The Amoeba-Prolog System

I. SHIZGAL
Computer Laboratory, University of Newcastle upon Tyne*, Claremont Tower, Claremont Road, Newcastle upon Tyne NE1 7RU

Although much good work has been done in the area of distributed/concurrent Prolog, this has concentrated on schemes that use multiple processors with the goal of executing some form of logic program more quickly than can be done on a sequential processor.

The present paper addresses the distributed use of logic programming, i.e. the sort of usage with the characteristics: (a) processes with no relationship, needing to establish a connection, cooperate for a while, and separate again; (b) processes where, averaged over process lifetime, the communication density and number of synchronisations between most pairs of processes is low. It takes the Amoeba distributed operating system and provides predicates for performing Amoeba transactions using Prolog terms.

Received May 1989, revised May 1990

1. INTRODUCTION

Current work in the area of distributed/concurrent Prolog is concentrated on schemes that use multiple processors, whether loosely or tightly coupled, with the goal of executing some form of logic program more quickly than can be done on a sequential processor. This includes, as two broad categories, languages that attempt to preserve the search characteristics of sequential Prolog and languages that do not. These domains may be characterised generally as consisting of programs performing tightly coupled computations, whether or not the physical computing substrate is tightly coupled, because of their high communication bandwidth and the number of synchronisations required. Although much good work has been done, the success in terms of significant performance gains in a usable system is somewhat debatable.

In contrast to this, comparatively little work has been done in the area of distributed use of logic programming, although existing work is applicable to some extent. The characteristics that distinguish this sort of usage are these:

(i) Processes with no relationship (i.e. no common ancestor in the process-tree sense) must be able to establish a connection, cooperate for a while, and separate again.
(ii) Averaged over process lifetime, the communication density and number of synchronisations between most pairs of processes (or process groups) is low.

This is the model typically found in distributed systems organised on the client–server model, such as Amoeba, V and Chorus.

1.1 Brief summary of this work

This paper describes work oriented toward using Prolog in such an environment, specifically the Amoeba distributed operating system. The simplest model of communication is used – predicates are provided for performing Amoeba transactions using Prolog terms.

An interesting feature is the ability of an AmProlog program to duplicate itself dynamically on to another virtual Prolog machine. It may be used to build fairly sophisticated models of distributed systems quickly, including worm-type programs which can be modelled in a controlled, safe environment entirely in Prolog.

1.2 Layout of this paper

Section 2 describes the organisation of the AmProlog system. A brief introduction to the semantics of the Amoeba primitives is given in Section 3. Some sample programs are given in Section 4. Section 5 describes the mechanisms used to implement communication of Prolog terms. The program propagation mechanism and a discussion of the design choices in its implementation are the subject of Section 6. Finally, possible improvements and directions for further work are presented in Section 7.

2. AMPROLOG CONFIGURATION

2.1 Overall organisation

Two important goals guiding the design of AmProlog were portability, and the ability to produce normal executable files with no special properties deriving from their origin as Prolog programs. Consequently the basic design is a simulator written in C for an abstract machine almost identical to the Warren abstract machine (colloquially referred to as the WAM). The Prolog system itself was never planned for general release, but was intended from the beginning specifically to be a vehicle for experimentation. Thus the possibility was foreseen that portions might have to be rewritten or significantly modified in the course of trying ideas. With this in mind – and the extra man-hours that would result for both the initial effort and for all subsequent structural changes to a full-featured implementation – it was decided to restrict the implementation to only those features deemed absolutely necessary for the intended type of usage. This has resulted in a system that is easily manageable and understandable, but resembles conventional programming language environments in that it lacks an interactive mode.

A program (written in C) accepts Prolog source as input, and generates C source as output. C source is a convenient output format. Aside from being human-readable (easier to verify output), by having the compiler...
generate the C code that dispatches to built-in predicates, references are generated only to those built-in predicates that are needed, avoiding the inclusion of unneeded code in the executable. This indirect approach avoids the tendency of interpreter-based systems to statically link in all code that might ever be required at run time. The generated source is compiled using the C compiler and linked to the WAM simulator and library of built-in predicates. The final result of the compilation process is a standard executable file with no indication of its Prolog origins, and no implicit run-time dependencies on any auxiliary files.

From the point of view of the Prolog program, execution begins by trying to solve the goal:

\[ \text{go(Command line)} \]

where the \textit{Command line} argument consists of a list of atoms built from the standard C language argv vector.

2.2 Version support

The compiler exists in four versions. These are a pure version, for systems with a C compiler and a minimal Unix-compatibility library, a Unix version which takes advantage of more advanced Unix features, an AmUnix version, for Unix systems extended with the Amoeba transaction primitives, and a native Amoeba version generating files suitable for native Amoeba systems. By the simple expedient of avoiding obvious Unix-isms, AmProlog code can be written that can be compiled either for AmUnix or native Amoeba. This is a useful property in the evolving Amoeba environment.

The primary requirements for porting the system are that the target have a C compiler and be byte-addressable. (This results from the use of the two lowest address bits as a tag, assuming that a pointer with these bits zeroed addresses a 32-bit word.) It has been successfully compiled for and run on a variety of machines, including various VAX versions, SUN/3, SUN/4 (SPARC), Hewlett-Packard Bobcat, Harris HCX-7 (sometimes known as the Tahoe processor), and the MIPS R2000 (Digital Equipment Corporation 3100 workstation).

