2.1 Don't-Care Non-Determinism

A PARLOG clause is a Horn clause optionally augmented with a commit operator, ':', which is used to separate the right hand side of the clause into a conjunction of guard conditions and a conjunction of body conditions:

$r(t_1, \ldots, t_k)$ <- <guard conditions> : <body conditions>

where $t_1, \ldots, t_K$ are argument terms.

Both the <guard conditions> and the <body conditions> are conjunctions of relation calls. There are two types of conjunction: the parallel '//' (C1 // C2) in which the conjuncts C1 and C2 are evaluated concurrently and the sequential '&' (C1 & C2) where C2 will only be evaluated when C1 has successfully terminated.

In the evaluation of a relation call $r(t_1', \ldots, t_K')$, all of the clauses for the relation r will be searched in parallel for a candidate clause. The above clause is a candidate clause if the head $r(t_1, \ldots, t_K)$ matches the call $r(t_1', \ldots, t_K')$ *and* the guard succeeds. It is a non-candidate if the match or the guard fail. If all clauses are non-candidates the call fails, otherwise one of the candidate clauses is selected and the call is reduced to the substitution instance of the body of that clause. There is no backtracking on the choice of candidate clause. We 'don't care' which candidate clause is selected. In practice, the first one (chronologically) to be found is chosen.

The search for a candidate clause can be controlled by using either the parallel clause search operator '\\' or the sequential clause search operator ';' between clauses. If a relation is defined by the clauses:

```
Clause1 \\
Clause2 ;
Clause3.
```

Clause3 will not be tried for candidacy until both Clause1 and Clause2 have been found to be non-candidate clauses.

Often the programmer will not care whether clauses and calls are evaluated sequentially or concurrently. PARLOG therefore supports a neutral conjunction operator ',' and a neutral clause search operator '..'. These are compiled to either parallel or sequential operators, depending on the granularity of the parallelism supported by the target machine.

2.2 Modes

Every PARLOG relation definition has a mode declaration associated with it, which states whether each argument is input (?) or output (^). For example, the relation merge(X,Y,Z) has the mode (?, ?, ^) to merge lists X and Y to list Z:

```
mode merge( X?, Y?, Z^ ).
merge( [ E | X ], Y, [ E | Z ] ) <- merge( X, Y, Z ) \\
merge( X, [ E | Y ], [ E | Z ] ) <- merge( X, Y, Z ) \\
merge( [ ], Y, Y ) \\
merge( X, [ ], X ).
```

(Dec-10 Prolog syntax [Clocksin and Mellish, 1981] is used throughout this paper. Upper-case letters denote variables. Lower-case letters, integers and strings enclosed in single quotes denote constants.)

Concurrently evaluating relation calls communicate via shared variables: the modes impose a direction on this communication. Non-variable terms that appear in input argument positions in the head of a clause can only be used for input matching. If an argument of the call is not sufficiently instantiated for an input match to succeed, the attempt to use the clause suspends until some other process further instantiates the input argument of the call. For example, the first clause for merge has [E | X] in its first input argument position. Until the call has a list or partial list structure of the form [E | X] in the first argument position the first clause is suspended.

If all clauses for a call are suspended, the call suspends. A candidate clause can be selected even if there are other, suspended, clauses.

2.3 PARLOG's Process Interpretation

A logic program can generally be given both a *declarative* and a *procedural* interpretation. Its declarative interpretation indicates how it represents knowledge, whilst its procedural interpretation indicates how it can be used to solve problems. Concurrent logic languages permit a third interpretation, a *process* interpretation, which is useful when explaining the operational behaviour of certain programs in these languages. This interpretation appears particularly relevant to systems programs written in these languages.

The process interpretation of logic was first described in [van Emden and Lucena, 1982]. When applied to concurrent logic languages, it allows programs to be viewed as defining networks of communicating processes. Reduction modifies this network. A PARLOG relation:

```
P <- A // B
```

thus describes the replacing of a process P by two new processes, A and B. Tail-recursive relations define long-lived (or *perpetual*) processes. Non-shared argument terms can be regarded as describing local state. Variables that are shared between two or more processes define communication channels. For example, in a conjunction:

```
user( Requests ) // server( Database, Requests )
```

the user process may pass messages of the form query(Query, Result) to server by appending them to the Requests stream. Result is assumed to be a variable. server receives these messages and processes Query relative to its local state, Database. It can then instantiate the Result variable to true or false to indicate to user the result of the query. The passing of the Result variable to server and its subsequent instantiation is an example of a programming technique first illustrated in [Shapiro, 1984], who termed it *incomplete messages*. This paper uses the alternative description, *back communication*, introduced in [Clark and Gregory, 1984a]. It is a very powerful feature of concurrent logic languages.

The process interpretation, though operational in nature, is thus a useful descriptive tool. In particular, it indicates how PARLOG can be used to represent changing state without side-effecting data base operations: perpetual processes can iterate with a different local argument. server can for example also process update messages, which generate a new local state NewDatabase:

```
mode server( Database?, Requests? ).

server( Database, [ query( Query, Result ) | Requests ] ) <-
        process_query( Database, Query, Result ) ,
        server( Database, Requests) ..
server( Database, [ update( Update, Result ) | Requests ] ) <-
        process_update( Database, Query, Result, NewDatabase ) ,
        server( NewDatabase, Requests ).
```

The implementation of the PPS, described in the next section, makes extensive use of perpetual processes to represent entities whose state may be subject to change over time.

2.4 The PARLOG Control meta call

The use of PARLOG for systems programming and other applications where one PARLOG program must control the execution of another is greatly facilitated if a control meta call is available. Clark and Gregory originally proposed a three-argument control meta call primitive [Clark and Gregory, 1984b] and showed how it could be used both for systems programming and to reduce or-parallel evaluation of guards of alternative clauses to and-parallel evaluation. Foster has proposed an extended five-argument meta call primitive for general systems programming work [Foster, 1986b] and it is this form of the call that is used in the PPS implementation of PARLOG.

The PARLOG control meta call is a sophisticated meta control mechanism that enables PARLOG

meta programs to invoke the PARLOG machine on arbitrary goals and then interface to the monitoring and control functions of the machine with streams. A call to this primitive has the general form:

call(Module?, Resources?, Goal?, Status^, Control?)

The primitive initiates an attempt to evaluate Goal using the code in Module with bounded resources Resources. The call only fails if its arguments are invalid. Otherwise it generates a Status stream of messages about the evaluation and it consumes a stream of Control messages. The meta program that initiated execution of Goal can use these streams to monitor and control the execution of Goal. call can also be invoked without the Resources or Module arguments, or, as in Prolog with only the Goal argument.

The meta call accepts Control stream messages **stop, suspend, continue** and **resources**. The first three cause the goal evaluation to be stopped, suspended and resumed respectively. The fourth modifies the resources allocated to the goal evaluation.

The meta call may generate Status stream messages **failed, succeeded, stopped, suspend, continue, exception(_)** and **exception(_,_,_)**. The first three of these represent termination states of the Goal evaluation and are produced as the meta call succeeds. (Note that the meta call succeeds even when Goal fails.) **suspend** and **continue** are echoed when the corresponding Control stream messages are received. The single-argument exception message is generated when various sorts of exceptions occur, such as deadlock or excessive use of resources. Its argument is bound to the type of exception that occurred and the goal evaluation is suspended. The three-argument exception message is generated when what are termed *pseudo-exceptions* [Foster, 1986b] occur. This message has the general form exception(Type, Goal, NewGoal). Type indicates the sort of exception that occurred, for example, a call to a relation not defined in Module. Goal is the goal that caused the pseudo-exception. This goal is replaced with the new goal call(NewGoal), which then suspends waiting for the variable NewGoal to be bound.

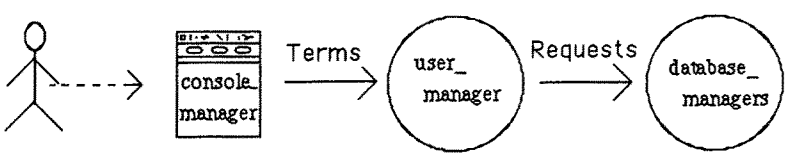
The pseudo-exception message allows a monitoring meta program to process pseudo-exceptions in a number of ways. It may instantiate NewGoal to false, which has the effect of failing the call that caused the exception. Alternatively, for an undefined relation exception, signalled by a Type value undefined, the monitor program may instantiate NewGoal to a meta call that attempts to solve Goal in another data base: eval(OtherModule, Goal), where OtherModule names some other object code module. Subsequent sections of this paper will indicate how this meta call feature is exploited in the implementation of the PPS.

