

April - Agent PProcess Interaction Language

F.G. McCabe, K.L.Clark
Dept. of Computing
Imperial College
London
{fgm,klc}@doc.ic.ac.uk

November 25, 1994

Abstract

In this paper we introduce key features of a programming language for building DAI and other types of distributed applications requiring the transmission and manipulation of complex symbolic data. The language is high-level and yet also offers a simple and smooth interface to other programming languages such as “C”.

April is oriented to the *implementation* of multi-agent systems. However, **April** is NOT a ‘multi-agent applications language’. It does not directly offer high level features such as: planners, problem solvers and knowledge representation systems that a multi-agent applications language might be expected to include. **April** is more an object based concurrent language with objects as processes. As argued in [10], this is a highly suitable base for extension to DAI and multi-agent application platforms.

1 What is April?

Any programming language which is suitable for implementing multi-agent systems must span a diverse collection of areas. Clearly, it must have strong support for problem solving and knowledge representation; but also it must be capable of being ‘distributed’ – i.e., programs in the language should be able to execute over networks of computers and on parallel computers – and it must be able to respond to real-time events.

April is a process oriented symbolic language. It contains facilities for defining processes, and for allowing processes to communicate with each other in a distributed environment in a uniform manner. It also has powerful data structuring and expression handling features as might be found in any high level symbolic programming language. **April**’s symbolic structures are based on tuples, usable as lists, records or sets. Overall, it is aimed at giving as convenient a vehicle as possible for symbolic programming in a distributed environment.

In addition to its ‘computational aspects’, **April** also incorporates certain syntactic features – it is strongly typed, it has higher order features and it has an operator precedence syntax linked with a macro processing sub-language.

The type system that is built into **April** serves many purposes – in addition to the traditional role of types as a technique which helps to ensure that programs ‘work first time’, types in **April** also form the basis of pattern matching on messages and set-style search operations.

April’s higher order features are used both for program structuring (modules are higher order objects) and to allow functions and procedures to be passed from one process to another.

The macro-processing facility is important to the usability of the language. With it, it is possible to build language ‘layers’ on top of the basic language to incorporate some additional features. For example, it is possible to define new operators in the language as macros. We have used this facility to provide an Object Oriented extension to **April**[13].

In particular, a package of macros and library procedures could form the basis of MAIL [18] – a high level language intended to capture many common multi-agent applications. The MAIL specification, and a preliminary version of **April**, were developed as part of an ESPRIT project - Imagine. **April** was explicitly designed to serve as an implementation language for MAIL, acting as an intermediary between MAIL and C. MAIL was prototyped using a distributed logic programming system, IC-Prolog

II [3]. **April** was distilled from the features of the IC-Prolog II system that were found to be of most use in implementing MAIL.

Sadly, the ESPRIT project ended before the implementation of MAIL on **April** could be seriously investigated. Current plans are to implement more specialized multi-agent platforms on top of **April**. For example, in co-operation with British Telecom, we are investigating the use of **April** for building agent based management systems for multi-service networks, and more generally, for the management of any distributed service.

This paper introduces the key features of **April** and illustrates their use for agent style applications. For example, we show how the contract net protocol could be implemented, and how a skills server agent might be programmed. Finally, we sketch how it might be used to implement an agent based programming language similar to the recently proposed AgentSpeak [7]. We also show how we can program migrating agents.

2 Specific features of April

2.1 Publically named processes

All processes in **April** have names – called *handles* – associated with them. Normally, the name will be automatically generated by **April** when the process is started but a programmer assigned name, such as **agent0**, or **expert1**, can given to the process as it is forked. Handles can be passed around as arguments of function and procedure calls, and in messages sent to processes. A message is sent to a process by sending it to its handle.

Programmer assigned handles, such as **agent0**, are local to the **April** invocation (the Unix process running **April**) unless they are made public. An assigned handle is made public by registering it with the name server to which the **April** invocation is linked. An **April** name server has much the same role as an Internet domain name server. Thus, at Imperial we have a local **April** name server called **nameserver@lg.doc.ic.ac.uk**. As with Internet domain name servers, this is linked with higher level name servers such as **nameserver@doc.ic.ac.uk** and **nameserver@ac.uk**. All **April** invocations running on machines on our research network link with this name server. A forked process within such an invocation can therefore have its assigned name, say **agent0**, registered with this name server. Now, processes in **April** invocations running anywhere in the world, providing their local name server is linked to **nameserver@lg.doc.ac.uk** via some hierarchy of name servers, can send a message to this process using the name **agent0@lg.doc.ic.ac.uk**. Incidentally, all the nameservers are themselves just **April** processes.