2.3 Performance

The speed of execution of AmProlog programs should reveal no surprises to Prolog system implementors. Using the standard naive-reverse benchmark (including the execution of 13 garbage collections, so that this is the steady-state rate) yields about 2700 LIPS on a VAX-station 2000 (which uses Digital Equipment Corporation's Microvax II chip set -- about 0.9 VAX mips), and about 7400 LIPS on the Sun 3/60. This is about twice the speed of a completely interpreted system, and approximately one-eighth the speed of a completely compiled and optimised implementation.

2.4 Prolog syntactic and lexical conventions

Prolog programs and code fragments in this text adhere to the de facto standard promulgated by Clocksin and Mellish in their now classic book.\(^5\) Although predominantly consistent with convention, the AmProlog system has some syntactic idiosyncrasies which have been removed in the illustrative code.

Variables are syntactically distinguished by beginning with either an upper-case letter or an underscore, where a lone underscore denotes an anonymous variable. Atomic symbols begin with a lower-case letter or are enclosed in single quotes. Integers are also accepted as atomic symbols.

3. Amoeba background

Amoeba is a distributed operating system designed for use on a local area network.\(^{12,13}\) It is based on the use of synchronous (i.e. unbuffered) message transactions, resembling remote procedure calls, for its main interprocess communication primitives.

3.1 Communication primitives

An Amoeba transaction is based on a client–server relationship. A server on the network issues a get\_request\(\text{(Port, Request_buffer)}\) call. A client process issues a transaction\(\text{(Port, Request_buffer, Reply_buffer)}\) call which sends a message to a server awaiting a request on the specified port number. The server get\_request\(\text{()}\) blocks until it is invoked by a matching transaction\(\text{(})\) by a client, which is in turn blocked until the server completes the transaction by issuing a put\_reply\(\text{(Reply_msg)}\). (Note that the actual call arguments are slightly more complicated and numerous, but have been simplified here for clarity.) The ports are abstract network addresses from a very large address space not corresponding to the physical network addresses space. In particular, a port number does not fix the location of a server accepting messages on it, and indeed multiple processes at varying physical addresses may be simultaneously listening for requests on the same Amoeba port. The Amoeba transaction semantics guarantee that at most one server waiting for a request on a given port will respond to a transaction request on that port.

Amoeba is object-based in the sense that operations are performed on objects named by capabilities (in the sense of Dennis and van Horn,\(^5\) Fabry\(^6\) and Levy,\(^10\) where a capability is like a ticket which gives the holder certain rights to access the object). An Amoeba capability contains the port of a server supporting the named object, and some information identifying the specific object referenced and permissions on it. The forging of capabilities and protection from impostor servers are accomplished via encryption techniques.

3.2 Processes and threads in Amoeba

Since the Amoeba transaction system calls block the caller, concurrency must be achieved by partitioning a job among a number of distinct processes. To support this efficiently, Amoeba supports two levels of processes. The ‘heavyweight’ process (referred to simply as a process) is the level at which a virtual address space is allocated. Within a process is a set of ‘lightweight’ processes (referred to as threads) whose state consists of a thread of execution and a transaction state. The overhead of switching between and starting threads within a process is low, since no adjustment of the memory map or duplication of memory segments is done. Thus the light/heavy nomenclature reflects the costs of the two types of process.
3.3 AmProlog predicates for Amoeba transactions

The basic transaction functions described above (as well as several others not mentioned but required for proper control, such as the setting of the server-locate timeout interval, and transaction-abort call) have their direct counterparts in AmProlog. The major difference is that these operate entirely on Prolog terms instead of unstructured buffers and binary ports and capabilities. Below are brief descriptions of the communication predicates needed to understand the example code. These AmProlog calls map one-to-one on to the underlying Amoeba system calls, with details such as term encoding, buffer management, etc. handled by the AmProlog runtime, which acts as an insulating layer between the Prolog user and the raw Amoeba calls. In the following descriptions the Port argument may also be a complete Amoeba capability.

- **trans**(Port, Request_to_send, Reply_received, Error_status)

  Port is the port of the server being called. Request_to_send is the term to be sent to the server. The Reply_received argument is unified with the term returned by the server, and the Error_status argument is unified with the status code for the transaction() operation itself (OK, failure, timeout, abort, etc.) The caller remains blocked until either an error occurs (e.g. timeout – failure to find a server) or the server’s reply is received.

- **getreq**(Port, Request_received, Error_status)

  Port is the port on which requests are to be accepted. Request_received is unified with the request message received from the client, and the Error_status is unified, as above, with the status of the get_request() call itself. The caller remains blocked until a client request is received.

- **putrep**(Reply_to_send, Error_status)

  Reply_to_send is the term to be returned to the client, terminating the interaction. Error_status is once again the status of the put_reply() operation itself. Note that no port is needed since transactions can never overlap within a thread. (Issuing further getreq calls without a putrep may fail or cause an exception, depending on the specific Amoeba implementation.)

