I.C. Prolog II: a Multi-threaded Prolog System

Damian Chu

Department of Computing
Imperial College of Science, Technology and Medicine
180 Queen's Gate, London SW7 2BZ
United Kingdom
email: dac@doc.ic.ac.uk

Abstract. This paper introduces IC-Prolog II – a new implementation of Prolog that is particularly suited to distributed applications. Unlike other work on distributed logic programming, we do not aim to improve the raw performance of a logic program nor do we require multi-processor machines or specialised hardware. Instead, our aim is to widen the applicability of logic programming to encompass new classes of practical applications which require the co-ordination of concurrently executing programs on separate workstations to communicate over the network.
IC-Prolog II features multiple threads, a Parlog sub-system and high-level communication primitives. Multiple threads enables the concurrent execution of independent goals. The Parlog sub-system allows local fine-grained parallelism to be specified. In IC-Prolog II, Prolog can call Parlog and vice-versa. The combination of the two logic languages offers greater expressive power than simply the sum of the two since different components of the same application may use either Parlog or Prolog or both. The high-level communication primitives provide the means for independent IC-Prolog II processes on different machines on a network to communciate. The result is a language well-suited for writing network-friendly applications.

Keywords: Multiple Threads, Distributed Applications, Concurrency

1 Introduction

In recent years, interest in distributed computing has risen very rapidly. We are realising that centralised monolithic systems are costly, inefficient and inflexible. Large mainframe computers are now being replaced by networks of workstations. This trend is responsible for the emerging importance of distributed applications, where programs executing on separate machines need to communicate over the network. Moreover, when previously standalone applications are enhanced with the ability of network communication, they can cooperate to solve problems that each would find unsolvable on its own. This is the rationale behind the expanding field of Intelligent Cooperative Information Systems.

Logic programming has been used successfully in many knowledge-based applications. However, very little work has been done to address the issues in implementing distributed applications of the type discussed above. Researchers in logic programming systems have generally concentrated their efforts on trying to exploit the inherent parallelism within a *single* logic program e.g., Andorra [10], Aurora [15], MUSE [1]. These systems generally require dedicated hardware or multi-processor machines with either shared or distributed memory. Although they can offer significant performance gains, they were not designed with distributed applications in mind, which generally involve the *concurrent execution* of two or more programs, usually on different machines.

IC-Prolog II (`ICP` for short) [9] is a new implementation of Prolog developed at Imperial College which attempts to address this need. It contains features such as a multi-threading capability, a Parlog sub-system, high level communication primitives and an object-oriented extension.[1] These features open up new application areas to logic programming such as distributed knowledge-based systems, cooperating expert systems and multi-agent systems.

[1] The object-oriented extension is not discussed in this paper.

2 IC-Prolog II

IC-Prolog II will be one of the results of the IMAGINE project. The IMAGINE project is concerned with building Multi-Agent Systems, in which a collection of semi-autonomous problem-solving agents coordinate and cooperate to either solve joint problems or achieve their own goals. The agents typically reside on separate computers connected by a network.

`ICP` was developed for prototyping Multi-Agent Systems using logic programming. From the initial project specification, it was clear that support for *concurrency* and *communication* was important.

'Concurrency' is the ability to handle multiple problems simultaneously. As a simple example, it is undesirable for a complex query to monopolise a database server while there may be simple queries waiting to be processed. A single thread of control is insufficient and we therefore needed parallelism. Both fine-grain and coarse-grain parallelism are supported in `ICP` through having a Parlog sub-system and implementing multiple Prolog threads.

'Communication' refers to the ability to send and receive messages between threads and across the network, exchanging data with other agents. We have a simple scheme called *pipes* for communication between local threads. Since TCP/IP is the most widely used communication protocol, we implemented a Prolog interface to the TCP/IP protocol suite. This enabled communication across a network and even interfacing with existing software possibly written in other languages, but it was not still powerful enough to support the kind of agent communication we needed which includes features like multi-casting and access control. We therefore added a more sophisticated communication model called *mailboxes* which could provide these additional functionalities.

This paper describes the main features of `ICP`. In Section 3, we introduce the notion of multiple threads and sketch how they are implemented in `ICP`. We show how they can communicate and give a simple example of the benefits they can bring to server-based applications. Section 4 provides an overview of the Parlog sub-system in `ICP` that allows us to express the highly-parallel components of an application. Section 5 takes a more detailed look at the communication facilities offered by `ICP` since these are at the heart of a distributed application. In Section 6, we give a full example of a distributed application showing how we might program a simple airline reservations system using `ICP`. We point out related work in Section 7 and finally give our conclusions in Section 8.

3 Multiple Threads

Traditional Prolog systems have only one thread of control and concurrent execution is therefore not possible. A number of Prolog systems such as Prolog II [8], NU-Prolog [22] and SICStus Prolog [3] get around the single thread restriction by implementing some form of co-routining whereby calls can be delayed until specified arguments are instantiated. However, this data-driven approach is not appropriate for the cases where no variables are shared between the co-routines. This is a common case in client-server applications where client transactions to be executed on the server are totally independent of each other.