3. The Implementation of the PPS

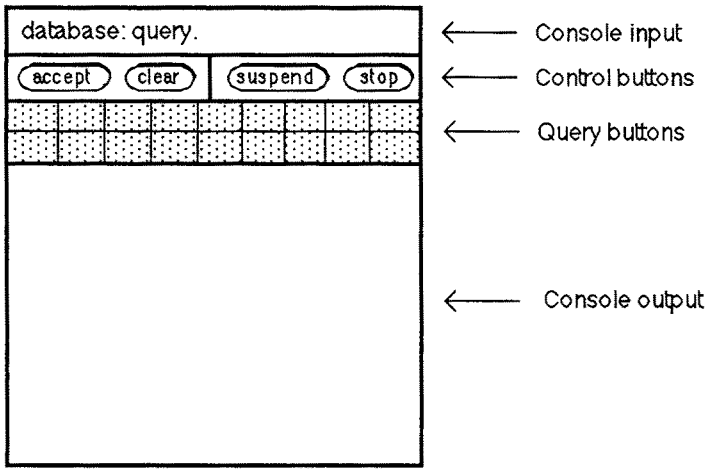
The PPS is a network of communicating PARLOG processes. The system is booted by a call to the relation pps with a top level definition:

```
pps <- console_manager( Terms ) //
       user_manager( Terms, Requests ) //
       database_managers( Requests).
```

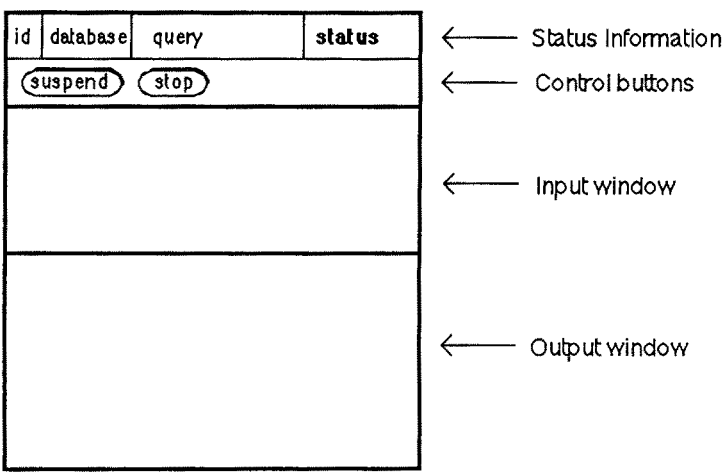
which sets up the initial network:



The console_manager accepts user input entered in a special console window. The console window has the layout:



The user enters queries in the console input sub-window or selects query buttons which encode certain common queries. Recall that a query has the form database: query. A query can be a request to evaluate some conjunction of calls or a *meta query* request to access or update a definition in a data base. The console manager thus generates a stream of PARLOG terms representing queries to named data bases. The user_manager processes these terms, spawning a *query manager* for each query it receives. The query manager will send an initial request on the Requests stream for permission to evaluate the query in the named data base. It then monitors the evaluation of the query, passing out on the Requests stream subsidiary queries to other data bases that may be generated by the evaluating query. It sends out a query termination signal when the query finally terminates. The query manager also creates a new query interface window through which the user can interact with the evaluating query. This has the layout:



The database_managers process will spawn a manager for each data base currently defined in

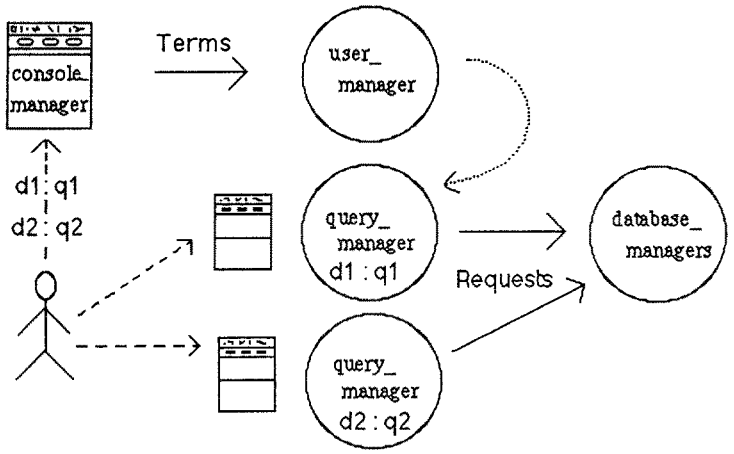
the PPS. The request stream from the user_manager (and the individual query managers that is spawns) carries all the access and update messages to the individual data base managers. Access to a data base is controlled by a data base manager in order to prevent the data base being updated whilst a query is using a relation defined in the data base. The request stream carries the messages for every data base, so the messages are tagged with the name of the destination data base. They are routed to the the correct data base manager by a message switching process which is also spawned by the database_managers process. Routing via a switch process enables any query manager to send a message to any data base by simply placing the appropriate tagged message on its output request stream. Back communication - the binding of variables in the message - is used to send the data base manager response directly back to the requesting query manager.

user_manager has the following definition:

```
mode user_manager( Terms?, Requests^ ).

user_manager( [ ':'( Database, Query ) | MoreQueries ], Requests ) <-
        query_manager( Database, Query, Requests1) //      % spawn query manager and
        merge( Requests1, Requests2, Requests ) //         % merge the requests from this
        user_manager( MoreQueries, Requests2 ) ..          % query with those of subsequent
                                                           % queries
user_manager( [ ], [ ] ).
```

Note that the processing of subsequent queries by the user_manager is executed in parallel with the evaluation of the first query. The user can therefore have several concurrently evaluating queries. Each query manager spawned will generate a stream of requests to access or update a particular data base. The various request streams generated are merged to give the single Requests stream that is passed to the message switching process created by the database_managers process. If two queries are entered by the user, two query managers will be spawned. Note that the user can independently interact with the console_manager and the two queries via their respective interface windows. The process communication network will be:



3.1 The Query Manager

The query manager initiates and controls the execution of a query in a data base. If the PPS only allowed a single program to be executed at a time, or did not permit updates to be made to data bases, the query manager could handle normal queries by simply looking up the name of the loaded module containing the compiled defintions for the data base and then executing the query via a meta call. No output request stream to a collection of data base managers would be needed and the query manager could

have a definition of the form:

```
mode query_manager( Database?, Query? ).

query_manager( Database, Query ) <-
        lookup( Database, Module ) ,
        query_panel( Panelin, Panelout, Database, Query ) //
        q_m( Panelin, Panelout, Database, Query, Status, Control ) //
        call( Module, Query, Status, Control ).
```

query_panel is a process which sets up and interacts with the query interface window and q_m is a query monitor process that interfaces between the meta call query evaluation and the query_panel control process using the meta call Status and Control streams and the Panelin, Panelout input and output streams of query_panel. It handles all the status and exception messages sent out on the Status stream of the meta call, reporting to the user where necessary via a message sent on Panelout. It also handles user input, such as button commands to suspend or stop the query, by reflecting these messages it receives from query_panel on the Panelout stream onto the meta call Control stream. A query manager of this form is just a slight generalisation of the shell programs presented in [Clark & Gregory, 1984b].

However, in a multiprocessing system which also allows users to update data bases the above query manager is too simple. It would be possible to initiate a second query that modified a relation definition currently being used in a previously initiated query. Though this is unlikely in a single-user system, the PPS design is intended to form the basis for a multi-user system in which update/use contention is much more likely. It is also logically incorrect to allow programs to be modified whilst they are being used. It is far better to cleanly separate the meta level operation of update from the object level operation of evaluation.

The PPS prevents update/use contention by requiring all requests to execute or modify progams in a data base to pass via a manager for the data base. The requests sent to the data base manager are of the form:

```
message( Database, Request, Result )
```

where Database is the destination data base, Request is the request and Result is a back communication variable. In the case of a normal request to evaluate a query the message term is of the form:

```
message( Database, query( Query, Done ), Result ).
```

and Result will generally be bound by the database manager for Database to the response term:

```
eval( Module, Query )
```

indicating that the unmodified query is to be evaluated using the compiled definitions in Module, the module corresponding to Database. More generally, the response could be a modified query to be evaluated relative to some other module, thus allowing for query transformation and virtual program structures. Such possibilites have not yet been explored in the PPS.

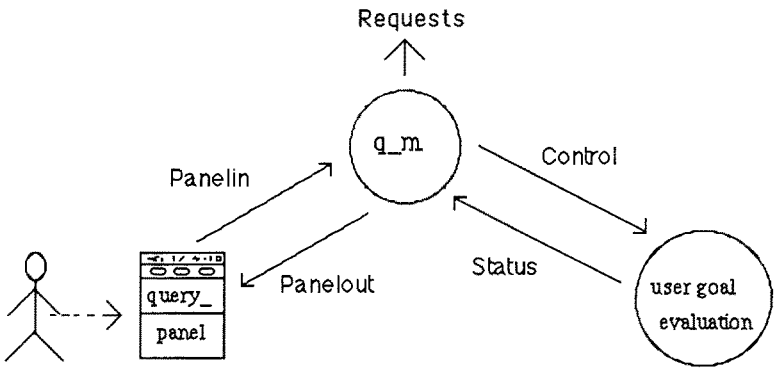
The Done component of the communicated message is initially an unbound variable which will serve to carry the query termination signal to the manager for Database. This variable, and the corresponding termination signal variables for all other concurrent queries to Database, are retained by its data base manager. The Done signal variable is bound to the constant done by the query manager of Query. The data base manager will only allow an update to precede if all the signal variables of the concurrent queries have been bound by their query managers. This is the lockout mechanism that prevents update during use.

The PPS query_manager relation is actually defined as follows:

```
mode query_manager( Database?, Query?, Requests^ ).

query_manager( Database, Query, [ message(Database, query(Query, Done), Result) | Reqs ] ) <-
        query_panel(Panelin, Panelout, Database, Query ) //
        q_m( Done, Panelin, Panelout, Database, Reqs, Status, Control ) //
        call( Result, Status, Control ).
```

This relation defines a process network that can be represented as follows:



The query monitor q_m is defined as:

```
mode q_m( Done^, Panelin?, Panelout^, Database?, Requests^, Status?, Control^ ).

q_m( Done, Panelin, Panelout, Database, Requests, Status, Control ) <-
        data( Panelin ) :
        user( Done, Panelin, Panelout, Database, Requests, Status, Control ) \\
q_m( Done, Panelin, Panelout, Database, Requests, Status, Control ) <-
        data( Status ) :
        status( Done, Panelin, Panelout, Database, Requests, Status, Control ).
```

Note the use of the data primitive in the above program. This is a PARLOG primitive which suspends if its argument is an unbound variable, and succeeds for any other argument. In the above program it delays the branch to either the user or the status subprograms until a message is received on either the user input stream Panelin or the meta call Status output stream.

3.1.1 Handling User Input

On the query interface window monitored by the query_panel process the user may select buttons to *suspend* or *stop* the evaluation of the query. Another button is created when the query is suspended that allows the user to *continue* evaluation. When one of these buttons is selected the query_panel process sends out a corresponding message on its Panelin stream. The suspended q_m process therefore reduces to a call to the user relation. This echoes the message received on Panelin onto its Control stream, which is the control input stream of the meta call call(Result,Status,Control) of the query_manager. This causes the query to suspend, continue or stop as required. user is defined as:

```
mode user( Done^, Panelin?, Panelout^, Db?, Requests^, Status?, Control^ ).

user( done, [ stop | Panelin ], Panelout, Db, Requests, Status, [ stop | Control ] ) ..
user (Done, [ suspend | Panelin ], Panelout, Db, Requests, Status, [ suspend | Control ] ) <-
        q_m( Done, Panelin, Panelout, Db, Requests, Status, Control ) ..
```

```
user( Done, [ continue | Panelin ], Panelout, Db, Requests, Status, [ continue | Control ] ) <-
        q_m( Done, Panelin, Panelout, Db, Requests, Status, Control ).
```

Note that the Done variable is instantiated if the query is stopped so that data base manager for the Db data base knows that this query has terminated.