4. EXAMPLES

4.1 Typical usage

The ordinary mode of operation of programs (whether AmProlog or otherwise) is of various different processes cooperating with each other via Amoeba transactions, some acting as servers, some as clients, many alternating between server and client behaviour. Such processes may have been created by the user himself, by colleagues, system support staff, or other users rendering specific services. An Amoeba system is usually equipped with a publicly accessible directory service to allow binding symbolic names to capabilities. Servers intended for general use have entries in publicly accessible directories maintained by the directory service, and a user’s personal (login) environment contains the capability of his home directory, which is his starting point for access to the general directory graph.

As an illustration an AmProlog program which accesses a database server might be written as follows.

Note that with suitable organisation, a technique similar to the familiar notion of remote procedure call stubs can be used.

```prolog
... database(Query, Result), ...
... database(Db_request, Result) :-
  dir_entry('servers/public',
  database, Capability),
  trans(Capability, Db_request,
  Db_reply, Error),
  database_process_response(Error, Db_reply, Result).
... database_process_response(ok, Reply, Reply). % normal response
... database_process_response(timeout, Reply, <special result>) :-
  <special error actions>
... database_process_response(abort, Reply, <special result>) :-
  <special error actions>
... etc.
```

The rest of the examples illustrate the use of the more unusual program propagation ability.

4.2 A generic host program

The simple code illustrated here establishes a virtual machine serving on a hard-wired port. (In general it is more useful to supply the service port via a file, environment variable, or command-line argument.) A pool of such processes can be set up on various network machines to achieve concurrency, or on a single machine (one per window on a workstation, for example) to ease debugging of a complex program. By choosing a private port, a completely contained environment can be created.

```prolog
go :-
  run.
run :-
  host(port(45,37,80,212,146,36)), !, run.
run :-
```

The host/1 predicate is a built-in predicate that accepts a propagation request on the specified port. When (and if) the host subgoal succeeds, the guest program will have executed to completion. A failed host/1 call indicates a propagation attempt that was refused. This can be the result of a protocol error, a failed transaction within the protocol, or a request that required more resources than the host machine could provide.

4.3 A generic host hunter-killer for the AmUnix environment

This program is designed to search for and kill host programs of the above type. It takes advantage of fairly advanced Unix features, and so is suitable only for the AmUnix environment.

It works by repeatedly trying to propagate itself to a
host. Each time it succeeds, the child copy of the program finds the host machine name and Unix process identifier of the host process. It then communicates this to the parent, which uses the information to kill the child’s host process using the Unix ‘rsh’ (remote shell) command. This assumes that the owner of the hunter/killer process has permission to terminate the host processes, but in the context of an experimental or debugging run this is not an unreasonable expectation. The program has actually been used for this purpose.

\[
g emails :=
\]

\[
\text{kill_hosts(port}(45,37,80,212,146,36))\text{.}
\]

\[
\text{kill_hosts(Hostport) :=}
\]

\[
\text{do_propagate(Hostport, Link),}
\]

\[
\text{getreq(Link, Hostname, Err),}
\]

\[
\text{putreq(ack, _),}
\]

\[
\text{kill(Hostname, Err),}
\]

\[
\text{kill_hosts(Hostport).}
\]

\[
\text{kill_hosts(_ :=)}
\]

\[
\text{print(done), nl.}
\]

It is the \text{do_propagate(Hostport, Linkport)} call in the \text{kill_hosts/1} predicate that performs the propagation operation. The \text{Hostport} input argument is the host service port, and the \text{Linkport} argument is unified with a unique port connecting parent and child. This tight loop will continue as long as there is an available host to propagate to. When no further hosts are available (this does not imply they are all dead) the propagation attempt will fail, terminating the loop. A successful propagation is immediately followed by a transaction call from the child to pass the process identifier to the parent.

\[
\text{kill(Hostname, ok :=}
\]

\[
\text{split(Hostname, ‘’, Host, Pid),}
\]

\[
\text{system([rsh, Host, kill, Pid], 0),}
\]

\[
\text{print(Hostname, ‘killed’), nl.}
\]

\[
\text{p_continue(Link) :=}
\]

\[
\text{host_name(Me),}
\]

\[
\text{trans(Link, Me, ack, _).}
\]

The final \text{p_continue/1} predicate is used as the starting goal for the child program. The \text{p_continue/1} goal is called automatically with the \text{Link} argument set to the same unique port as that returned to the parent through its \text{do_propagate} call. The built-in predicate \text{host_name/1} returns a name constructed from the host machine name and process identifier, thus uniquely identifying the process.

4.4 A more interesting example – a self-replicating server

This program is a skeleton for a replicated server that reliably maintains a preset number of copies of itself in the face of crashes (provided that enough independent AmProlog hosts are available). The example here tries to ensure that there are at least three copies of itself running at all times. This is achieved by periodically executing a ‘configuration check’ command that enumerates all available servers and results in one server generating replacement copies if the number falls below the programmed limit. (Note that the case of a server which is busy and thus unable to respond to a configuration check is not handled. The replication factor will thus be a lower bound, but not an upper bound. Clearly the configuration check could be modified so that any detected excess servers would be signalled that they are no longer required.)