The name server system allows one to build global **April** applications. The system will recover from name server failure and restart, and from failure and restart, even on a different machine, of any user process with a registered name. It recovers in the sense that no messages will be lost.

2.2 Communication

April incorporates a simple message passing mechanism for communicating between processes. Messages, which can be arbitrarily complex symbolic structures, can be sent between processes independently of the location of the processes involved. This is possible because the handle of a process is unique at a system wide level and the system can uniquely map each process handle to a single location. The message send primitive is a send to one or more process handles. The message is put into the message buffer of each identified target process, each process having *exactly one* message receive buffer.

Processes use patterns to determine the messages that they are ‘willing to accept’ at any given point. Thus, a process can pick out from its message queue only those messages which it is able to accept at any given time. Typically it does this by entering a *message receive choice* statement, a sequence of alternative Dijkstra style guarded commands [8], the guards of which are message receive patterns and tests. This has the form:

```
{
  Ptn1 -> Act1
| Ptn2 -> Act2
.
| Ptnk -> Actk
}
```

Here, $|$ is the operator that separates the alternative guarded commands. **April** will test each message in the buffer in turn, in arrival order, to see if it matches any of the patterns Ptn_1, \dots, Ptn_k , the patterns being tried in that order. As soon as a message is found matching one of the patterns, Ptn_i , the message is removed from the buffer and the associated action Act_i is executed.

Usually the matching pattern Ptn_i will contain variables that extract values from the removed message, and the action will use these extracted values. Messages skipped over before a matching message is found are left in the buffer, perhaps to be picked up by another message receive choice statement that is executed later. (A skipped over message could even be picked up as part of the action Act_i .)

If no message is found in the buffer that matches any of the patterns the message receive choice statement suspends, causing a suspension of the process. The process is reactivated as soon as a new message is placed in its buffer. This new message is then tested against each of the alternative patterns. If it matches none of the patterns, the process again suspends. It will continue testing each new message until one is received that is accepted by the choice statement. The **April** semantics for a message receive choice statement is similar to that used in Erlang [2].

Order of arrival of two messages M1 and M2 in the message buffer of a process is not necessarily time order of transmission. **April** cannot preserve time order for messages sent from different processes, because there is no global clock or clock synchronization assumed. Order of messages sent between a given pair of processes is, however, preserved. A message sent from one process P1 to another process P2 cannot overtake any earlier messages sent from P1 to P2. This property is important for many client/server applications.

The communication primitives use the TCP/IP protocol as the underlying transport medium¹. Whilst this choice is not essential to the design of **April**, it does permit applications to interact across a wide area network such as the Internet, and also permits access to non-**April** based applications.

2.3 Environment

April is intended to be used in an heterogeneous environment, where application systems are typically constructed in a variety of languages. **April** supports this in two distinct ways: external function/procedure can be linked in to an **April** application and an **April** program can communicate using its TCP/IP protocol with other applications.

From the point of view of the rest of the **April** system, an application which has been linked using TCP/IP in this way appears as another named process. From the point of view of the application, the **April** system appears as a kind of intelligent file – the commands to send and receive messages are similar in spirit to ordinary “C” function calls to read and write to files.

The actual linking with the **April** system is done by calling one of the interface library functions that puts the name of the application into the **April** name server for the local network on which the application will run.

An interface from **April** to a Prolog system, and to DialoX, a dialogue and graphics server running on top of Motif, have been implemented in this way.

2.4 Real-time

April has aspects of a real-time programming language. That is, it has constructs in it which allow a program to be synchronized with real-time events. For example, it is possible to set processes to be activated at particular ‘wall-clock’ times (such as midnight on Jan 1st 2000). It is also possible to control the length of time spent waiting for messages by setting an appropriate time out.

However, since **April** is a symbolic language it is not always possible to predict precisely how long a given operation will take. This means that **April** is not particularly suitable for time-critical real-time applications where response times must be guaranteed with tight tolerances.

2.5 Higher order features

April has a few simple higher-order features. In particular it is possible to use a *lambda abstraction* of an expression as a first class data item denoting a function. Like other values, a lambda abstraction can be stored in a tuple, passed as an argument or returned as a value, or passed to another process in a message.