3.1.2 Status Messages

The query monitor may receive status messages from the evaluating meta call informing it that the user query has succeeded, failed, or is suspended due to an exception. Status messages suspend and continue may also be received. These are echoed by the meta call onto its status stream when these control messages are received on its control stream. They can be ignored by the status stream monitor. Status messages are processed by the status relation:

```
mode status( Done^, Panelin?, Panelout^, Db?, Requests^, Status?, Control^ ).

status( done, Panelin, [ succeeded ], Db, [ ], [ succeeded | Status ], Control ) ..
status( done, Panelin, [ failed ], Db, [ ], [ failed | Status ], Control ) ..
status( done, Panelin, [ Type ], Db, [ ], [exception( Type ) | Status ], [ stop | Control ] ) ..

status( Done, Pin, Pout, Db, Req, [exception( Type, Goal, NewGoal ) |Status], Control ) <-
        exception(Done, Pout, Db, Req, Type, Goal, NewGoal, NewPout, NewReq ) ,
        q_m(Done, Pin, NewPout, Db, NewReq, Status, Control) ;

status( Done, Panelin, Panelout, Db, Requests, [ Other | Status ], Control ) <-
        q_m( Done, Panelin, Panelout, Db, Requests, Status, Control ).
```

The first three clauses handle status messages that announce termination of the query for one reason or another. On receiving the message the status process binds the Done signal variable to the constant done so that the data base manager for the Db data base knows that this query has terminated. The message is echoed on the Panelout message stream to be reported to the user in the query interface window. In the case of the single argument exception message, which remember signals that the query is suspended due to such conditions as deadlock and memory limitation, a stop message is appended to the control stream to terminate evaluation. Note that the output Request stream to the data base managers is also terminated.

The fourth clause calls a relation exception to process pseudo-exception messages before recursively calling q_m with potentially modified Panelout and Requests arguments. This allows the exception handler to send messages to the user or some data base manager whilst handling the exception.

The fifth clause is a default clause which ignores the other status messages. Note the essential use of the sequential clause-search operator between clauses 4 and 5 to ensure that the last clause is treated as a default clause.

3.2 Simple Exception Handling

Recall that the three argument exception message is used to report conditions such as an attempt to evaluate a relation not defined in the current module. A very simple exception handler would just report the exception to the user and then cause the query to terminate as though the user had entered a **stop** command. A slightly more sophisticated exception handler might instead display the call to the user and allow the user to enter an alternative call. The user could then enter false, to force failure of the call to the undefined relation, or he could select the stop button to abort the evaluation.

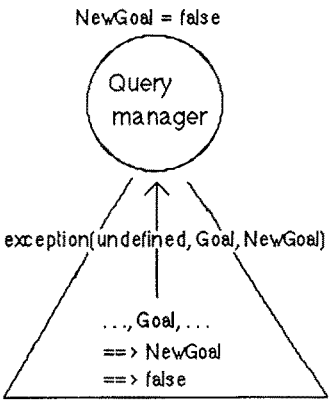
An exception handler that passes calls to undefined relations to the query_panel process to allow for such a response from the user can be defined as follows:

```
mode exception( Done?, Panelout^, Db?, Requests^, Type?, Goal?, NewGoal^, NewP?, NewR?).

exception( Done, [ input(Goal, NewGoal) | Panelout ], Db, Requests, undefined, Goal,
                                                    NewGoal, Panelout, Requests) ;
exception( Done, [ output( [exception,Type, in,Goal] ) | Panelout ], Db, Requests, Type, Goal,
                                                    false, Panelout, Requests) .
```

The first clause processes calls to undefined relations by passing an *input* message to the query_panel process. On receiving the message input(Goal,NewGoal), query_panel will display Goal in the query interface window and bind NewGoal to the input entered by the user. The second clause is a catch-all clause that processes all other pseudo-exceptions by instantiating NewGoal to false and sending an *output* message to the query control panel to inform the user of the exception.

Recall tbat the PARLOG exception handling mechanism which generates the exception messages exception(Type, Goal, NewGoal) also replaces the goal Goal with the variable NewGoal in the user program. Instantiating NewGoal immediately causes the user program to resume execution by evaluating NewGoal. Suppose that NewGoal is instantiated to false. This situation can be represented as follows:



The query manager monitoring the user computation receives the exception message, determines that Goal should be failed, and instantiates NewGoal to false. The user process evaluating Goal is thus replaced with a call to false.

Section 5 presents a much more sophisticated exception handler that allows calls to relations undefined in a module to be evaluated relative to another module. This exception handler uses the special PPS program structuring data which links program modules.

3.3 Database Managers

The source and compiled code for the current PPS data bases are stored on a *logical disk* [Foster and Kusalik, 1986]. This is essentially an indexed term storage device. Access to the logical disk is controlled by a disk server, which processes the *retrieve* and *store* messages that are used to retrieve and store information on the disk. The term with index 0 on the logical disk is always a list of terms of the form db(Database, SourceId, ObjectId) giving the names of all current PPS data bases as well as the logical disk indexes of the source and object code for these data bases.

The call database_manager in the pps intialisation program invokes the disk server and sends it the message retrieve(0, DbList) to access term 0 to retrieve the list of db terms. It then calls an auxilary relation db_ms to spawn one data base manager process per data base. It also spawns the switch process to route the messages received from query managers to the appropriate data base managers:

```
database_managers( Requests ) <-
        disk_server( [ retrieve(0, DbList) | Disk ] ) //      % Retrieve list of data bases.
        db_ms( DbList, Disk, SwitchList ) //                  % Spawn data base managers.
        switch( Requests, SwitchList ).                       % Spawn switch (described below).
```

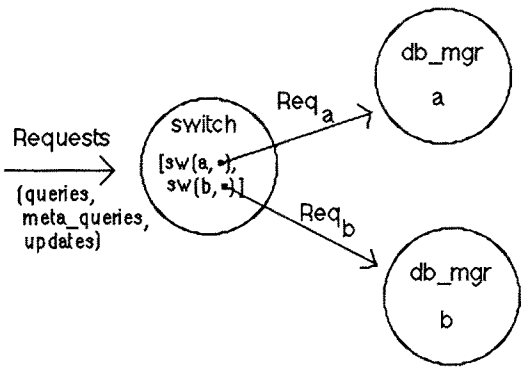
The process db_ms recurses down the DbList returned by the disk server, spawning a data base manager, a call to db_mgr, for each data base on the list.

```
mode db_ms( DbList?, Disk^, SwitchList^ ).

db_ms( [ db( Db,Srcld, Objld ) | DbList ], [ retrieve( Srcld, Source ), retrieve( Objld, Object ) | Disk ],
                                                    [sw( Db, ReqStream ) | RestofSwitchList ] ) <-
        ( load( Object, Module ) &
        db_mgr( ReqStream, Db, Source, Module, [ ] ) ) //
        db_ms( DbList, Disk, RestofSwitchList ) ..
db_ms( [ ], [ ], [ ] ).
```

Each db_mgr spawned has an input request stream ReqStream which is initially an unbound variable. The switch process will send messages to this data base manager by routing them from its input Requests stream to the ReqStream for the Db named in the message. The db_ms process therefore also contructs a switch list of terms of the form sw(Db,ReqStream) which is output to the switch process. db_mgr also has as arguments the name of the data base it is managing, the Source code for the data base returned by the retrieve(Srcld,Source) message sent to the disk_server, and the name of the in-core Module into which the object code has been loaded by the load call. Notice that the sequential connective after the load call delays the spawning of db_mgr until the load has terminated. This prevents the manager allowing a query to start evaluating before the object code has been fully loaded.

Assume disk term 0 is a list of two db terms. Then the process network created by data base_manager is as follows:



A data base manager receives messages in the form message(Request, Result), where Request may be a query, meta_query or update request.

Normal query requests are of the form:

```
query( Query, Done )
```

and will usually be handled by binding the Result variable to the term:

```
eval( Module, Query ).
```

which is then evaluated by the meta call of the query manager that sent the message. Alternatively, if some form of access control needs to be supported, messages can be augmented with user passwords and Result could be instantiated to a call to the primitive raise_exception to inform the user that access to this data base is denied.

Remember that the Done variable will be bound by the query manager when the query terminates. The list of all the Done variables for the active queries is held as the last argument of the db_mgr process. Hence the argument is initialised to the empty list when the manager is invoked.

Meta queries are sent to a data base manager when a user program calls one of the special meta relations described in the next section. They allow the user to access and analyse source programs. A meta query message has the form meta_query(MetaQuery) and is processed by the data base manager by invoking the auxiliary relation:

```
process_meta_query( MetaQuery, Result, Source, NewDone )
```

This is passed the Result variable of the incoming message and the Source for the data base. It processes the MetaQuery relative to the Source data base, binding Result to true or false depending on whether the meta query succeeds or fails. (In addition it may also bind variables in the MetaQuery term of the message.) NewDone is a fresh synchonisation variable which process_meta_query will bind to the constant done when the meta query has terminated in exactly the same way that the Done variables of normal queries are bound on termination by the query managers. It will prevent an update to the source before the meta query has terminated.