In this design each copy of the program makes use of a special built-in predicate \text{delay(P, M, N)} which creates a thread that sends the message \text{M} in a transaction on port \text{P} after a delay of \text{N} seconds. This is used to generate a configuration-check message after a fixed interval, with each program instance choosing a different interval based on its ordinal position in the last enumeration. This means there will be one outstanding configuration check message for each server instance. To ensure that not all of these actually result in (rather expensive) enumerations being initiated these messages have embedded serial numbers. Each completed configuration check advances the serial numbers of all participating processes, which allows old configuration messages to be discarded without being acted upon. Since the delayed wake-up message is duplicated everywhere a server is, a single surviving server would eventually try to bring the service up to its full complement.

On startup (with the initial \text{go} clause) the program sets the timeout interval for transactions, creates a copy of itself (using the \text{replace/1} predicate) and exits. Meanwhile the copy it has spun off sends a delayed configuration check message and enters its main server loop. When executed the program thus seems to terminate quickly, but launches a full long-lived replicated system.

\[
\%-----------------------------
\]

\[
\% The server configuration structure:
\]

\[
\% config(Service_port, Serial_no,
\]

\[
\% Current_replication, Ideal_replication)
\]

\[
\%-----------------------------
\]

\[
\% Initial goal
\]

\[
\% go :-
\]

\[
\text{timeout(10),}
\]

\[
\text{replace(config(port(221,208,0,201,48,206),1,1,5)).}
\]

The main loop(\text{serve/1}, below) waits for a request and dispatches it via the command switch. It never terminates, program termination being accomplished by explicit command.

\[
\%-----------------------------
\]

\[
\% Main server loop
\]

\[
\% Arg1- configuration
\]

\[
\text{serve(Config) :-}
\]

\[
\text{arg(1, Config, Sp),}
\]

\[
\text{getreq(Sp, Req, ok),}
\]

\[
\text{cmd(Req, Config, Configp),}
\]

\[
\text{!,}
\]

\[
\text{serve(Configp).}
\]

\[
\text{serve(Config) :-}
\]

\[
\text{serve(Config).
\]

\[
\]

\[
\]

\[
\% There is also a corresponding predicate for spinning off an asynchronous server. This is the predicate \text{supply(X, P, M). It starts a}
\]

\[
\text{thread that responds to the next X transactions on a port P with a pre-}
\]

\[
\text{programmed response message M, ignoring the contents of the requests.}
\]

\[
\text{It can be used to create a supply of message resources, or to discard a}
\]

\[
\text{series of transactions, replying to each with a ‘pre-recorded’ message.}
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]
The command switch (below) is perfectly straightforward. The last clause acts as a catch-all for unrecognised commands. In this skeleton server only three commands are recognised:

**check:** A configuration-check command that enumerates the currently available servers to see if replacements are required.

**who:** A test command that merely collects the machine and process identifiers of the participating processes.

**kill:** A command allowing a single server instance to be terminated.

```prolog
%---------------------
% Main service command switch
% % Arg1- command (in)
% % Arg2- service configuration (in)
% % Arg3- service configuration (out)
cmd(check(Ser_no,I), Oldconfig, Newconfig) :-
  check(Ser_no,I, Oldconfig, Newconfig).

cmd(who(L, Config, Config) :-
  who(L, Config).

cmd(kill, _, _) :-
  kill.
% error catch-all
cmd(_, Config, Config) :-
  putrep(error, _).
```

Next we have predicates for handling each of the individual commands.

The configuration check command (check/4) consists of two clauses, the first of which accepts messages with valid serial numbers, and the second of which responds to messages with outdated serial numbers, taking no further action. Its auxiliary predicate (check_1/5) distinguishes between the case where it is not the last server in the nested call chain (clause 1), and the case where it is (clause 2). The main distinction between these cases is the value accepted as a valid count of active servers. If the recursive call succeeded, then the correct count was returned in its reply message. Otherwise the correct value is one more than the count we received as input.

```prolog
%---------------------
% Configuration check command
% % Arg1 - serial number of request
% % Arg2 - number processes in the chain ahead of this one
% % Arg3 - old configuration
% % Arg4 - new configuration
check(Ser_no, I, config(Sp, Ser_old, _, Irep), Config_out) :-
  Ser_no > Ser_old,
  I is I+1,
  trans(Sp, check(Ser_no, I), Reply, Err),
  check_1(Err, Reply, Ser_no, I, config(Sp, Ser_old, _, Irep),
    Config_out).

check(_, C, C) :-
  putrep(error, _).
% no error on recursive call
```

The predicate next_check/3 generates a delayed configuration check message. Note that since the delay is a function of the server's position in the enumeration chain (passed in argument I), each server will use a different delay.

```prolog
% recursive call failed
check_1(_, _, Ser, I, config(Sp, _, Irep),
  config(Sp, Ser, I, Irep)) :-
  putrep(check(Ser_no, N), _,
    K is Irep-N, replacements(K, I, config(Sp, Ser_no, Irep, Irep)).
```