¹ In fact we use the simpler UDP/IP protocol.

In addition, there are two other related abstractions – *procedure abstractions* and *pattern abstractions*, both of which generate first class data items. The former allows the runtime construction of a new procedure, the latter the runtime construction of a new pattern. Supplied with variables of the appropriate type, a pattern abstraction can be used wherever a pattern can be used, for example as a message receive guard.

By sending a pattern abstraction, a process P1 can give another process P2 the ability to recognize new sorts of messages and to extract from these messages appropriate values.

Alternatively, P1 can use a pattern abstraction to pass to P2 the method to be used in retrieving data from a ‘data base’ internal to P2, without letting P2 know what that method is. This is because the pattern abstraction passed from P1 to P2 will not be decomposable by P2. P2 will only be able to apply the pattern abstraction. This is the key advantage of ‘higher order’ versus ‘meta order’.²

Below we shall give an example of a process that acts as a skills server. As well as ordinary inquiries for information about agents with a specific skill, it can be sent pattern abstractions that it will use to retrieve appropriate agent handles from its skills data base using enquirer specific criteria embedded in the abstraction. Of course, this means that the inquirers must know about the structure of the skills data base.

3 A simple server program

A server is a process which receives a series of requests to perform some task - to print files, to update a data base, whatever. A simple **April** program which implements a server for some task, passed to the server as a procedure argument, T, is outlined in Program 3.1.

```
server([any]{}?T)
{
  repeat{
    [do,any?arg] -> {
      T(arg);
      done >> replyto
    }
  } until quit
};
```

/* A request to perform task T */
/* report its done */
/* loop terminates on a quit */

Program 3.1: A simple server program

Typically, the program would be invoked with a process fork statement of the form:

```
server1 public server(procname)
```

where **procname** is the name of the procedure that performs the server’s task. The statement forks the server process. It also registers the name: **server1**, with the local **April** name server.

Another process within the **April** invocation that forked **server0** can send messages to it using a message send statement of the form:

```
[do,taskargument] >> handle?server1
```

Processes in other **April** invocations, use a message send:

```
[do,taskargument] >> handle?server1@domain
```

where **nameserver@domain** is the name of the local **April** name server with which the **server1** name was registered.

Within the body of the **server** procedure we have the construction:

²Originally **April** had meta order features, source code, as tuple data, could be passed and ‘eval’ executed as in Lisp and Prolog. We decided to drop this in favour of the higher order features because of its simpler implementation, cleaner semantics and its security advantage.

```
repeat{
...
} until quit
```

This is a loop of message processing that ends when the message `quit` is received by the `server1` process and no `do` messages precede the `quit` message in its message buffer, i.e when `quit` is the next message for the process.

The body of this loop consists of a single *guarded command*:

```
[do,any?arg] -> {T(arg);...}
```

A guarded command has the form:

```
guard -> action
```

In this case, the guard is a message receive pattern. The pattern specifies a pair – or a 2-tuple – where the first element of the pair must be the `do` symbol and the second element is any value.

If the match succeeds – i.e., if there *is* a `do` message that contains some data value – then that component is assigned to the variable `arg`, which is local to the guarded command. If the match did not succeed, because it was not a `do` message or no argument was given, then the variable will not be given a value.

If we had been more specific about the type of the argument that `T` must take, for example, requiring it to be an integer, then we would use:

```
[do,integer?arg]
```

to match the message and

```
server([integer]{}?T)
```

as the the header for the procedure. `[integer]{}?T` is the `April` type descriptor for a procedure with a single argument which is an integer. The previous `[any]{}?T` denotes a procedure with a single argument of unspecified type.

When the server finds a `do` message of the required form, the message will be removed from the buffer and the action part of the guarded command will be executed. This is the statement group:

```
{T(arg); done >> replyto}
```

Here, the server process invokes the procedure `T` with the argument the value that was received in the message.

After completing the `T` procedure, a reply message is sent to the *reply* process associated with the received `do` message, normally the process that sent the message, to let it know that the task has been completed. This is done using the statement:

```
done >> replyto /* report its done */
```

This sends a message – consisting of the single symbol `done` – as the reply.

The symbol `replyto` is an `April` keyword. When used inside the action part `A` of a message receive guarded command:

```
Pattern -> A
```

it identifies the reply process for the message that matched `Pattern`. This is unless `A` contains another message receive guarded command:

```
Pattern -> {
...
Pattern' -> A'
...
}
```

and `replyto` is used inside `A'`. Then it refers to the reply process for the message that matched `Pattern'`.