A more general solution is to allow multiple threads. Each thread is a distinct WAM-like [23] Prolog engine. Having multiple threads allows independent programs to run concurrently in pseudo-parallel. Pseudo-parallel execution was a feature of both the original IC-PROLOG in 1979 [7] and Epilog [19]. We differ from both those systems since our threads do not share the same data area. We considered using Unix process forking to implement multiple threads, however this would have been very expensive in terms of performance. In our implementation, all the threads execute within a single Unix process so we needed to do our own scheduling. Each thread contains its own stack area, its own set of WAM registers and some housekeeping information. The structure of a thread is shown in Figure 1.

The two link pointers are used to chain all the threads together in a double link list. The status flag indicates whether the thread is currently runnable and the two channels record where the current input and output of the thread is. The stack area includes space for the usual WAM stacks i.e., the heap, the evaluation stack and the trail. The code space however is shared by all the threads.

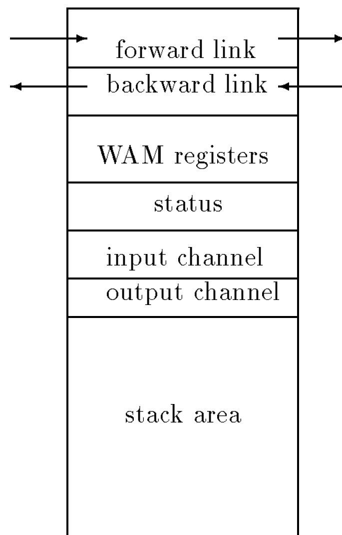

Fig. 1. Structure of a Thread

In ICP, we implemented primitives to fork a new thread, to suspend a thread and to resume a suspended thread. At any time, only one thread is running. A thread may suspend by explicitly calling the `suspend` primitive or more usually, because of the non-availability of input data/messages. When the current thread suspends, the next runnable thread in the chain is resumed and becomes the current thread. To prevent a single thread from monopolising the resources, a time-slice mechanism was implemented. When a time slice occurs (currently every tenth of a second), the current thread remains in the runnable state but temporarily stops executing to allow other runnable threads to execute.

Since each thread has it own stack areas, this implies that the variables of a thread are private. Unification within a thread cannot bind variables in other threads. Communication between threads is achieved through explicit message passing only. The primitives which read messages will suspend if no data is available, thus allowing the data-driven style of co-routining to be programmed.

In ICP, the primitive to create new threads is `fork/1`, the single argument being the goal to be executed in the new thread. Operationally, the `fork/1` primitive always succeeds immediately regardless of the success or failure of the forked goal. From the point of view of the current thread, `fork/1` behaves as if it is the goal `true`.

The query

```
| ?- fork(producer), fork(consumer).
```

creates two new threads which run concurrently. The system will automatically time-share between the two threads. This behaviour is very different from normal Prolog evaluation which does not start the second goal until the first has succeeded.

3.1 Inter-Thread Communication

In our example, the producer must communicate what it produces to the consumer. Communication between threads is via asynchronous message passing, though there are many forms that this can take. For local threads created within the same Unix process, the most direct way is to use *pipes*.[2] A *pipe* is a uni-directional communication channel and is implemented using a memory buffer. It has two ends called *ports*. One end is the output port and the other is the input port. Data written to the output port can be read from the input port. Since ports are accessible by all threads, if we can arrange for the producer to write to the output port, and the consumer to read from the input port, then inter-thread communication is achieved.

[2] These are not the same as Unix pipes.

A pipe is created using the pipe/2 primitive. This returns in its two arguments the identifiers for the output port and the input port. So we could run our example as

```
| ?- pipe(Out,In),            % create a pipe
     fork(producer(Out)),     % producer writes to output port
     fork(consumer(In)).      % consumer reads from input port
```

The producer program would use the write_pipe/2 primitive to send data to the consumer, which would read it using the read_pipe/2 primitive.[3] These message passing primitives are non-backtrackable which implies a commit-on-write semantics. Variables may be sent in messages, but they act only as placeholders since the receiver of the message will create a variable in its own data area. The corresponding variables in the sending and receiving threads are in no way linked, i.e., there is no distributed unification.

3.2 Writing Server Programs

A server program is usually a tail-recursive loop which services one client request per iteration. A typical *iterative* server could be written in ICP as follows :

```
server(In) :-                 % parameter specifies where to read from
     read_pipe(In, Req),      % read next request
     service(Req),            % service the request
     server(In).              % recurse
```

In an iterative server, the requests are serviced sequentially. This can create bottlenecks as some requests may take a long time to service. A *concurrent* server avoids this problem by allowing the requests to be serviced concurrently. This style of server could not be programmed in traditional Prolog systems since it requires multiple threads. In ICP, we can code it like this :

```
server(In) :-                 % parameter specifies where to read from
     read_pipe(In, Req),      % read next request
     fork(service(Req)),      % create a new thread to service the request
     server(In).              % recurse
```

During each iteration of the loop, the server reads one request and forks a program to service that request. Since the server does not need to wait for the forked program to complete, it can go on to read the next request. If there are no more requests, the server suspends until data is available.