The predicate `next_check/3` generates a delayed configuration check message. Note that since the delay is a function of the server's position in the enumeration chain (passed in argument I), each server will use a different delay.

```prolog
% Set a timer for the next configuration check
% % Arg1 - position of this process in the chain
% % Arg2 - serial number of current configuration
% % Arg3 - service port
next_check(I, Ser_no, Sp) :-
  Ser_nxt is Ser_no+1,
  Delta is 10*I+1,
  delay(Sp, check(Ser_nxt, 0), Delta), !.
  next_check(_, _, _).
```

The `who` command takes no part in configuration maintenance. It serves as a demonstration command which merely obtains a list of server process names (and incidentally serves as a simple illustration of how an enumeration works). A list of server names is accepted as input, the server adds its own name to the front of the list, and passes the list recursively to the next server. If there is a next server then its reply contains the complete list, otherwise the list constructed locally is already complete.

```prolog
%---------------------
% Handler for 'who' command
% % Arg1 - list of servers enumerated so far
% % Arg2 - current configuration
who(List_in, config(Sp, Ser, N, _)) :-
  host_name(me),
  timeout(10),
  Mylist = [e(Me, Ser, N)][List_in],
  trans(Sp, who(Mylist, Listp, Err),
    who_1(Err, Mylist, Listp).
```

512 THE COMPUTER JOURNAL, VOL. 33, NO. 6, 1990
Here is the \texttt{p\_continue/1} predicate which serves as the starting goal in the child copy of the program. All that need be done in this case is to obtain the current system configuration from the parent via a transaction on the link port, set our own configuration-check timer (using \texttt{next\_check/3}), and enter the main server loop.

\begin{verbatim}
\%---------------------
\% handler for 'kill' command
\%---------------------
kil:-
host\_name(Me),
putrep(ack(kill, Me), _),
exi.\end{verbatim}

The following code does the actual generation of replacement servers. Since the check message generated a number of participating servers, we can now use the number to ensure that only one server (number 1) actually generates replacement copies. Thus only server 1 will succeed in unifying with the constant 1 in the second argument of the first clause for \texttt{replacements/3} and loop until the required number of duplicates have been generated. All other servers will use the alternative clause, which does nothing.

\begin{verbatim}
\%---------------------
\% Generate replacement server instances
\% Arg1 - count of server instances to generate
\% Arg2 - position of this server in the enumeration chain
\% Arg3 - current configuration
\%---------------------
replacements(K, 1, Config) :-
K >= 1, !,
replace(Config),
K is K-1,
replacements(K, 1, Config).
replacements(_, _, _).
\end{verbatim}

The \texttt{replace/1} predicate does the actual work of generating a replacement server instance. Note that the actual propagation operation is followed by a transaction with the child (via the automatically generated link port) to transmit the configuration information. In a more realistic example it would also be necessary to exchange replicated information at this point.

\begin{verbatim}
replace(config(Sp, Ser, _, Irep)) :-
do\_propagate(port(45, 37, 80, 212, 146, 36), Link),
getreq(Link, _, ok),
putrep(config(Sp, Ser, Irep, Irep), ok),
\% failure can be logged here
replace(_).
\end{verbatim}

\section{Encoding of Information for Network Transmission}

The encoding of general Prolog terms for network transmission, and the transfer of complete symbol tables for the purposes of program propagation, are accomplished using a mixture of code written in the underlying implementation language (C, in this case) and Prolog.

In both types of communication there is a dependency on a fixed number of native atoms which all instances of the Prolog virtual machine understand, and for which they all agree on the functor numbers (i.e. the representations of those functors match exactly). These are required both to ensure that arithmetic expressions have the same meaning when evaluated by different virtual machine instances (without forcing the expression evaluation predicate to deal with strings), and to supply a set of common definitions which forms a basis for communication.

Prolog term encoding has been separated into two parts.

(a) \textit{Encoding of symbol table information}. Atom symbol definitions are required in order to supply interpretations for the functors in a term. Integer constants can be understood without any special arrangements.

(b) \textit{Encoding of structural information}. Prolog terms must be traversed and flattened into a standard representation.

A message is first processed by Prolog predicates (the code for which is included from a standard file) that recast the message into a form containing no information dependent on the local context. This sanitised message is passed on to the built-in communication predicate to be invoked, which automatically packs it into a message buffer and transmits it.

A special-purpose built-in predicate converts between an atomic symbol and a literal representation of the string value associated with that symbol. These literal strings are supported as an extended type of Prolog object that unifies with nothing except an unbound variable. They can, however, be passed around (uninterpreted) as part of normal Prolog structures as long as they are treated only as 'magic cookies' by all except special predicates. (These actually have other uses, which are not relevant here.)
5.1 Symbol table information
The pre-processing performed on terms removes all dependencies on non-native functors. This is done by expanding symbols into literal strings and encapsulating the string values of functors with their argument lists using native symbols understood by all. Thus, using quotes to indicate literal atom strings and assuming # is a functor universally understood:

```
foo(a, b)
```

can be represented by:

```
t(‘foo’, ‘a’, ‘b’).
```

An alternative representation that could save space in a message in some cases is:

```
t(1, 2, 3)
```

with an associated atom list:

```
[ ['foo' | 1], ['a' | 2], ['b' | 3] ]
```

sent as a separate part of the message. By first traversing the atom list and unifying each variable with an atom created by entering the literal atom value into the symbol table, the term can be trivially reconstructed. Note that a compound term forms a better capsule than a list (as would be produced by the uninit/2 predicate) because a list representation would take twice the storage and require counting the list length to recover the term’s arity, whereas finding the arity of a compound term is a constant time operation.