Updates are processed similarly, but in addition they lead to changes to the local Source argument of the manager process. The in-core module Module containing the data base's compiled code is also modified. Updates are only allowed if there are no currently active queries using the compiled code or accessing the source of the data base.

```
mode db_mgr ( InStream?, Db?, Source?, Module?, DoneVars? ).

db_mgr( [ message(query(Query, Done), Result) | InStream ], Db, Source, Module, DoneVars ) <-
        Result = eval( Module, Query ) //
        db_mgr( InStream, Db, Source, Module, [ Done | DoneVars ] ) ..
db_mgr( [ message(meta_query(MetaQuery), Result) | InStream ], Db, Source, Module, DoneVars ) <-
        process_meta_query( MetaQuery, Result, Source, NewDone) //
        db_mgr ( InStream, Db, Source, Module, [NewDone | DoneVars] ) ..
db_mgr( [ message(update(Update), Result) | InStream ], Db, Source, Module, DoneVars ) <-
        process_update( Update, Result, Db, Source, Module, DoneVars, NewSource ) &
        db_mgr( InStream, Db, NewSource, Module, [ ] ).
```

If there any active queries, that is, if DoneVars has any unbound variable, process_update can either delay the update or refuse the update and bind Result to false. A guard call to the relation active_queries(DoneVars) can be used to delay the update. This relation suspends if any variable on DoneVars is unbound. This will delay the update until all queries and meta queries that are accessing this database have terminated.

```
mode active_queries( DoneVars? ).

active_queries( [ Done | DoneVars ] ) <-
        data( Done ) &
        active_queries( DoneVars ) ..
active_queries( [ ] ).
```

3.4 Message Passing in the PPS

The switch process is defined in PARLOG as follows:

```
mode switch( InStream?, OutStreams^ ).

switch( [ message( To, Message, Result ) | InStream ], SwList ) <-
        perform_switch( To, Message, Result, SwList, NewSwList ) ,
        switch( InStream, NewSwList ).

mode perform_switch( To?, Message?, Result^, SwList?, NewSwList^ ).

perform_switch( To, Message, Result, [ sw( To,Stream ) | SwList], [ sw(To, NewStream ) | SwList ] ) <-
        Stream = [ message( Message, Result ) | NewStream ] ;
perform_switch( To, Message, Result, [ Other | SwList ], [ Other | NewSwList ] ) <-
        perform_switch(To, Message, Result, SwList, NewSwList ) ..
perform_switch( To, Message, false, [ ], [ ] ) .
```

Each time a message message(To, Message, Result) is received from a query manager, perform_switch is called to recurse down the list of sw(DatabaseName, Stream) terms. If it finds the data base named To it appends the message message(Message, Result) to its stream. Otherwise it instantiates Result to the constant false to indicate failure to route the message.

This use of a switch process provides very flexible communications. Query managers (and hence users) can communicate with any data base without possessing an explicit stream to that data base. A simple extension of the above switch program will allow new data bases to be created easily at run-time. On receiving a create_new_database message, a new data base manager process can be invoked and its request stream added to the switch list of sw(data baseName, Stream) pairs held by the switch process.

The switch process may appear to be a potential bottleneck. However, the table lookup implemented in the above program by a recursion down the switch list can be implemented as a special language primitive. Moreover, the merges required to concentrate the requestsstreams from the different query managers passed into the switch can also be efficiently implemented as language primitives [Shapiro and Safra, 1986]. They do not need to be implemented by the merge program given in section 2.

4. Meta Relations

An important aspect of the PPS is the support it provides for meta programs: programs that reason about other programs. This support is provided by *meta relations*. These are relations which, when encountered in a user program, are dispatched by the query manager to the database that they want to access. They permit a program to access other programs as data.

Logic can be used to represent knowledge in two ways: as terms and as relations [Kowalski, 1979]. A meta program that analyses another program can thus either work on a term representing that program, perhaps structured as follows:

```
database( my_db, [ ( 'f/1', . . . ), ( 'g/1', . . . ) , . . . ] )
```

or by accessing a data base of relations:

```
definition( my_db, 'f/1', . . .).
definition( my_db, 'g/1', . . .).
. . .
```

The PPS adopts the latter approach, providing meta relations that can be viewed as extending a user program in one data base with (implicit) sets of clauses describing other data bases. Meta relations supported by the PPS include:

databases(Databases^)
    retrieves a list of all data bases defined in the PPS.
dict(Database?, Dict^)
    succeeds if Database is a data base, retrieving a list Dict of defined relations.
definition(Database?, Relation?, Definition^)
    succeeds if Relation is defined in Database, retrieving its Definition.
definition(Database?, Relation?, Time?, Definition^)
    succeeds if Relation is defined in Database at or prior to Time,
    retrieving its Definition at Time.
history(Database?, Relation?, History^)
    succeeds if Relation is defined in Database, retrieving a list History of
    timestamped definitions.

Note that meta relations such as definition retrieve a PARLOG term representing the parsed definition, not the source text of the definition. Thus if a PPS data base rectangle included the relation:

```
area(A) <- height(H) // width(W) // mul(H, W, A).
```

then the meta relation call

```
definition( rectangle, 'area/1', Definition )
```

would bind the variable Definition to the term:

```
relation( area, 1, [?], clause( [A], [ ], calls( parallel, [ height(H), width(W), mul(H, W, A) ] ) ) )
```

This is a term giving the essential structure of the definition.

4.1 The Implementation of Meta Relations

All calls to meta relations are recognised by the PARLOG compiler and converted into calls to the raise_exception primitive with arguments **primitive** and the meta relation call, so that they will be reported by the meta call as a **primitive** exception. The call is then handled by the exception handler, which needs to be redefined as:

```
mode exception( Done?, Panelout^, Db?, Requests^, Type?, Goal?, NewGoal^, NewP?, NewR? ).

exception( Done, Pout, Db, [ message( To, meta_query( Goal ), NewGoal ) | Requests ], primitive,
                                                Goal, NewGoal, Pout, Req uests) <-
    is_meta( Goal ) :
    decode_destination( Goal, To ) ..
exception( Done, Pout, Db, [ message( To, update( Goal ), NewGoal ) | Requests ], primitive,
                                                Goal, NewGoal, Pout, Requests ) <-
    is_update( Goal ) :
    decode_destination( Goal, To ) ..
exception( Done, [ input( Goal, NewGoal) | Pout ] , Db, Req, undefined, Goal, NewGoal, Pout, Req ) ;
exception( Done,  [ output( [exception, Type, in, Goal] ) | Pout ], Db, Req, Type, Goal, false, Pout, Req ) .
```

The first two clauses identify calls to meta relations and updates and generate meta query or update messages to the appropriate data base. decode_destination looks at the meta relation arguments to determine which data base the message should be sent to. The third clause passes other undefined calls to the query's interface window as before, whilst the fourth default clause fails all other exceptions.

4.2 The Application of Meta Relations

Meta relations facilitate the construction of programs such as static analysers, meta interpreters and debuggers. Programs which would be hard to write in conventional language systems are easy to write in

the PPS. For example, a program that analyses a data base, reporting relations defined but not called, relations called but not defined and meta relations called is given here. Calls to meta relations are in bold.

```
mode analyse( Database?, Analysis^ ).

analyse( Database, Analysis ) <-
        dict( Database, Dict ) ,
        calls( Database, Dict, [ ], Calls ) ,
        separate( Dict, Calls, Analysis ).

mode calls( Database?, Dict?, CallsIn?, CallsOut^ ).

calls( Database, [ ], Calls, Calls) ..
calls( Database, [Relation | Dict], Calls, Calls2 ) <-
        definition( Database, Relation, Definition ) ,
        calls_relation( Relation, Definition, Calls, Calls1 ) ,
        calls( Database, Dict, Calls1, Calls2 ).
```

The relation calls_relation walks over a relation definition, outputing all relation calls on a difference list. separate compares lists of defined and called relations and outputs lists of relations defined but not called, relations called but not defined and meta relations called.

A number of meta programs (such as analyse) are included in the PPS. These can be both called by the user directly and incorporated in user programs either by run-time import or by compile-time copying. The way in which this is done is described in the next section. New tools can thus be constructed from old. For example, the user could combine the analyse tool listed above with a program of his own to generate a new tool that performs the same job for a set of data bases:

```
mode user_analyse( Databases?, Analysis^ ).

user_analyse( [ Database | Databases ], Analysis ) <-
        analyse( Database, Analysis1 ) ,
        user_analyse( Databases, Analysis2 ) ,
        merge_analyses( Analysis1, Analysis2, Analysis ).
```

This use of tools can be compared with both Unix shell-scripts (which allow Unix tools to be combined to give new programs) and compile-time linking of standard Unix libraries. The inheritance of code from system objects in SmallTalk is a related mechanism.

5. Describing Program Structure: Meta Clauses

It has been shown how the PPS allows users to execute programs located in data bases and write programs that manipulate other programs as data. The utility of the data base would be very limited however if it were not possible to construct new programs from program fragments located in various data bases and to use the data base as a program structuring tool.

The ability to construct programs from several data bases implies also the ability to partition large programs into fragments located in distinct data bases. This allows the user to:

- separate static and dynamic (changeable at run-time) program fragments.
- structure programs to make them easier to understand and maintain.
- reuse old code.

Some mechanism is required to represent the linking of data bases. The mechanism used in the PPS is the *meta clause*: a logic clause describing where a relation that is called but not defined in a data base should be evaluated. Like module import lists, meta clauses can be used to stitch together program

components from various sources. However, meta clauses are a much more declarative representation of program structure than module import lists. PPS meta clauses are located in data bases and can thus be accessed and modified using the same mechanisms used to access and modify other programs.