4 Parlog

Multiple threads allows a limited form of parallelism. The parallelism is coarse-grained and message passing is explicit. For highly parallel applications, the overhead of creating a large number of short-lived threads is significant. Parlog is a much more suitable language for describing fine-grained parallelism.

Parlog[14] belongs to the family of Committed Choice Non-Deterministic logic languages. Parlog allows two forms of parallelism – stream AND-parallelism and committed OR-parallelism. Stream AND-parallelism is the concurrent evaluation of goals which share variables, with the value being implicitly communicated incrementally between the goals. Goals which do not share variables are trivially subsumed by stream AND-parallelism. Committed OR-parallelism is the concurrent search for candidate clauses to match a goal. When a match is found, the clause *commits* and the other choices are discarded. An important difference between Parlog and Prolog in that because of the restriction of Committed Choice, Parlog programs do not backtrack.

Prolog and Parlog are complementary languages. Parlog's fine grain parallelism appeals to highly parallel applications which cannot be coded in standard Prolog, or could not be implemented

[3] Messages can also be read using the look_pipe/2 primitive which does not remove the message from the communication channel.

efficiently using the much coarser ICP threads. On the other hand, Prolog's backtracking search capabilities to find all solutions cannot be emulated by a Parlog program. To make the best use of both languages, Parlog is included in ICP as a separate thread. Communication between Parlog and Prolog is done in the same way as other inter-thread communications i.e., via message passing. Prolog can call Parlog by using the `parlog/1` primitive. This passes a message to the Parlog thread to execute a goal. If solutions are required by Prolog, they can be explicitly communicated back using the pipe mechanism discussed previously. Similarly, Parlog can call Prolog using the `prolog/1` primitive. There are variations of this primitive to get single solution or get all solutions from Prolog and to control whether they should be generated eagerly or lazily.

ICP provides a very loose coupling between Prolog and Parlog. We have not attempted to combine them into a single language. In fact, the Parlog sub-system in ICP is implemented by making minor changes to the original standalone Parallel Parlog system [11] to convert it into a thread. Within ICP, we are thus able to make use of Parlog's Don't Care Non-Determinism and Prolog's Don't Know Non-Determinism within the same application. There have been other proposals for combining these two types of non-determinisms. Clark and Gregory's scheme [6] offers a very powerful hybrid language but requires substantial changes to the unification algorithm, thus severely affecting performance. Naish's PNU-Prolog [17] is essentially a preprocessor for NU-Prolog. It uses coroutines and therefore cannot express applications which require true time-sharing between goals.

5 Communication Primitives

For distributed applications, communication facilities are very important. Threads can communicate using pipes if they are running in the same ICP process. However, communication between ICP processes (possibly on different machines) or between ICP and other processes must be done in a different way.

The most widely-used protocol on computer networks is the TCP/IP protocol suite. By defining an interface from ICP to the TCP/IP system calls, we can use ICP primitives to communicate across the world-wide network.

5.1 TCP/IP Primitives

Communication protocols can be divided into connectionless and connection-oriented protocols. In connectionless protocols, each message is sent individually and therefore must include the destination address. There is also no guarantee that a sequence of messages sent to the same address will arrive in the order in which they were sent. In connection-oriented protocols, a link must first be established between the two parties. Thereafter, no destination address need be specified in any message since it is implicit in the link. Furthermore, messages sent from one party to the other is guaranteed to be received in the same order that they were sent.

TCP/IP provides both connectionless and connection-oriented protocols. Both styles of protocol may be used in ICP, though connection-oriented protocol is preferred since it is order preserving and is more reliable. Connectionless protocol is used when interfacing to existing software which use this protocol or when writing applications which require multiplexing.

Below we will give a flavour of how TCP/IP primitives are used in ICP. There are many finer details to network communication and users of ICP should consult the manual [9] along with books on network programming [21].

Connectionless Communication

First we will consider connectionless communication since it is simpler. A network address consists of two parts - a machine number and a port number. The channel of communication is called a *socket* which is analogous to a stream identifier for file I/O. For two programs to communicate, we first create a socket and assign a port number to it. We do this using the following primitive

```
tcp_connectionless(+Port, -Socket)
```

We use the prefix notation '+' to denote an input argument and '-' to denote an output argument. A machine number was not needed because it defaults to the machine on which we are running.