5.2 Message format (Prolog level)
Messages presented to the lower level (the built-in predicates) for transmission are of the form:

```
msg(Time_stamp, Encoded_term, Atom_list).
```

The Time_stamp argument is the reading of the sender’s clock and is used to ensure a correct partial-ordering of message transmission events, as described in Ref. 9. (A built-in predicate time/1 is provided that allows the current clock value to be read or set forward, but never backward.) The Encoded_term argument is the term to be transmitted, with structures recursively encapsulated in one of the above-described forms. (Lists are recognised as a special case and are sent ‘native’, with head and tail encoded recursively.) The Atom_list argument is an atom list as in the alternative representation described above. Having this argument present allows the two atom representations to be mixed freely, and is of negligible cost if unused. In practice the choice is always made in favour of in-line expansion because of the speed advantage in generating it, but the option of doing otherwise in special circumstances is left open.

5.3 Structural term information
This is handled by C code. Structured terms are traversed and packed into a message buffer with normalised byte-order and relativised pointers. Variables and structure or list references are packed recursively only the first time they are encountered. Once a copy has been made the address of the structure is entered into a hash table along with the offset of the packed copy. Further references to the same structure or variable generate a packed reference of the appropriate type to the existing copy. This is essential for the correct transfer of multiple instances of the same variable, or variables that are bound only to other unbound variables. References to the same variable in the sender must reference (a new instance of) the same variable in the receiver. (However, sending and receiving of unbound variables does not result in a connection being maintained between the processes concerned.) Any other benefits resulting from this structure handling, such as the ability to pack circular structures (theoretically never required in Prolog) or saving of some copying, is incidental to the primary requirement that variables be handled correctly.

Constants, including literal atom strings, are not analysed beyond what is required by the different byte-ordering characteristics of integers or other normal Prolog values. When used with the standard communication predicate stubs this code never encounters functors known only locally.

5.4 Symbol table transfer for propagation
Unlike the case for messages in a transaction, to propagate a Prolog program over the network the program’s entire permanent symbol table must be duplicated. This is done in Prolog using special built-in predicates to enumerate the atoms in the parent program and mark them as non-garbage-collectable in the child copy.

A small digression is in order here. Atoms can be partitioned into three types in this implementation, as follows.

(a) Native atoms. These are identical in all machines, and are never garbage-collected.

(b) Program atoms. These are atoms defined in a program text. These are also exempt from garbage collection, but may not be assumed to be known by other processes.

(c) Dynamically created atoms. This category consists of atoms generated during execution of a program. They are subject to garbage collection.

Propagation of a program need not transfer atoms in category (a) (these are already available) or (c) (these are unnecessary since the entire state of the program is not transferred), but must duplicate those in category (b) exactly. Standard code has been written in Prolog, which takes care of this task correctly with the help of built-in predicates.

5.5 Comments
This dual-level encoding procedure results in the term being traversed and copied twice – once in Prolog and again in C. This is wasteful and could be improved by combining both operations (structural flattening and the construction of a list of functor definitions) into one pass implemented in C. However, the best method of representing structures was not immediately obvious, and some experimentation took place before it stabilised. Only lack of time prevented the final version from being re-coded in C for greater speed.

6. THE PROGRAM PROPAGATION MECHANISM

6.1 The meaning of propagation
‘Program propagation’ here refers to a mechanism that supports the duplication of an AmProlog program from
one virtual machine to another at run-time. It is weaker than a Unix style fork() operation in the sense that program state is not duplicated. On the other hand, the propagation notion is somewhat more general in that the copy is not restricted to the same machine, or even the same type of machine. (For example, AmProlog programs have duplicated themselves in both directions between Sun workstations and Digital Equipment Corporation VAXstation 2000 workstations – computers based on incompatible processors.) Although possible otherwise (if one had available pre-compiled executables for each processor type), this feat is made much easier because the run-time system is based on a virtual machine. The Prolog machine instructions are independent of the underlying hardware and so can be shipped between processors easily without any prior arrangements for different processor types.

6.2 State information in a Prolog program

The decision whether or not to transfer the complete state requires consideration of exactly what sort of state information there is, and what is gained or lost by transferring or not transferring various parts of it. The state information in a Prolog program can be categorised into three types. These are as follows.

(a) Explicit state information. Unlike most procedural languages Prolog programs contain no information hidden in global program-defined variables. Virtually all explicit program state information is easily accessible via variables local to clause instances that need it. The explicit state of a Prolog program is thus much simpler than that of a program in a typical procedural language. The exceptions to this statement are non-logical extensions to Prolog such as I/O streams as controlled by the family of predicates see/1, tell/1, etc. None of this information is sent automatically, but all except the hidden I/O stream can be sent explicitly. (If all I/O is performed via Amoeba transactions, then only capabilities need be sent. These are quite manageable.)