The `replyto` process for a message is usually the process that sent the message, but it can be different. To make it different, the sender uses a message send:

```
msg ^^reply_handle >>target
```

This makes `reply_handle` the `replyto` process for the message. `April` has another keyword, `sender`, the always denotes the actual sender of the message.

3.1 Discarding all other messages

If the server process is sent any other form of message (other than the `do` and `quit` messages) it is simply left in the message buffer and ignored.

To remove any such message from the buffer, as soon as it reaches the front of the buffer, we add another message receive guarded command to the loop body of the process. It becomes:

```
{
  [do,any?arg] -> {T(arg); done >> replyto}
| any?Other -> [unrecognized,Other] >> replyto
}
```

The body of the loop is now a *message receive choice* statement with two alternative guarded commands, the alternatives being separated by the `|`. The pattern of the second guarded command matches any pattern, so if the first message in the buffer is neither a `do` nor a `quit` message, it will be picked up by this default guarded command. It is removed from the buffer and a suitable complaint is sent in reply.

3.2 A process for each do message

One problem with Program 3.1 is that while the task `T(arg)` is being performed the server process cannot respond to any further messages it may receive.

We can solve this problem by introducing a sub-procedure – called `DoTask` – which is forked each time there is a task to perform. The definition of this procedure is contained in Program 3.2.

```
DoTask( []{}?Task)
{ Task();
  done >> creator /* report its done */
};
```

Program 3.2: A do task procedure

`DoTask` has one parameter – a no-argument procedure `Task` to execute. Notice that it sends a `done` message to `creator`. This is another `April` keyword that identifies the process that spawned the `DoTask` process. In our case, `creator` will be `server1`. The `Task` it will execute will be `T(arg)` which must be passed as a procedure abstraction. In `April` this is denoted by a `mu` expression. (There are analogous `lambda` expressions for function abstractions.)

```
mu(){T(arg)}
```

is the way we denote the no argument procedure abstraction, which when called, will execute `T(arg)` with the value of `arg` being the value it has when the procedure abstraction is created. It is used in Program 3.3, a program for a server which forks off processes for each job request it receives.

The `fork` used in the Program 3.3 is the *anonymous fork* as a function. It is anonymous because there is no program assigned name, just a system generated name returned as the value of the `fork`.

```
TaskHandles := TaskHandles <> [[replyto,fork DoTask(mu(){T(arg)}]]
```

forks a `DoTask` process and adds its handle, paired with the handle of the process that requested the task execution, to a tuple of pairs of handles held in the variable `TaskHandles`. The variable is declared to be of type tuple of handle pairs in the initialization statement:

```
[handle,handle] []?TaskHandles := [];
```

`[handle,handle]` is the type of a pair of handles and the following `[]` makes the type a tuple of such pairs.

The `server` program also has a third alternative guarded command in its message processing loop. This is to accept a `done` message sent back from one of its spawned `DoTask` processes. This message receive guarded command has the form:

```
pattern :: test -> action
```

```

server([any]{}?T)
{
  [handle,handle] []?TaskHandles := [];
  repeat{
    [do,any?arg] ->                               /* A request to do task T */
      TaskHandles := TaskHandles<>[[replyto,fork DoTask(mu()){T(arg)}]]
  | done :: [handle?Req,sender] in TaskHandles ->
    {TaskHandles := TaskHandles ^\ [Req,sender];
     done >> Req}                                 /* Send done message to requester */
  | any?Other -> [unrecognized,Other] >> replyto
  } until quit                                   /* loop terminates on a quit */
};

```

Program 3.3: A forking task server

which is the general form for such a command. (The construction `pattern :: test` can be used anywhere in `April` programs where a pattern can be used.) The message that matches `pattern` is only accepted if `test` also succeeds. In this case, the test checks that the message was sent by a `DoTask` process that has been forked by the server and which therefore has its handle recorded in the `TaskHandles` tuple. If the test succeeds, the variable `Req` is bound to the `replyto` handle associated with the message that caused the forking of that `DoTask` process.

The sender of the `done` message is identified by the keyword `sender`. On receiving any such `done` message, the `server` process deletes the handle pair from `TaskHandles`. This is achieved by the `April` statement:

```
TaskHandles := TaskHandles ^\ [handle?Req,sender]
```

`^\` being `April`'s pattern match operator for deleting all elements from a tuple that match some pattern. The message `done` is then sent to the `Req` process.