The other program running on a different machine executes the same primitive. When both sides have created a socket, we can send messages using

```
tcp_sendto(+Socket, +Message, +Port, +Machine)
```

specifying the message to be sent and the destination network address. To receive messages, we use the primitive

```
tcp_recvfrom(+Socket, -Message, -Port, -Machine)
```

This will return the message and the sender's network address. The primitive will suspend until a message arrives.[4] Finally, we close the socket using

```
tcp_close(+Socket)
```

Figure 2 shows the primitive calls used in a typical connectionless communication.[5] There are many options which can be set such as specifying a timeout value for a primitive so that it fails after a specified time instead of being suspended forever. However, the basic primitives described above are sufficient to program simple communications.

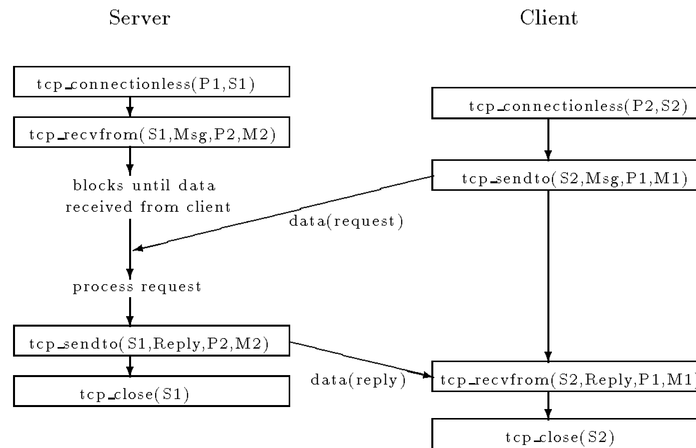

Fig. 2. TCP/IP Primitive Calls in Connectionless Communication

Connection-Oriented Communication

TCP is based on the client-server model of communication. In this model a server program is started first which after initialising, waits for clients to connect to it. When a client program contacts the server to establish a link, it wakes up to allow communication to take place. When the session is over, the server goes back to waiting for the next client.

To code this in `ICP`, the server program creates a socket and waits for connections using `tcp_accept/2` as follow :

```
tcp_server(+Port, -Socket),
tcp_accept(+Socket, -NewSocket)
```

[4] We can optionally specify a timeout value after which the suspended primitive fails if no message is received within that time.

[5] adapted from a figure in [21]

The call to tcp_accept/2 will suspend until a connection request from a client is received whereupon a new socket will be created specifically for that connection.

On the client side, we create a socket and initiate a connection at the same time using

```
tcp_client(+Port, +Machine, -Socket)
```

specifying the address of the server program to connect to. This primitive suspends until the connection is accepted.

Once the connection is set up, messages can be sent using

```
tcp_send(+Socket, +Message)
```

Note that the server needs to use the *NewSocket* number returned by tcp_accept/2 instead of the original socket which is used exclusively for new connection requests. To receive messages, we use

```
tcp_recv(+Socket, -Message)
```

and at the end we use tcp_close/1 as before.

Again, there are many more options and variations to the basic primitives available. Those described above are the most important ones.

Figure 3 shows the primitive calls used in a typical connection communication.[6]

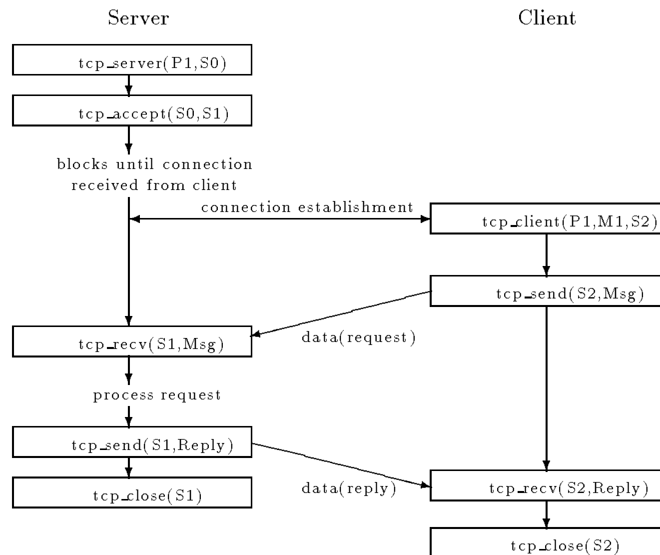

Fig. 3. TCP/IP Primitive Calls in Connection Communication

With multiple threads in ICP, it is possible to fork a thread to handle each connection and thus be free to accept new connections concurrently. Here is the concurrent server example again, this time using TCP instead of pipes.

```
concurrent_server(Port) :-
   tcp_server(Port, Socket),    % where to listen for connections
   multi_serve(Socket).

multi_serve(Socket) :-
   tcp_accept(Socket, New),     % got a new connection
   fork(service(New)),          % create new thread to service it
   multi_serve(Socket).         % look for more connections
```

[6] adapted from a figure in [21]

Communication with Foreign Processes

The TCP/IP protocol is widely used by many applications written in other programming languages such as C. By having the `ICP` interface to TCP/IP, it is possible to write Prolog code which communicate directly with these external programs/packages. For example, we have written a simple `ICP` program which sends email by interfacing with the standard Unix mail daemon.