(b) Implicit state information. For a Warren-style Prolog implementation, the implicit state consists of the heap, the stack containing clause activation records and choice points, the trail, and the Prolog-machine registers. This information is not sent automatically and is not explicitly available to running programs.

(c) Program information. This consists of the code for the program clauses, and the associated symbol table. This information is sent via a combination of implementation-level code and Prolog code.

6.3 The basis for the decision to ship only code

Since there are no data structures in the Prolog implementation that are not known ahead of time, it is clearly possible to analyse the complete state of a program on one machine and reproduce an equivalent state on another. This will generally be a lot of work, both in terms of programming effort and machine cycles at execution time, and the choice must be an engineering decision that balances the cost of each choice against the expected benefit.

In the absence of empirical data on how the feature would eventually be used (since a decision on implementation obviously had to precede experience with the result) it was necessary to be guided by expected-usage scenarios. These fell into two basic types.

The first is the replication of an instance of an Amoeba server. In this case the child would be expected to either reinitialise itself from scratch, perhaps reading a file whose capability could be sent via an AmProlog transaction, or from a copy of the parent’s database, which could also be accomplished using AmProlog transactions. In either case the child copy would be expected to follow initialisation with entry of a main server loop. Duplication of the parent’s choice-point history would be wasted effort.

The second scenario is that of a program that wants to create a pool of helper processes to make explicit use of parallelism. Reproducing the choice-point history of children in such an application would be pointless unless the allocation of alternatives (as recorded in the choice points) to different cooperating processes were controlled. Unfortunately, such control implies that either slave processes must have their alternatives pre-programmed or they must negotiate for alternatives through a coordinating authority. Both of these alternatives either change the nature and goals of the project considerably, or require that so much explicit code be added to the basic program that the reorganisation into a set of cleanly defined helper processes becomes the simpler approach. (Recall that this mechanism is explicitly not being proposed as a method of achieving fine-grained parallelism.)

On the basis of the above reasoning, and in the interests of a timely implementation (this was, after all, an experiment), the decision was made in favour of transferring no implicit state information.

6.4 Usage

A program wishing to offer its virtual Prolog machine as a host does so by executing the predicate:

\[
\text{host(Host_port),(...)}
\]

The Prolog machine state of the process executing this predicate is saved and an Amoeba get_request() is done on the specified port. This results in the process being suspended until the get_request completes. If a properly formatted request is received, a special-purpose protocol is executed in which the requestor submits his resource

* The idea is to create a set of slave processes which act as servers, each awaiting a request. The master process sends a subgoal to each slave in a transaction. One possible way to control this arrangement would be to create a thread in the master to perform the transaction with each slave and have each transaction complete as its slave’s result becomes available. This presents problems in that (i) the Amoeba mechanisms for inter-thread signalling have not yet been implemented, making it difficult to abort no-longer-needed alternatives when enough successful solutions have been found, (ii) explicit use of threads would cause the AmProlog code for native Amoeba and AmUnix versions of a program to differ, and (iii) the interposition of a virtual machine between the actual machine state and Prolog execution makes explicit use of threads difficult to achieve efficiently at the Prolog level. Consequently the current method of choice is to send the slaves subgoals in completed transactions, following which the master becomes a server to await the first n replies from successful slaves. Unneeded slave transactions to report results can either be left to time out, or sloughed off using the supply/3 predicate to catch and discard unwanted reports.

---

THE COMPUTER JOURNAL, VOL. 33, NO. 6, 1990 515
requirements to the host. If the host is able to allocate the requested resources, the host’s context is pushed on to a stack, appropriate heap, stack, trail, registers, and symbol table are initialised, and the guest code is transferred.

The program wishing to duplicate itself (the parent) executes:

```plaintext
... propagate(Host_port, Link_port), ...
```

If the parent is successful the `propagate/2` predicate will succeed, and the `Link_port` argument is unified with the Amoeba port connecting parent and child. This creates a unique communication channel between each parent-child pair.

Execution in the copied code begins by trying to solve the special goal:

```plaintext
continue(Link_port)
```

where the `Link_port` argument is unified with the port supplied to the parent. The state of the host program is restored when the guest exits (whether a controlled exit or one resulting from an error), and the host program is resumed. Although the guest code may itself execute the host predicate, there would not seem to be any benefit from this. It simply falls naturally out of the re-entry of the code that implements propagation.

In practice the minimum amount of work is done in the implementation language (i.e. the new Prolog machine is initialised and the Prolog operation-code array is transferred), and execution of Prolog code is begun as soon as possible. The remaining information – the program-defined symbols – is duplicated by Prolog code with the assistance of built-in predicates. This code, like that for the standard message encoding/decoding predicates, is supplied in a standard file containing stub predicates for `propagate` and `continue`.

7. CONCLUSION

7.1 Summary

A portable Warren-style Prolog system has been described which makes use of explicit Amoeba interprocess communication operations on high-level Prolog structures, and has the unusual ability to duplicate a program across machine boundaries at run time. The basic system described here has been stable for some time, subject only to occasional minor improvements and bug fixes.

7.2 Performance improvements

7.2.1 Faster term encoding for messages

It is obviously desirable to have the encoding and decoding of Prolog terms done as quickly as possible. A significant improvement in Amoeba transaction and propagation performance could be achieved by moving all term encoding/decoding into the implementation layer as mentioned in Section 5.5.