This program structure is very common in `April`. It is the structure of a very simple agent. More generally, a server agent would be able to execute several different types of task, each identified by a different sort of message. The executing task would also usually communicate with the agent that requested the task whilst it is executing the task, it would not just send a message on completion. This dialogue can take place with all the messages being routed to each task process via the server agent. Alternatively, the server agent can send a message to the requesting process giving the handle of the forked task process allowing subsequent direct communication between the client process and the forked task process. The server might also accept a message from the requester to suspend or kill off the executing task. `April` has primitives to kill, to suspend and then resume processes identified by their handles, so message receive guarded commands to process such messages are straightforward to implement.

```

[kill,handle?TaskH] :: [TaskH,sender] in TaskHandles -> {
  TaskHandles := TaskHandles ^\ [TaskH,sender];
  kill(TaskId)}

```

is a guarded command to process a kill message for a task identified by its handle `TaskH`.

4 A Second Example - Contract Net Protocol

The contract net protocol [16] is used by an agent – the contractor – when it isn't sure which other agent should be asked to perform a given task. It therefore 'asks' a set of agents to bid for the work. Each of these agents – the contractees – receives a job proposal message; and evaluates whether it is able to respond and how to bid for the contract. On receiving the various bids, the contractor agent evaluates them and selects the most appropriate.

The program in Program 4.1 is a sketch implementation of a simplified contract net protocol as seen from the point of view of the agent issuing the contract. It is simplified because we assume that the

```

task_desc ::= [symbol?id,any?task,time?expiry];           /* Record type declaration */
proposal ::= [symbol?id,any?details];                   /* Record type declaration */
reply ::= [handle?contractor,proposal?bid];            /* Record type declaration */
CNet(task_desc?job,handle[]?Bidders,{[reply] []->reply}?select)
{
  [TASK_ANNOUNCEMENT,job] >> Bidders;                  /* Invitation to tender */
  reply[]?Replies :=
    collect                                             /* collect bids */
      {repeat {[BID,proposal?P] -> elemis [replyto,P]} /* We have a bid */
       until alarm job.expiry-now};
  reply?Best := select(Replies);                       /* select best proposal */
  [reject,job.id] >> (Bidders\[Best.contractor]);
  [accept,job.id] >> Best.contractor;
                                                    /* Send accept message to contractor... */
};

```

Program 4.1: A skeleton procedure for implementing contract net

contractor waits until the bid expiry time and considers all bids received before then. We also assume that at least one bid arrives in time.

The type declarations introduce user type names, `task_desc`, `proposal` and `reply`. They are all for record types. For example, `task_desc` is a record of three fields with types `symbol`, `any` and `number`, with field selectors `id`, `task` and `expiry` respectively.

The first argument of the `Cnet` process is a description of the task for which a contract must be placed. The expiry field of this description is the ‘wall clock’ time by which all bids must have been received. The second argument of the process is a tuple of handles which are the process identifiers of the potential contractors. The contract is multi-cast to them using:

```
[contract,job] >> Bidders;                               /* Invitation to tender */
```

The last argument of the process is a function, `select`. This will take a tuple of reply records and return a single selected reply record, the contractor to whom the contract will be awarded paired with their bid proposal.

The tuple of reply records, each one comprising a bid proposal and the handle of the contractor process that sent it, is constructed using the:

```

collect                                             /* collect bids */
  {repeat {[BID,proposal?P] -> elemis [replyto,P]} /* We have a bid */
   until alarm job.expiry-now}

```

expression.

The `collect` expression is a programming language ‘equivalent’ of the mathematical *set abstraction* notation. In common mathematical parlance, an expression such as

$$\{x|P(x)\}$$

means

the set of all x ’s such that $P(x)$ is true.

In `April`, the expression

```
collect{ ... elemis x ... }
```

has a similar function. `collect` converts a statement, usually an iterative statement, into an expression. The value of `collect` expression is a tuple, the elements of this tuple are generated by the `elemis` statements executed inside the `collect`. Every time an `elemis` statement is executed within the body of the collect expression another element is added to the set. The construction of the set is completed when this statement terminates. In `April`, tuples are used to represent sets, and so we often use the terms interchangeably.