5.2 Mailbox Primitives

In TCP, every communication channel set up between a server and a client is a separate network connection. The system overheads of creating sockets and setting up connections over the network become significant if the conversations are short and there are many such conversations. For example, if a thread on one machine is interacting with ten threads on another machine, there will be ten network connections between the two machines. Furthermore, TCP primitives are resticted to one-to-one communication.

An alternative communication model is the mailbox model proposed by V. Benjumea. In the mailbox model, the instrument of communication is a *mailbox*. A mailbox is simply a repository for messages. Mailboxes may be created freely by any thread since they are very cheap to implement. There is only one network connection between each pair of machines regardless of the number of mailboxes created. Messages can be sent to and removed from a mailbox. Instead of having to create two sockets to communicate, we now need to create only one mailbox. For two threads to communicate, all they need is to share the mailbox identifier. The sender places a message in the mailbox, and the receiver removes it. A mailbox can store multiple messages. Messages are kept in arrival order so that it naturally simulates connection-oriented communication.

In `ICP`, a mailbox is created using

```
mbx_create(-Id)
```

This returns an identifier naming the newly created mailbox. Mailbox identifiers are globally unique in the network, so the exact same identifier may be used by any thread from any machine. To send and receive messages to/from mailboxes, we use

```
mbx_send(+Id, +Message)
mbx_recv(+Id, -Message)
```

A name may be associated with a mailbox identifier using

```
mbx_bind(+Id, +Name)
```

This *registers* the name with an external name-server program so that other threads or `ICP` processes may obtain the identifier by querying the name-server using the primitive

```
mbx_getid(+Name, -Id)
```

A link may be established between two mailboxes. When a message is sent to the first mailbox in a link, the message is automatically forwarded to the second mailbox. Note that the link is uni-directional only – messages sent directly to the second mailbox will not by forwarded to the first. The second mailbox remains an ordinary mailbox while the first becomes a *linked* mailbox. Linked mailboxes do not store any messages. Mailboxes are linked together by specifying the two mailbox identifiers in the primitive :

```
mbx_link(+From, +To)
```

A mailbox may be linked to multiple mailboxes simultaneously. In this case, messages sent to the linked mailbox will cause a copy of the message to be forwarded to each and every link. This is how we can configure one-to-many communication. Note that there is a distinction between the case where multiple receivers each receive copies of all messages (i.e., multi-casting), and the case where one and only one of the multiple receivers receive each message. In the former case, we can use linked mailboxes but in the latter case, one mailbox is sufficient.

Many-to-one communication can also be configured by linking multiple mailboxes to the same mailbox. Indeed, any arbitrary communication topology may be built up using links.

Finally, to destroy a mailbox, we use the primitive

```
mbx_close(+Id)
```

There are many other advanced features of mailboxes such as permission control, timeouts, polling and non-destructive reads which are described in the `ICP` manual [9].

6 An Airline Reservations Example

To illustrate use of the language, we will develop a small airline reservations system as an example. This will make use of multiple threads, TCP/IP communication, mailboxes and Prolog's dynamic database.

To simplify the example, we will not concern ourselves with dates and times of flights, or the different types of fares. We will assume that flight information is stored as dynamic clauses using the relation `db_seats/2`. For example,

```
db_seats(ia101,23).
```

represents the fact that there are 23 seats remaining on Imperial Airlines flight number ia101.

The airline reservations system must be able to service multiple sessions concurrently since travel agents all over the world may be trying to book flights at the same time. To manage these concurrent connections, we need a session manager program. The session manager program creates a socket on a publicised port number[7], and waits for connections to that port from travel agents. It forks a new thread to handle each session. This is coded as in Figure 4. For the purposes of this example, we will assume that the Imperial Airlines reservations system uses port number 6789 and runs on a computer with Internet address 146.169.21.1.

```
session_manager :-
   tcp_server(6789, Socket),   % publicised port number
   session_manager(Socket).

session_manager(S) :-
   tcp_accept(S, NewS),        % got a new connection
   fork(session(NewS)),        % fork new thread to handle it
   session_manager(S).         % listen for more ...
```

Fig. 4. Session Manager Program

During each session, requests may be made to find out availability information or try to reserve a number of seats. The system will reply with the number of available seats in the case of a query, or the result `ok` or `failed` in the case of a reservation. To avoid the lost update problem, a flight number must be locked before a reservation can be made, and unlocked after updating the number of remaining seats. If the flight number is already locked, availability queries can still be answered but reservations will be suspended until the flight number is unlocked. This is expressed in Figure 5.

Locking and unlocking flight numbers is done by sending messages to a lock manager program. We could have used pipes, TCP/IP or mailboxes to communicate with the lock manager. We have chosen to use mailboxes on this occasion since this is the most efficient method for creating short-lived communication channels. To lock a flight number, a new mailbox is created for the reply from the lock manager. The identifier of this new mailbox is sent to the lock manager along with the flight number to be locked. Unlocking is much simpler since it does not require a reply. In both cases, we need to first obtain the mailbox identifier of the lock manager. The code for these two operations is shown in Figure 6.