7.2.2 Arbitrary message size handling

The current implementation suffers the limitation that it supports messages of only a single Amoeba transaction buffer in size. This is currently 32K, but plans for Amoeba version 4.0 call for a radical extension of this limit. One way of removing the current limit is, once Amoeba 4.0 is available, to allow the encoding buffer to be dynamically re-sized in the sender, and provide a new reply message type to allow the receiver of an oversized message to indicate which part of the message was lost due to buffer overflow.

7.3 Extensions / Future work

An interesting extension in the same spirit of distributed processing (see the distributed vs parallel distinction in Section 1) is to regard exported (unbound) variables as first-class Amoeba objects. Upon encountering a variable in a term being processed for export, responsibility for it would be turned over to a network variable service which would assign it a capability. All further attempts to bind or de-reference the variable could then be made only by performing a transaction on the variable’s capability, allowing the variable server to provide synchronisation. An attractive way of controlling the direction of information flow through such a variable is to allocate bits in the capability rights field to denote ‘bind’ and ‘de-reference’ permissions. The variable server would return a pair of capabilities – one with ‘bind’ permission and one with ‘reference’ permission, and it would be up to the client to decide which to send and which to keep.

Interesting questions arise naturally from such a proposal. For example, garbage collection of variables which will no longer be referenced immediately suggests itself as a problem in this scheme. Perhaps a combination of expiry times and reference counts is sufficient to deal with the problem. (Thus: a reference count maintained by the variable service is incremented each time a ‘de-referenceable’ capability for the variable is exported. The counter is decremented on each successful de-reference request, and the variable discarded when the counter reaches zero. The expiry time could ensure that debris left by crashed programs eventually disappears.)

Such work falls into an interesting area between distributed operating systems, logic programming and object-oriented programming.

REFERENCES


---

**Announcements**

9–12 September 1991

**Canterbury, Kent, United Kingdom**, IFIP Joint WG8.3/WG8.4 Working Conference on Support Functionality in the Office Environment, organised by IFIP working groups 8.3 and 8.4 in co-operation with the British Computer Society

**Second Call for Papers**

Gradual shifts in emphasis are noticeable within the fields of Decision Support Systems (DSS) and Office Information Systems (OIS). DSS apply information systems technology to increase the effectiveness of decision makers, focusing on the enhancement of human judgements. OIS are concerned with the support, and communication in connection with, human activities aimed at achieving office goals. Both are concerned with supporting organisational activity. Both have made great strides as a result of the increase in computational capability. They differ as to the aspects of organisational activity that they support, but are similar in that using such systems leads to the realisation that the systems need to be integrated.

Today DSS and OIS are blending into their organisational environments and an integrated capability is emerging. Organisations are processing their own information, within the office environment, to support their decision making. This makes it necessary to integrate DSS tools within OIS structures. The tools and infrastructure needed to permit this integrated operation can be classified under the title 'support functionality'.

If one concludes that we are moving towards the integration of DSS and OIS into a single system which is capable of providing effective support to information workers in the office, on the shop floor, on the road or wherever, then a joint conference of two TC8 Working Groups is a natural approach to a further understanding of the many problems that remain.

**Topics**

The working conference will address a number of theoretical, as well as practical issues.

Original contributions are called for that deal with any one, or a combination of, the following issues:

- approaches to designing and developing 'office support systems';
- analysis of the functionality of existing systems and requirements for future systems;
- the components (tools and structures) available or conceivable for incorporation into such systems;
- the architectures, based on such components, that satisfy the functionality requirements for decision processes in the office environment;
- the issue of designing and realising such architectures.

The conference structure will permit meetings of task groups, the presentation of technical papers, posters and videos, the running of tutorials and the demonstration of working systems.

Contributions might range over all levels of abstraction, and either concentrate on integration issues or on a closer consideration of more specific aspects of DSS operating within OIS.

**INFORMATION FOR AUTHORS**

**Technical papers and posters:**

Submit five copies of a double-spaced paper (twenty pages maximum), or electronic submission (MS-DOS document) of equivalent length to:

Professor Dr Henk G Sol
Department of Information Systems

Delft University of Technology
P O Box 356
2600 AJ Delft
The Netherlands
Phone: (31)-15-78 44 12
FAX: (31)-15-78 66 32
e-mail: WINFSOL@HDETUD1.BITNET

**Other contributions:**

Contact:

Sylvia Wilbur
Department of Computer Science
Queen Mary and Westfield College
Mile End Road
London E1 4NS
UK
Phone: (44)-81-980-4811
FAX: (44)-81-980-6533
e-mail: sylvia@uk.ac.qmw.cs

**CRITICAL DATES**

Receipt of papers: 15 January 1991
Notification of acceptance: 25 March 1991
Final paper due: 1 May 1991
Conference: 9 September 1991

**FURTHER INFORMATION**

For further information about the conference (BUT NOT ABOUT THE SUBMISSION OF PAPERS) contact:

Karen O'Sullivan
BISL
703/4 Delta Business Park
Wilton Road, Swindon SN5 7XS
Tel: (44)-793-480269
FAX: (44)-793-480270