5.1 Meta Clauses: Location and Application

Meta clauses are located in data bases termed *meta data bases*. These do not differ syntactically or structurally from other data bases but are distinguished as such by clauses defining the relation meta_database, located in the data base system. For example, when developing the analysis program user_analyse listed in section 4 in a data base my_db, the user might associate a meta data base my_meta with the data base. This association can be represented by an explicit clause in the data base system:

```
meta_database( my_db, my_meta )
```

my_meta might then contain the following definition of the specially recognised refer relation:

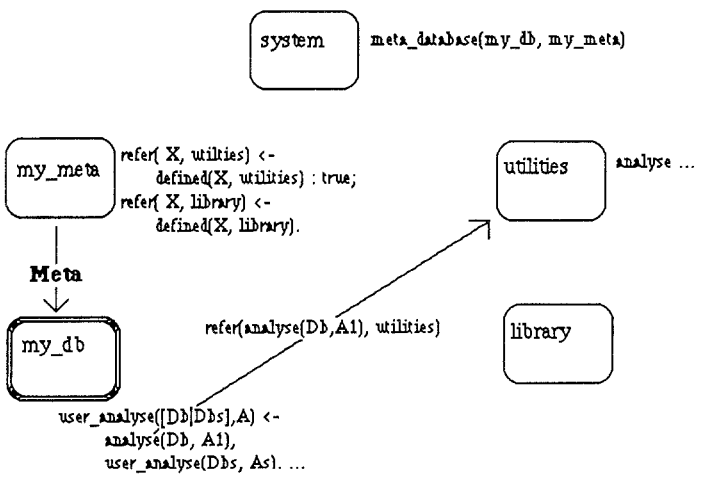
```
my_meta:
    mode  refer( Relation?, Database^ ).

    refer( X, utilities )  <-  defined( X, utilities )  :  true;
    refer( X, library )    <-  defined( X, library ).
```

where defined uses the meta relation dict to determine whether a relation is defined in a data base:

```
defined( Relation, Database ) <-
    dict( Database, Dict ) ,  on( Relation, Dict ).
```

refer(Relation, Database) is a meta clause that states that Relation should be referred to Database for evaluation. Such a meta data base has defined the program structure:



A user query such as:

```
my_db: user_analyse( [ database1, database2 ], A )
```

leads to calls to the relation analyse. The PPS refers to the meta data located in my_meta at this point and determines that these calls should be referred to the data base utilities.

Alternatively, the user might wish to link in his own version of analyse, located in the data base my_db1. This can be done by adding a new first clause:

```
refer( analyse(X, Y), my_db1 ) ;
```

to the above refer definition.

5.2 Meta Clauses: Implementation

The implementation of meta clauses requires an extension to the exception handler described in section 4. Rather than referring all calls to undefined relations to the user, the exception handler must access the meta clauses associated with the data base in which the query is being evaluated in order to determine whether the calls can be referred to another data base. If done naively, this would require a message to system to query the meta_database relation, followed by a message to the appropriate meta data base to query the refer clauses, for each call to an undefined relation. This overhead can be avoided if the system data base is queried initially to find the name of the meta data base which is then queried to find the name of its compiled code module. The meta data base module name is passed as an extra argument to the query monitor and exception handler.

The third clause in the exception program given in section 4.1 above must be replaced with the following three clauses. Note the extra MetaModule argument which is the cached name of the meta data base's compiled code module:

```
exception( Done, [ input( Goal, NewGoal ) | Panelout ], Db, MetaModule, Requests, undefined,
                        Goal, NewGoal, Panelout, Requests) <-
    call( MetaModule, refer( Goal, user ) ) : true ;
exception( Done, Panelout, Db, MetaModule,
                        [ message( ReferDb, query(Goal, Done), NewGoal ) | Requests],
                        undefined, Goal, NewGoal, Panelout, Requests) <-
    call( MetaModule, refer( Goal, ReferDb ) ) : true ;
exception( Done, Pout, Db, MetaModule, Requests, undefined, Goal, false, Pout, Requests) ;
```

The first clause has a guard which succeeds if Goal is to be referred to the user. If so, an input message to the query control panel is generated. The second clause determines whether Goal can be referred to another data base for evaluation. If it can, a query message is generated and the goal that led to the exception is replaced with the goal returned by the data base manager to which the query message is directed. This is done by passing the NewGoal variable to the data base manager in the query message. The third clause deals with goals that cannot be referred elsewhere. These are failed.

Note that the Done variable sent to the ReferDb database manager is the Done variable associated with the query when its query manager is spawned, not a new variable. The same variable is sent to the meta data base when it is initially queried to retrieve MetaModule. Update to any database referenced by a query evaluation is thus prevented during the course of the query evaluation. The only exception to this rule is the database system, which can be updated at any time that it is not being directly queried.

5.3 Query-the-User

Examples above have shown how meta clauses can specify that calls to undefined relations are to be referred both to other data bases and to the user for evaluation. Referring calls to the user implements a programming technique known in logic programming as *query-the-user* [Sergot, 1982]. This can be used both for interactive, top-down program development and for implementing declarative interactive systems.

In the PPS, calls to undefined relations that are referred to the user are displayed in the 'input' window in the query's control panel. For example, consider the query:

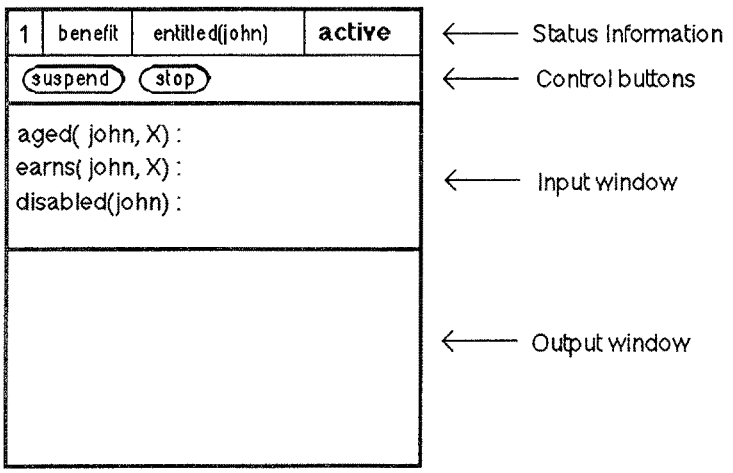
```
benefit : entitled( john )
```

where the data base benefit contains the single relation:

```
mode entitled( Person? ).

entitled( Person ) <-
        aged(Person, Age), lesseq( 60, Age ) : true \\
entitled( Person ) <-
        earns(Person, Salary ), lesseq( Salary, 2000 ) : true \\
entitled( Person ) <-
        disabled( Person ) : true.
```

which states that a person is entitled to a benefit if they are more than 60 years old, *or* they earn less than 2000, *or* they are disabled. Assume that the single meta clause refer(X, user) is linked to benefit. The query interface panel that would result from evaluating this query is:



The input window presents three queries to the user:

```
aged( john, X) ? :
earns( john, X) ? :
disabled( john) ? :
```

The user can provide answers to any of these queries by typing a new goal after the query. In the example, providing the answer true to the third query (that is, indicating that John is disabled) suffices to succeed the original query. John's age and earnings do not need to be specified. Other more powerful goals can be provided as answers. The goal X = 35, provided as an answer to the first query, indicates that John is aged 35. The user can also invoke debuggers or meta interpreters. For example, the goal evaluate_in(john_info, earns(john, X)), provided as an answer to the second query, replaces the goal with a call to a meta interpreter that attempts to solve the goal earns(john, X) using definitions located in the data base john_info.

5.4 Efficient Use of Meta Clauses

When prototyping an application it is clearly very convenient to be able to link together components of existing programs. Meta clauses allow the user to do this very easily. The overhead of evaluating meta clauses at run-time may however be excessive in a production system, particularly if imported relations are frequently used. In this case, meta clauses can be transformed or applied at compile-time to

transform an inefficient prototype into a more efficient program. This can be done at several levels.

Firstly, the meta clauses themselves can be transformed. If they define a search-path, they can be partially evaluated with respect to the current contents of the data bases in that path to give a bulkier but more efficient set of meta clauses. For example, the contents of the meta data base my_meta listed in section 5.1 above can be transformed to a set of unit clauses:

```
refer( analyse(X,Y), utilities ) ..
refer( ..., ... ) ..
etc ...
```

If the underlying implementation supports indexing, access to such a set of meta clauses will be very rapid indeed. This transformation can be compared with the use made in Berkeley Unix of hash tables to speed up directory searches.

A more powerful transformation can be applied to the object program by using the meta clauses as instructions to a compiler. This can copy imported program components to generate a new data base in which goals can be executed directly without run-time evaluation of meta clauses. This is effectively what a linker does in conventional language systems. This approach will generate an extremely efficient program, but is liable to be expensive.

A third approach recognises that programs are frequently constructed from sets of data bases that are only modified between runs, rather than at run-time. If these *data base families* are defined as such by the user and the PPS ensures that no data base in a family can be modified whilst any data base in that family is being queried, then query messages are not required within the family and can be replaced with direct calls to relations in other data bases within the family. This gives the efficiency of copying without the space overheads. Direct calls can be generated by adding linking clauses to data bases within a data base family. For example, consider the example programs in section 4.3. If the databases my_db and utilities form a family, the user_analyse program requires the addition of the linking clause:

```
analyse(Database, Analysis) <- eval(util_module, analyse( Database, Analysis) ) ,
```

to the database my_db, where **util_module** is the name of the compiled code module associated with utilities.

Direct calls to relations within a family can be combined with referral to data bases outside the family.

5.5 Other Meta Clauses

The application and implementation of the refer meta clause has been presented and it has been shown that this meta clause allows certain program structures to be described in logic. Other meta clauses are required to describe more complex program structures.

Evaluation of a call to an undefined relation that is referred to another data base may in turn lead to calls to other relations undefined in the referred-to data base. When refer clauses are used, as in the examples above, the original data base and its meta clauses are referenced to at this point. The refer meta clause thus addresses similiar problems to the 'self' message in object-oriented programming systems.

An alternative meta clause is refer_remote. This specifies that when a call to an undefined relation occurs in a referred-to data base, the meta clauses associated with the referred-to data base should be consulted. These may in turn lead to calls to other data bases which the user need not be aware of. refer_remote thus permits the modularisation of programs and the description of data base hierarchies.

6. Resource Access

A programming environment must provide user programs with access to resources such as secondary storage, i/o devices and data bases. This should be done using abstractions that enable programs to perform this access simply and elegantly. Previous sections have shown how meta relations and meta clauses allow users to access PPS data bases as source and program. This section describes how the PPS supports input and output, secondary storage and data base update.

6.1 Input/Output

Every query evaluating in the PPS has a query interface window associated with it. This has input and output sub-windows and is effectively a virtual terminal for the query evaluation. The user program can access this resource in several ways. Input and output *meta relations* can be used to access the window directly. Alternatively, *request streams* can be allocated and used to provide more controlled access to the window: the user program appends requests to streams rather than calling meta relations directly. Lastly, meta clauses can be used to refer undefined relations to the user and hence initiate *query-the-user* interactions, as described in the previous section.