The lock manager maintains a list of flight numbers which are locked i.e., currently being updated. The list consists of entries of the form

[7] analogous to the globally known telephone number for an airline's reservations system

```
session(Id) :-                    % argument is socket no.
   tcp_recv(Id, Request),         % received a new request
   service(Request, Id).

service(quit, Id) :- !,           % terminate session
   tcp_close(Id).
service(info(Flight), Id) :- !,   % availability query
   ( db_seats(Flight, Num) ->
       tcp_send(Id, Num)          % reply with no. of seats
   ; tcp_send(Id, error('no such flight'))
   ),
   session(Id).
% the next clause is for making reservations
service(reserve(Flight, SeatsReqd), Id) :- !,
   lock_flight(Flight),           % this may suspend
   reserve(Flight, SeatsReqd, Reply),
   unlock_flight(Flight),
   tcp_send(Id, Reply),           % send back result of reservation
   session(Id).
service(Other, Id) :-             % errors
   tcp_send(Id, error('invalid request')),
   session(Id).

reserve(Flight, SeatsReqd, ok) :-
   db_seats(Flight, Num),                 % valid flight ?
   SeatsLeft is Num - SeatsReqd,
   SeatsLeft >= 0, !,                     % enough seats ?
   asserta(db_seats(Flight, SeatsLeft)),  % update database
   retract(db_seats(Flight, Num)).        % delete old info
reserve(Flight, SeatsReqd, failed).
```

Fig. 5. Handling Flight Requests

```
lock_flight(Flight) :-
   mbx_create(ReplyBox),                   % new mailbox for reply
   mbx_getid(lock_manager, Mbx),           % where to send to
   mbx_send(Mbx, lock(Flight, ReplyBox)),  % send lock message
   mbx_recv(ReplyBox, yes),                % wait for permission to proceed
   mbx_close(ReplyBox).

unlock_flight(Flight) :-
   mbx_getid(lock_manager, Mbx),           % where to send to
   mbx_send(Mbx, unlock(Flight)).          % send unlock message
```

Fig. 6. Access Control Operations

```
locked(FlightNum, WaitingList)
```

where `WaitingList` is a queue of mailbox identifiers to notify when the current lock is removed. To guarantee the consistency of the database, it is crucial that the lock manager performs its operations sequentially rather than concurrently. The code is shown in Figure 7.

To start the reservations system, we start the session manager and lock manager in two separate threads like this :

```
| ?- fork(lock_manager), fork(session_manager).
```

```
lock_manager :-
   mbx_create(Mbx),                    % create mailbox for access requests
   mbx_bind(Mbx, lock_manager),        % make mailbox identifier public
   manage_locks(Mbx, []).              % initialise with no locks

manage_locks(Mbx, Locks) :-
   mbx_recv(Mbx, Request),             % received new request
   lock_action(Request, Mbx, Locks).   % process the request

/*
  If flight number is already locked, add this
  request to the queue of waiting threads
*/
lock_action(lock(Flight, ReplyBox), Mbx, Locks) :-
   member(locked(Flight,Waiting), Locks), !,     % already locked
   delete(locked(Flight,Waiting), Locks, Residue),
   append(Waiting, [ReplyBox], NewWaiting),
   manage_locks(Mbx, [locked(Flight,NewWaiting) | Residue] ).
/*
  If flight number is not locked, notify requesting thread to
  proceed and add new entry to list of locks, initialising
  the queue of waiting threads to be empty.
*/
lock_action(lock(Flight, ReplyBox), Mbx, Locks) :-
   mbx_send(ReplyBox, yes),
   manage_locks(Mbx, [locked(Flight,[]) | Locks] ).
/*
  When unlocking, if there are other threads waiting for this
  flight number, notify the first in queue to proceed. If
  no thread is waiting, remove the entry from list of locks.
*/
lock_action(unlock(Flight), Mbx, Locks) :-
   delete(locked(Flight,Waiting), Locks, Residue),
   ( Waiting = [First|Rest] ->
       mbx_send(First, yes),
       manage_locks(Mbx, [locked(Flight,Rest) | Residue] )
   ;   manage_locks(Mbx, Residue)
   ).
```

Fig. 7. Lock Manager Program

This completes the description of the server code being run on the airline's computer. A simple client interface program that could be run on any travel agent's computer worldwide[8] is shown in Figure 8. The travel agent or indeed any computer user connects to the Imperial Airlines reservations system by calling the goal `imperial_airlines`. He then interacts with the system, sending requests and getting replies until he decides to quit.

This example has shown that it is possible to write network-ready applications involving concurrency and synchronisation using the `ICP` system. [5] shows `ICP` being used to specify another example of a multi-agent system involving cooperative problem-solving.

7 Related Work

`ICP` belongs to the class of Process Oriented Prologs. The processes or threads are explicitly

[8] as long as it is connected to the Internet

```
imperial_airlines :-
   /* Connect to Imperial Airlines reservations system    */
   /* The address is public knowledge, so it is hardwired */
   tcp_client(6789,'146.169.21.1',Socket),
   session(Socket).

session(Socket) :-
   write('request : '), flush,     % display prompt
   read(Req),                      % read request
   tcp_send(Socket, Req),          % send request
   ( Req == quit ->                % finished ?
       tcp_close(Socket)
   ;   tcp_recv(Socket, Reply),    % wait for reply
       write(Reply), nl,           % display reply
       session(Socket)             % recurse
   ).
```

Fig. 8. Simple Airline Reservation Client Program

forked sequential Prolog programs. The processes run concurrently and communicate using explicit asynchronous message passing.

Delta Prolog [18] is the oldest of the Process Oriented Prologs. It defines constructs for split goals, event goals and choice goals, which roughly correspond to ICP's forking of threads, inter-thread communication and Parlog's OR-parallel search respectively. There are notable differences however. In split goals and event goals, two-way unification and distributed backtracking are implicit and fundamental. In contrast, ICP has uni-directional message passing only and no distributed backtracking. This is less powerful but much simpler to implement. We believe that two-way unification and distributed backtracking may be simulated explicitly using message passing in ICP on the occasions when it is needed. Delta Prolog's choice goals can be viewed as a form of committed-choice non-determinism restricted to having event goals only in the guard. The Parlog thread in ICP is more general and does not suffer from this restriction.

CS-Prolog [13] has the same expressive power as Delta Prolog but makes a distinction between backtrackable and non-backtrackable primitives for creating new processes and receiving messages. The authors recommend the use of the non-backtrackable versions of the primitives for practical applications since, in common with Delta Prolog, there is a heavy performance penalty for using distributed backtracking. The facilities provided in ICP correspond only to the non-backtrackable primitives.

The system most closely related to ICP is PMS-Prolog [24]. Although Delta Prolog and CS-Prolog allows concurrent processes, the scheduling is data-driven – there is no time sharing. PMS-Prolog has a scheduler which pre-empts processes after a fixed number of Prolog calls whereas ICP uses fixed time-slices. Both systems do not allow backtracking on communication and therefore rely on coarse-grain Prolog processes to provide messages with high information content. PMS-Prolog enforces this style by adding a module construct to Prolog which allows coarse-grain processes to be declared. In ICP, we can use the object-oriented extensions [16] to give a similar structuring facility though it is not obligatory to use it.

ICP has better support for highly parallel programs than the above three systems simply because it has a built-in Parlog thread. The combination of Parlog and Prolog offers greater expressibility compared to the rather limited form of don't-care non-determinism supported by other Prolog systems. Also, whereas the other systems are aimed at executing programs on multi-processor machines, ICP offers network communication facilities to enable programs to communicate over the network.

There are other systems such as Shared Prolog [2], Multi-Prolog [12] and Linda Prolog which allow communication between Prolog programs. These all use a blackboard communication model. We believe that blackboard systems are inherently restricted as a medium for inter-process com-

munication because the blackboard itself becomes a bottleneck in the system.

8 Conclusions

`ICP` does not aim to give a performance improvement over normal sequential Prolog systems, as the large body of work in parallelising Prolog aims to do. Rather, we aim to expand the expressive power of Prolog to encompass distributed applications.

`ICP` has a comprehensive set of communication primitives but this is of no great significance on its own. Many Prolog systems have interfaces or can easily construct interfaces to TCP/IP to allow network communication. For example, both Quintus Prolog [20] and SICStus Prolog [4] include a TCP/IP interface with their distribution. However, since neither of these systems offer multiple threads, network programming is in our opinion rather difficult. The application needs to be turned 'upside-down' into a single threaded event-driven loop or rely on interrupt handling. This is analogous to the process of converting a teletype-based application to use a graphical user interface. It requires a totally different programming style. Even then, they are unable to truly time-share between two concurrent goals which we contend is a prerequisite of network-friendly applications. In contrast, the same programs written in `ICP` retains the style of a sequential Prolog application and therfore we maintain, is more natural to write.

`ICP` is suitable for programming applications such as cooperating expert systems where each expert system resides on a different machine connected through a local area network. This opens up the more general application area of Multi-Agent Systems and Distributed Artificial Intelligence, where distributed logic programming has much to offer.

A compiled version of the `ICP` system for Sun Sparc machines is available by anonymous ftp from `src.doc.ic.ac.uk` (Internet: 146.169.2.1) in the directory

```
/computing/programming/languages/prolog/icprolog
```


Acknowledgements

The author wishes to thank Frank McCabe and Keith Clark for many valuable comments and discussions. This work was supported by the European Commission under the ESPRIT program project IMAGINE (project number 5362).