6.1.1 I/O Meta Relations

Providing access to input and output resources poses problems in logic languages because the side-effecting primitives usually used to provide this access do not have a declarative reading. Concurrent logic languages provide a partial solution to this problem: a concurrent logic program can be regarded as co-operating with a user, viewed as a process, to construct a stream of i/o request terms.

One way of representing this view of i/o in a concurrent logic language is to augment the original goal with an extra argument representing the request stream. This approach has the disadvantage that every relation that could conceivably perform i/o, and the relations that call these relations, must also have this extra argument. Furthermore, the various request streams generated must be merged. This obscures the meaning of programs and introduces possible sources of programmer error. For example, a program that walks over a tree represented as a recursively defined structure tree(Node, LeftTree, RightTree), displaying all nodes:

```
mode show_tree( Requests^, Tree? ).

show_tree( Requests, tree( Root, LeftTree, RightTree ) ) <-
        show_tree( Requests1,LeftTree ) //
        show_tree( Requests2, RightTree ) //
        merge( [output( Root ) ], Requests1, Requests2, Requests ) ..
show_tree( [ ], [ ] ).
```

where merge is a fair three-way merge. The clumsiness of this approach motivated the designers of Logix [Silvermann et al, 1986] to provide syntactic sugar for appending a request to the request stream. For example:

```
kb # request
```

This is expanded to the appropriate stream operations by a precompiler. The above example can be represented as follows using this notation:

```
mode show_tree(Tree?).

show_tree( tree( Root, LeftTree, RightTree ) ) <-
        kb#output( Root ),
        show_tree( LeftTree ),
        show_tree( RightTree ) ..
show_tree( [ ] ).
```

which is easier to understand and precludes the possibility of programmer error.

The PPS takes a different approach. It is sematically equivalent to this syntactic sugar (and thus to streams and merges), however it avoids the need for streams and merges altogether by exploiting the PARLOG control meta call's exception handling mechanism. Certain i/o relations are defined to be meta relations and are thus trapped by the PPS when called in user programs. The exception handler then sends them to the query_panel process. I/o meta relations include input_term and output_term. The above program can be written as follows in the PPS:

```
mode show_tree( Tree? ).

show_tree( tree( Root, LeftTree, RightTree ) ) <-
        output_term( Root ),
        show_tree( LeftTree ),
        show_tree( RightTree ) ..
show_tree( [ ] ).
```

A call to input_term has as arguments a prompt string and a variable which is bound to the user response. The call suspends waiting for the response. If several input requests are made at about the same time, the user may respond to them in an order different from the display order in the query window. For example, the user could write:

```
. . .// input_term( 'Prompt A', A ) // input_term( 'Prompt B', B ) // undefined( X ) // . . .
```

where undefined(X) is a call to an undefined relation. All three of these calls would thus be trapped by the PPS. Assuming that a meta clause said that the undefined call was to be referred to the user, the PPS would display three input requests in the input sub-window of the query's interface window:

```
Prompt A :
Prompt B :
undefined( X ) :
```

The user can select to provide input to any (or none) of these requests.

An input_term request can be regarded as a query to a remote database in the same way as a query-the-user interaction. The prompt represents the query.

6.1.2 Explicit Sequencing of Input and Output

A disadvantage of using meta relations (or annotations such as kb#request) is that the only way to explicitly sequence input and output requests is to sequence the reduction of goals in a program. This may sometimes place unacceptable constraints on parallelism. For example, given a program:

```
mode compiler( Tokens?, Code^ ).

compiler( Tokens, Code ) <-
        parser( Tokens, ParseTree ) //
        code_generator( ParseTree, Code ).
```

it may be desired to sequence output so that diagnostics generated by parser are displayed before those generated by code_generator. If meta relations or annotations are used for output, this requires rewriting the program using a sequential conjunction operator:

```
mode compiler( Tokens?, Code^ ).

compiler( Tokens, Code ) <-
        parser( Tokens, ParseTree ) &
        code_generator( ParseTree, Code ).
```

This severely reduces the potential for parallelism in the program.

This example can be used to illustrate the advantages of explicit request streams, which can be *appended* to sequence i/o independently of the computation strategy specified for associated goals. Using explicit request streams, the example can be rewritten as:

```
mode compiler( Requests^, Tokens?, Code^ ).

compiler( Requests, Tokens, Code ) <-
        parser( Requests1, Tokens, ParseTree ) //
        code_generator( Requests2, ParseTree, Code ) //
        append( Requests1, Requests2, Requests ).
```

Explicit request streams can thus sometimes be very useful. Yet using such streams for all i/o is clumsy. The PPS therefore introduces another meta relation, io_request, which returns an i/o stream to which i/o requests can subsequently be appended. This enables a program to use stream-based i/o when it needs to. For example, the compiler program given here could be called in the context of another program as follows:

```
. . ., io_request( Stream ), compiler( Stream, Tokens, Code ), . . .
```

The io_request relation creates a sub-window within the query interface window, to which i/o requests placed on the stream are sent. Request streams created using io_request can thus be used concurrently with i/o meta relations such as output_term. Request streams might typically be used for normal program output, whilst meta relations are used for diagnostic output, a combination which can be compared with Unix's standard output and standard error streams.

6.2 Data Base Update

As has been seen, PPS data bases can be modified by means of update messages. A suitable abstraction is required to allow user programs to generate these message in meaningful ways. The simplest approach (and that currently implemented in the PPS) is to support a meta relation that asserts a new definition for a PARLOG relation:

```
new_definition( Database, Relation, Definition )
```

When called in a program, this is trapped by the PPS (using the PARLOG meta call's exception handling mechanism) and translated into an update message to the appropriate data base. The use of this meta relation can be illustrated with an example: a program that copies a relation from one data base to another:

```
copy( Relation, FromDatabase, ToDatabase ) <-
        definition( FromDatabase, Relation, Definition ) ,
        new_definition( ToDatabase, Relation, Definition ).
```

where definition is the meta relation that retrieves the definition of Relation in FromDatabase and new_definition is the updating meta relation. Such a meta relation can also be used to implement more sophisticated editors and program transformation tools.

Asserting a new version of a relation in a data base extends the relation history, which is maintained as a list of (time, definition) pairs, where time is a timestamp. Subsequent queries and meta queries will access the new definition. Previous versions can still be accessed using meta relations such as:

```
definition( Database, Relation, Time, Definition )
```

which retrieves the definition of Relation which applies at Time. This meta relation can be used to write a meta program that restores an earlier version of a relation:

```
restore( Database, Relation, Time ) <-
        definition ( Database, Relation, Time, Definition ) &
```

```
new_definition( Database, Relation, Definition ).
```

Programs that execute queries with respect to earlier versions of a data base or generate new data bases containing previous states are also easy to write.

6.3 Persistent data bases

Secondary storage devices allow users to store data for long periods of time. They normally support file systems, which provide structure for user data and allow users to experiment with changes to programs, maintain alternative versions and restore previous versions. They may also serve as a mechanism for transporting data, both between applications and between physical systems. The PPS provides a storage abstraction that subsumes many of these functions. This is the data base, which is implemented as a persistent history. Data bases provide for the persistence of user data, provide a structuring mechanism and provide backups of old versions. They also provide a common format for all data in the PPS. The term *persistence* indicates that the user can refer to data without concern for its location or longevity [Atkinson et al, 1983].

Section 3.3 described how a PPS data base is implemented using a perpetual process that maintains the definitions of the relations in the data base as a PARLOG term. Meta queries are evaluated with respect to this term, as are updates which if successful generate a new term representing a modified data base. If the PPS were implemented on hardware that guaranteed the persistence of PARLOG computation, this representation of data bases would be sufficient. In general, however, the PPS must access secondary storage to load data bases and record modifications.

The program db_ms in section 3.3 showed data base source and object code being loaded in its entirety by queries to the disk server. In practice, the PPS may contain many data bases, many of which may not referenced in any one session. Furthermore, the source history in particular may be large and seldom referenced. The overhead, both in memory and processing time, of always transferring all data bases into memory is unacceptably high. Mechanisms to permit the incremental loading of object and source code are therefore introduced.

6.3.1 Virtualising Object Code

The object code for a data base is only loaded when the data base is queried. Though a suitable exception handler could load the object code for individual relations as required, the object code for the entire data base is currently loaded as a unit for greater efficiency. The program db_ms that spawns the data base managers must thus be rewritten to spawn *data base initiators*:

```
mode db_ms( DbList?, Disk^, SwitchList^ ).

db_ms( [ db( Db, SourceId, ObjectId ) | DbList ], Disk, [ sw( Db, ReqStream ) | RestofSwitchList ] ) <-
        db_initiator( ReqStream, Db, SourceId, ObjectId, Disk1 ) //
        merge( Disk1, Disk2, Disk) //                    % merge Disk streams.
        db_ms( DbList, Disk2, RestofSwitchList ) ..
db_ms( [ ], [ ], [ ] ).
```

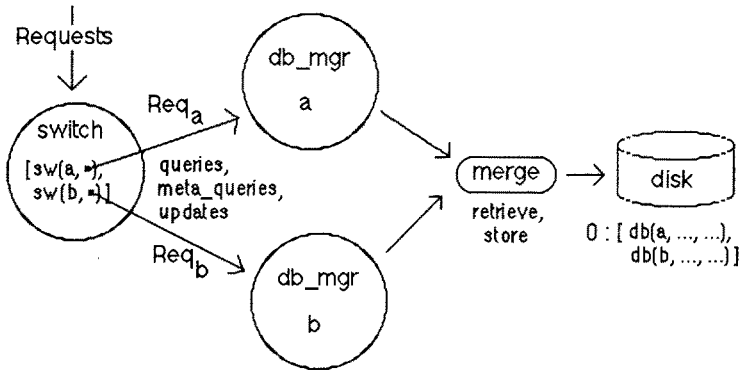
The data base initiator simply waits for an access to the data base and then loads the data base's object code before initiating the database manager:

```
mode db_initiator( InStream?, Db?, SourceId?, ObjectId?, Disk^ ).

db_initiator( InStream, Db, SourceId, ObjectId, [ retrieve( ObjectId, Object ) | Disk ] ) <-
        data( InStream ) :
        load( Object, Module ) &
        db_mgr( InStream, Db, SourceId, Module, [ ], Disk ).
```

Note that the data base manager has an extra argument which is a disk request stream and that it now carries the source index number as its source argument, instead of retrieved source. The process network

illustrated in section 3.3 is thus as follows:



6.3.2 Persistent Binary Trees

The source code, which is generally much larger and less frequently referenced than the object code, is maintained as a *persistent binary tree* [Foster, 1986c]. This is a binary tree ordered on relation name (for rapid access to a particular relation) in which each node contains disk identifiers for a relation history and two offspring nodes. The initial Source argument of the data base manager is the disk identifier of the first node in this tree. This and other nodes are accessed when the relations process_meta_query or process_update (which process meta queries and updates respectively) use the relation pb_dereference to dereference links in the persisent binary tree. This recognises disk identifiers (represented as integers) and generates retrieve messages to the disk server to retrieve nodes. In-memory sections of the tree are represented as PARLOG terms instead of disk identifiers. Repeated retrievals lead to the tree being gradually copied into memory.

```
mode pb_dereference( Node?, Term^, DiskOut^, DiskIn? ).

pb_dereference( Node, Term , [ retrieve( Node, Term ) | Disk], Disk ) <-
        integer( Node ) : true ;
pb_dereference( Node, Node, Disk, Disk ).
```

As pb_dereference can copy nodes into memory, both process_meta_query and process_update now generate a new Source term representing a possibly modified in-memory component of the tree. The data base manager is therefore as follows:

```
mode db_mgr( InStream?, Db?, Source?, Module?, DoneVars?, Disk^ ).

db_mgr( [message(query(Query, Done), Result) | InS], Db, Source, Module, DoneVars, Disk) <-
        Result = eval( Module, Query ) //                % // : concurrent evaluation of queries
        db_mgr (InS, Db, Source, Module, [ Done | DoneVars ], Disk ) ..
db_mgr( [ message(meta_query(MetaQuery), Result) | InStream], Db, Source, Module,
                                                                DoneVars, Disk) <-
        process_meta_query( MetaQuery, Result, Source, Disk1, NewDone, NewSource ) //
        merge(Disk1, Disk2, Disk) //                      % //, merge : concurrent meta queries
        db_mgr( InStream, Db, NewSource, Module, [ NewDone | DoneVars ], Disk2 ) ..
db_mgr( [ message(update(Update), Result ) | InStream ], Db, Srce, Module, DoneVars, Disk) <-
        process_update( Update, Result, Db, Srce, Module, DoneVars, Disk, NewSrce, NewDisk) &
        db_mgr( InStream, Db, NewSrce, Module, DoneVars, NewDisk).
```

6.3.3 Releasing Memory and Recording Updates

The previous sections showed how the PPS implements a form of virtual memory by copying source and object code into memory only as required. For these techniques to be useful, mechanisms are required to free memory when it is no longer needed. These mechanisms must ensure that modifications to source and object code are recorded on disk.

The PPS records modifications to object code on disk as they are performed. The memory occupied by object code modules can therefore be freed without further processing (using the primitive unload), with the proviso that data base managers must ensure that object code is not deleted whilst in use. A change to source, on the other hand, is recorded as modifications to the in-memory component of a persistent binary tree. A mechanism to free memory used to hold source must copy modified branches of the tree back to disk before allowing the source term to be garbage collected. The implementation of this mechanism in PARLOG is not difficult if in-memory persistent binary tree nodes are represented as 5-tuples:

```
node( RelationName, Value, LeftTree, RightTree, Modified )
```

where Modified is either the disk identifier of the node, if the node has not been modified, or true if it has. A recursive commit can then be defined, which writes modified nodes of a sub-tree with Root back to disk (by generating store messages to the disk server) and returns the disk identifier of the root of a new tree:

```
mode commit( Root?, DiskId^, Disk^, NewDisk? ).

commit( node( Name, Value, Left, Right, true ), DiskId, Disk, NewDisk ) <-
      commit( Left, CLeft, Disk, Disk1) ,
      commit( Right, CRight, Disk1, Disk2) ,
      Disk2 = [ store( node( Name, Value, CLeft, CRight ), DiskId ) | NewDisk ] ;
commit( node( Name, Value, Left, Right, DiskId ), DiskId, Disk, Disk ).
```

unload and commit thus provide the means to release memory. These mechanisms can be activated by special 'release' messages sent to the data base manager. A data base manager clause to process a release_source message is as follows:

```
db_mgr( [ message( release_source, Result) | InStream ], Name, Source, Disk ) <-
      commit( Source, NewSource, Disk, NewDisk ) ,
      db_mgr( InStream, Name, NewSource, NewDisk ) ..
```

Release messages can be generated by a central memory server, either periodically or when memory runs short. The first approach can be compared with the Unix 'sync' command, which is called periodically to flush buffered i/o to disk. The implementation of the second approach can exploit the resource control provided by the PARLOG meta call. All PPS entities can be run inside a meta call and allocated limited memory resources. If these resources are consumed, a **resources** exception message is generated. This is effectively a request for more resources. When the central server receives such a message it can generate release messages to selected data bases. The memory freed in this way can then be allocated to the entity that has exhausted its resources.

6.4 Other Resources

Previous sections have described how data bases are implemented in the PPS using data base managers that process query, meta_query and update messages relative to data bases maintained as both source and object code. Other resources can be modelled as managers which process the same messages in different ways. For example, a PPS entity termed the *name server* processes queries of the form unique_name(X) by unifying the argument X with a name that is unique in the sense that every query to the name server is unified with a different value. The *editor server* processes queries of the form run_editor(Definition, NewDefinition) by creating an edit window containing Definition and unifying NewDefinition with a structure representing the contents of the window when the user finishes editing. The editor server can be understood as solving the relation run_editor by querying the user. Its use can be illustrated with an example: a program that edits Relation in Database:

```
edit( Relation, Database ) <-
        definition( Database, Relation, Definition) ,
        run_editor( Definition, NewDefinition ) ,
        new_definition( Database, Relation, NewDefinition ).
```

This program retrieves the definition of Relation in Database using the definition meta relation, uses the relation run_editor to query the user for a new definition and then stores the new version of Relation using the new_definition meta relation described in section 6.2.

Another form of PPS resource is the *term data base*. These are data bases that are maintained purely as PARLOG terms representing PARLOG programs. Queries are evaluated with respect to these terms by a process of interpretation. Interpretation is slower than execution, but update of source is faster than recompilation. The PPS history data base and a number of other seldom-accessed but volatile data bases representing system state are therefore implemented in this way. The history data base records its contents as a list of tuples:

```
query( Identifier, Database, Query, Variables, State )
```

(representing a Query assigned Identifier executed in Database, generating variable bindings Variables and currently in State) and processes queries by performing a lookup on this list. Similar techniques can be used to incorporate Prolog (or other languages) into the PPS.

7. Comparison with Related Work

The PPS is a logic programming environment implemented in a logic programming language. It is characterised by its declarative architecture in which user and system progams and system state and structure are represented as logic clauses. These clauses are both executable as programs and accessible as data. This dual nature of PPS entities facilitates tool construction and allows program transformations to be described in logic. At a lower level, implementation of the PPS is characterised by a message-passing architecture which provides fundamental support for multiprocessing and conflict resolution.

Programming environments for Lisp [Sandewall, 1978] and Prolog (notably micro-Prolog [McCabe and Clark, 1980], with its Lisp-like internal syntax) commonly provide primitives that allow both programs to be accessed as data and data to be treated as program. The primitives that construct or modify programs at run-time are typically non-logical (but see [Bowen, 1986]). Mechanisms to prevent concurrent update and execution of programs are not generally provided.

Syntax-based editors allow users of certain procedural languags to interact with programs as data structures [Teitelbaum and Reps, 1981]. The identity of source and executable program is commonly lost in these systems, however. SmallTalk browsers [Goldberg, 1983] and MacProlog windows [French, 1985] maintain this identity. MacProlog, for example, automatically recompiles the contents of a window that has been edited prior to running programs in it.

Because the PPS represents both programs and their structure as logic clauses, it is easy to represent and retrieve information about programs. Programming environments for more procedural languages achieve this functionality by a data base approach [Linton, 1984], [Ramamoorthy, 1985]. Information about programs and their specifications is stored in a data base and can be reasoned about. Certain Lisp systems adopt a similar approach [Teitelman and Masinter, 1981], [CLP Project, 1986]. The user reasons about a data base rather than about the actual programs, however. This can complicate updates, as the data base and the entities it describes must be kept consistent.

The implementation of the PPS can be understood in terms of a message-passing process model that can be compared with Actor and object-oriented systems [Hewitt et al, 1973], [Goldberg, 1983]. The logic variable and the back communication that this allows leads to somewhat different programming techniques, however. A programming environment that can be compared with the PPS in many respects

is Logix [Silvermann et al, 1986], implemented in the concurrent logic language FCP [Mierowsky et al, 1985]. Logix provides a Unix-like user environment but does not appear to be structured as declaratively as the PPS. The use of concurrent logic languages for systems programming has also been considered by Kusalik [Kusalik, 1986].

The construction of operating systems and programming environments as entities connected by streams is not new: the message-passing paradigm is well-known and this use of streams has been proposed for functional operating systems [Henderson, 1982], [Jones, 1984]. Problems with stream 'spaghetti' has motivated proposals very similar to the PPS switch for functional languages [Stoye, 1986].