References

[1] K. A. M. Ali and R. Karlsson. The Muse Or-Parallel Prolog Model and its Performance. In S. Debray and M. Hermenegildo, editors, *Proceedings of the North American Conference on Logic Programming*, pages 747–768, Austin, 1990. MIT Press.

[2] A. Brogi and P. Ciancarini. The Concurrent Language Shared Prolog. *ACM Transactions on Programming Languages and Systems*, 13(1):99–123, January 1991.

[3] M. Carlsson. Freeze, Indexing and other Implementation Issues in the WAM. In *Proceedings of the Fourth International Conference on Logic Programming*, pages 40–58, Melbourne, 1987.

[4] M. Carlsson and J. Widen. SICStus Prolog User Manual. Research Report R88007B, Swedish Institute of Computer Science, Kista, 1988.

[5] D. A. Chu. I.C. Prolog II : a Language for Implementing Multi-Agent Systems. In S. M. Deen, editor, *Proceedings of the SIG on Cooperating Knowledged Based Systems*, pages 61–74. DAKE Centre, University of Keele, 1993.

[6] K. L. Clark and S. Gregory. Parlog and Prolog United. In *Proceedings of the Fourth International Conference on Logic Programming*, pages 927–961, Melbourne, 1987. MIT Press.

[7] K. L. Clark, F. G. McCabe, and S. Gregory. IC-PROLOG Language Features. In K. L. Clark and S.-A. Tarnlund, editors, *Logic Programming*, pages 253–266. Academic Press, London, 1982.

[8] A. Colmerauer. *Prolog-II Manuel de Reference et Modele Theorique*. Groupe d'Intelligence Artificielle, Universite d'Aix-Marseille, Luminy.

[9] Y. Cosmadopoulos and D. A. Chu. *IC Prolog II Reference Manual.* Logic Programming Section, Dept. of Computing, Imperial College, London, 1993.

[10] V. S. Costa, R. Yang, and D. H. D. Warren. The Andorra-I Engine: A Parallel Implementation of the Basic Andorra Model. In *Eighth International Conference on Logic Programming*, Paris, 1991. MIT Press.

[11] J. Crammond. The Abstract Machine and Implementation of Parallel Parlog. *New Generation Computing*, 10:385–422, 1992.

[12] K. De Bosschere. Multi-Prolog, Another Approach for Parallelizing Prolog. In D. J. Evans, G. R. Joubert, and F. J. Peters, editors, *Proceedings of Parallel Computing*, pages 443–448, Leiden, 1989. Elsevier North Holland.

[13] Sz. Ferenczi and I. Futo. CS-Prolog: a Communicating Sequential Prolog. In P. Kacsuk and M. Wise, editors, *Implementations of Distributed Prolog*, pages 357–378. John Wiley & Sons, Chichester, 1982.

[14] S. Gregory. *Parallel Logic Programming in PARLOG.* International Series in Logic Programming. Addison-Wesley Publishing Company, Wokingham, 1987.

[15] E. Lusk, D. H. D. Warren, and S. Haridi. The Aurora OR-Parallel Prolog System. *New Generation Computing*, 7(2,3):243–271, 1990.

[16] F. G. McCabe. *Logic and Objects.* Prentice Hall international series in computer science. Prentice Hall International (UK) Ltd., Hemel Hempstead, 1992.

[17] L. Naish. Parallelizing NU-Prolog. In R. A. Kowalski and K. A. Bowen, editors, *Proceedings of the Fifth International Conference and Symposium on Logic Programming*, pages 1546–1564, Seattle, 1988. MIT Press.

[18] L. M. Pereira and R. Nasr. Delta-Prolog: A Distributed Logic Programming Language. In *Proceedings of the International Conference on Fifth Generation Computer Systems*, pages 283–291, Tokyo, 1984.

[19] A. Porto. Epilog: a Language for Extended Programming in Logic. In J. A. Campbell, editor, *Implementations of Prolog*, pages 268–278. Ellis Horwood, Chichester, 1984.

[20] Quintus Corporation, Palo Alto. *Manual for Quintus Prolog Release 3.1*, 1991.

[21] W. R. Stevens. *UNIX Network Programming.* Prentice Hall Software Series. Prentice-Hall, New Jersey, 1990.

[22] J. Thom and J. Zobel. NU-Prolog Reference Manual, version 1.0. Technical Report 86/10, Dept. of Computer Science, University of Melbourne, 1986.

[23] D. H. D. Warren. An Abstract Prolog Instruction Set. Technical Note 309, Artificial Intelligence Center, SRI International, October 1983.

[24] M. J. Wise, D. G. Jones, and T. Hintz. PMS-Prolog: a Distributed, Coarse-grain-parallel Prolog with Processes, Modules and Streams. In P. Kacsuk and M. Wise, editors, *Implementations of Distributed Prolog*, pages 379–403. John Wiley & Sons, Chichester, 1982.

This article was processed using the LaTeX macro package with LLNCS style