The meta programming facilities of the PPS have parallels in other logic programming systems. For example, Multilog [Kauffmann and Grumbach, 1986] incorporates *worlds* and allows users to define inheritance relationships between worlds. These relationships are not defined in logic, however. Mandala [Furukawa et al, 1984] and Prolog/KR [Nakashima, 1984] incorporate similiar concepts. Bowen proposes a meta-level extension to Prolog that enables data bases to be manipulated as first class objects in a Prolog-like language [Bowen, 1986]. PPS data bases, though akin to Bowen's theories (in that they are collections of definitions manipulable by a meta program) are named entities rather than language objects. PPS data bases can also be compared with the entities in Kowalski's logic-based open system [Kowalski, 1986]. This models complex systems as collections of knowledge bases that communicate their beliefs by assertions. Entities receiving assertions may modify their beliefs. The PPS supports knowledge bases which can assimilate knowledge, but requires that computation occur in query managers. In object-oriented programming terms, these can be viewed as *instances* of the entities described by data bases. The PPS is thus a somewhat more sophisticated (though less general) structure.

8. Conclusions and Further Work

This paper has combined a description of essential features of a logic progrmaming environment with a presentation of the use of the concurrent logic language PARLOG for systems programming. This combination was possible because the declarative nature of PARLOG allows PARLOG programs to be read as specifications.

PARLOG is well-suited to the implementation of programming environments. Because it is a high-level, declarative language, complex systems can be specified succinctly and elegantly. Its concurrent constructs and process interpretation allow it to describe concurrent activity. Its control meta call allows it to control computation effectively.

The logic programming environment described is the PARLOG Programming System, which is characterised by its declarative architecture in which user and system progams and system state and structure are represented as logic clauses. These clauses are both executable as programs and accessible as data. This facilitates tool construction and allows program transformations to be described in logic. At a lower level, implementation of the PPS is characterised by a message-passing architecture which provides fundamental support for multiprocessing and conflict resolution.

The PPS is currently implemented on the PARLOG system implemented at Imperial College [Foster et al, 1986]. This is an emulator for an abstract machine, the Sequential PARLOG Machine [Gregory et al, 1986] and runs on Unix machines. A version for Sun workstations incorporates an interface to the Sun window system, which is exploited by the PPS. Certain limitations of this implementation have restricted the PPS's development. Firstly, object code cannot be represented as PARLOG terms. This complicates the implementation of persistent databases, as separate mechanisms must be used to manipulate object and source code. Secondly, the resource-bounded deduction described by the five argument meta call is not supported. This prevents the PPS from tackling interesting resource allocation problems. Both these limitations should be resolved in a new PARLOG implementation currently being designed. As the target machine for this implementation is a multiprocessor, the PPS will become a multiprocessor programming environment without further modification.

Further work aims to extend the PPS to multi-user operation. This will require the introduction of

additional system structures termed *contexts* that define both name spaces (mapping user names for data bases to global names) and access permissions. More complex mechanisms to prevent contention may also be required. Another extension to the PPS aims to allow properties (such as 'created by', 'description', 'analysis data') to be associated with relations. Relations are currently stored in the PPS as simple (timestamp, definition) pairs. The ability to associate arbitrary properties with relations will facilitate the implementation of 'knowledge-based' tools. It is also hoped to develop the user interface to the PPS by introducing browsers and other interactive tools. The range of meta clauses supported by the PPS will also be extended and efficient implementation of these clauses using compilation and transformation techniques investigated. The PPS will also be extended to support Prolog data bases. This will allow investigation of techniques for combining PARLOG and other logic languages.

Acknowledgements

This work was supported by the Science and Engineering Research Council. The authors are grateful to current and visiting members of the PARLOG research group at Imperial College, Alistair Burt, Andrew Davison, Steve Gregory, Tony Kusalik, Melissa Lam and Graem Ringwood for their contributions to PARLOG implementation, for valuable discussions on a variety of topics and generally for providing an agreeable research environment. The authors are also grateful to Martyn Cutcher from ICL, who has shown a continued interest in the research.

References

Atkinson, M.P., Bailey, P.J., Chisholm, K.J., Cockshott, W.P. and Morrison, R. (1982), "An Approach to Persistent Programming", *The Computer Journal*, 26(4), pp 360-365.

Bowen, K.A. (1986), "Meta-level Programming and Knowledge Representation". In *New Generation Computing*, 3, pp 359-383.

CLP Project (1986), "Introduction to the CLF Environment", USC Information Sciences Institute.

Clark, K.L., and Gregory, S. (1984a), "PARLOG: Parallel Programming in Logic", Research Report DoC 84/4, Department of Computing, Imperial College, London, and in *ACM Trans. on P rogramming Languages and Systems*, 8 (1), pp. 1-49, 1986.

Clark, K.L., and Gregory, S. (1984b), "Notes on Systems Programming in PARLOG". In *Proc. Intl. Conf. on 5th Generation Computer Systems* (Tokyo, November 1984), ed Aiso. H., Elsevier North Holland, pp. 299-306.

Clocksin, W.F. and Mellish, C.S. (1981). *Programming in Prolog*, Springer-Verlag, New York.

van Emden, M.H., and de Lucena, G.J. (1982), "Predicate Logic as a Language for Parallel Processing". In *Logic Programming*, eds Clark, K.L and Tarnlund, S-A., Academic Press, London, pp 189-198.

Foster, I.T., Gregory, S., Ringwood, G. A., and Satoh, K. (1986). "A Sequential Implementation of PARLOG". In *Proc. of the 3rd Intl. Logic Programming Conf.*, London, 1986.

Foster, I.T. (1986a). "The PARLOG Programming System: Reference Manual". Dept of Computing, Imperial College, London.

Foster, I.T. (1986b), "Logic Operating Systems: Design Issues", Research Report, Dept. of Computing, Imperial College, London. Submitted for publication.

Foster, I.T. (1986c), "Persistent Parallel Logic", Research Report (in preparation), Dept. of Computing, Imperial College, London.

Foster, I.T. and Kusalik, A.J. (1986), "A Logical Treatment of Secondary Storage". In *Proc. Symp. on Logic Programming*, Salt Lake City, pp 58-69.

French, P. (1985), *MacProlog User Guide*, Logic Programming Associates Ltd.

Furukawa, K., Takeuchi, A., Kunifuji, S., Yasukawa, H., Ohki, M., and Ueda, K. (1984), "Mandala, a Logic Based Knowledge Programming System". In *Proc. Intl. Conf. on 5th Generation Computer Systems* (Tokyo, November 1984), ed Aiso. H., Elsevier North Holland, pp 613-622.

Gregory, S., Foster, I.T., Ringwood, G.A. and Burt, A.D. (1986), "The Sequential PARLOG Machine", Research Report (in preparation), Dept. of Computing, Imperial College, London.

Goldberg, A. (1983), *Smalltalk-80: The Language and its Implementation*, Addison Wesley.

Henderson, P. (1982), "Purely Functional Operating Systems". In *Functional Programming and its Applications*, Eds Darlington, Henderson and Turner, CUP.

Hewitt, C. et al (1973), "A Universal Modular Actor Mechanism for Artificial Intelligence". In *Proc. I JCAI*, 1973.

Jones, S.B. (1984), "A Range of Operating Systems Written in a Purely Functional Style", Technical Monograph PRG-42, Oxford University Computing Laboratory, 1984.

Joseph, M., Prasad, V.R., Narayana, K.T., Ramakrishna, I.V. and Desai, S. (1978), "Language and Structure in an Operating System". In *Proc. 2nd Intl Conf. on Operating System Theory and Practice.*

Kauffmann, H., and Grumbach, A. (1986), "MULTILOG: Multiple Worlds in Logic". In *Proc. 7th ECAI*, Brighton, UK, pp 291-305.

Kowalski, R.A. (1979), *Logic for Problem Solving*, North Holland.

Kowalski, R.A. (1986), "Logic-Based Open Systems", Research Report, Dept. of Computing, Imperial College, London.

Kusalik, A.J. (1986), "Specification and Initialisation of a Logic Computer System", in *New Generation Computing*, 4, pp. 189-209, 1986.

Linton, M. (1984), "Implementing Relational Views of Programs", *Proc ACM SIGSOFT/SIGPLAN Software Engineering Symp. Practical Software Development Environments.*

McCabe, F.G. and Clark, K.L., (1980), *Micro-Prolog 3.0 Programmer's Reference Manual*, Logic Programming Associates Ltd.

Mierowsky, C., Taylor, S., Shapiro, E., Levy, J., and Safra, M. (1985), "The Design and Implementation of Flat Concurrent Prolog", Technical Report CS85-09, Weizmann Institute, Rehovot.

Nakashima, H. (1984), "Knowledge Representation in Prolog/KR". In *Proc. Intl. Symp. on Logic P rogramming*, IEEE Computer Society, pp 126-130.

Ramamoorthy, C.V. et al (1985), "GENESIS: An Interactive Environment for Development and Evolution of Software". In *Proc. COMPSAC*, 1985.

Sandewall, E. (1978), "Programming in an Interactive Environment: The Lisp Experience". In *Computing Surveys*, 10(1), pp 35-71.

Sergot, M.J. (1982), "A Query-the-user Facility for Logic Programming". DoC Report 82/18, Dept. of Computing, Imperial College. In *New Horizons in Educational Computing*, Yazdani, M., ed, Ellis Horwood.

Shapiro, E.Y. (1984), "Systems Programming in Concurrent Prolog", in *Proc. 11th ACM POPL Symp.*, Salt Lake City, Utah.

Shapiro, E.Y. and Safra, S. (1986), "Multiway Merge with Constant Delay in Concurrent Prolog". In *New Generation Computing*, 4(2), pp 211-216.

Silverman, W., Hirsch, M., Houri, A., and Shapiro, E.Y. (1986), "The Logix User Manual, Version 1.21", Technical Report CS-21, Weizmann Institute, Rehovot.

Stoye, W. (1986), "A New Scheme for Writing Functional Operating Systems", Research Report, University of Cambridge Computing Laboratory, Cambridge, 1986.

Teitelbaum, T., and Reps, T. (1981), "The Cornell Program Synthesiser: A Syntax-Directed Programming Environment". In *CACM*, 24(9).

Teitelman, W. and Masinter, L. (1981), "The Interlisp Programming Environment". In *IEEE Computer* 14(4), pp 25-33.

Wulf, W.A. (1981), *HYDRA/C.mmp: An Experimental Computer System*, McGraw-Hill, N.Y.