In the contract net example above, we have a `collect` iteration which constructs a tuple where the iteration to which it is applied is bounded by time. On each iteration the next element to be inserted in the tuple is denoted by the `elemis` statement executed in the loop. In this case, an element is put into the tuple whenever a `bid` reply is received. The expression `job.expiry-now` specifies how long the loop should execute. (`now` is an `April` function that evaluates to the current clock time.) The loop terminates when this time has lapsed. When the expiry time is reached, the `select` function is applied to all the collected replies to find the best reply. This is assigned to the variable `Best`.

A `reject` message is sent to all `Bidders` except the bidder `Best.contractor`. The expression:

```
(Bidders\[Best.contractor])
```

evaluates to the tuple of handles of these unsuccessful bidders. (It removes the `Best.contractor` handle from `Bidders`, `\` being `April`'s set difference operator.)

Finally, an `accept` message is sent to the bidder, identified by the contractor field of `Best`, to award the contract.

4.1 Macro defined features

In Program 4.1 the loop construct:

```
repeat C until alarm T
```

where `T` is a time and `C` is a message receive choice statement was used. This form of statement is in fact macro expanded into:

```
handle?new := fork alarm(T);
repeat C until bell :: sender=new
```

where `new` is a fresh variable name. That is, it is implemented as a source transformation into a fork of an alarm process and a loop which terminates when a `bell` message is received from that alarm process. `alarm` is a system process defined as:

```
alarm(time?T){
    delay(T);                                     /* suspend for T time */
    bell >> creator}                             /* then send a bell message to creator process */
```

Macros are used extensively in `April` to extend the syntax and functionality of the language. In fact, the

```
repeat C until M
```

loop construct is itself macro expanded into:

```
while true do {M -> break | C}
```

before being compiled. The macro that does this is given in Program 4.2.

```
#macro repeat ?C until ?M ->
    while true do{M -> break | C}
```

Program 4.2: Macro for repeat loop

In macros a `?` prefix on its own introduces a macro variable (implicitly of type `any`). The left hand side of the macro (before the `->`) gives the pattern of the program fragment to be expanded. The macro variables introduced here will be bound to parts of this program fragment. The right hand side gives the replacement program fragment which can use the values assigned to the macro variables by a successful match of the left hand side. When the `repeat` macro is used the macro variable `C` is bound to the message accept choice statement of the repeat loop and variable `M` is bound to the termination message. The macro for the `repeat.... until alarm ...` loop is only slightly more complicated.

The contract procedure of Program 4.1 will be forked each time an agent needs to place a contract. So typically, there will be several contract processes running within each agent. To invoke the process, the agent needs to have the tuple of handles of the **Bidders**, the agents to whom the contract announcement will be sent.

In a multi-agent environment, the suitable bidders may be continually changing. To cope with this, we might have a directory agent that keeps track of all the current agents and the skills that they have. The contractor agent can then query such a **SkillServer** agent to find a suitable bidder list for some contract. All he needs to know is the skill or skills that are needed for the contract.

5.1 The role

The role of a **SkillServer** agent is to match agents with skills. At its simplest level, the **SkillServer** agent receives a request in the form of a message such as:

```
[request,skill]
```

and replies with a tuple/set of agents who have the required skill.

In addition to requesting agents' handles, an agent can also inform the **SkillServer** agent that it is willing to provide services; i.e., that it skills which it can perform. Therefore, an agent interacts with the **SkillServer** server in two ways:

1. it makes requests for the identities of agents which may be willing to perform a given service, and
2. it informs the server of its own skills and services; thus allowing other agents to access it.

5.2 The implementation

The **SkillServer** server contains a database of skills and agents which have registered them. This database is used to answer requests by any agent which wishes to locate a service. The skills database is updated whenever an agent registers or deregisters. It comprises a tuple of records. Each component record is of the form:

```
[agent_handle,[skill1,...,skilln]]
```

The main structure of the **SkillServer** agent, as shown in Program 5.1, consists of a loop which waits for messages such as **request**, **register** and **deregister**.

The variable **skillsDB** contains the skills database – it is represented using a tuple/set of records with the type declaration:

```
skill_entry ::= [handle?agent,symbol[]?skills];
```

Each pair consists of the handle of an agent, which can offer one or more skills, paired with a tuple of names for these skills, which are symbols.

In Program 5.1 the statement:

```
skill_entry[]?rs:=skillsDB~/([any,symbol[]?sks]::sk in sks)
```

retrieves and assigns to variable **rs** all the **skillsDB** entries in which the requested skill **sk** appears. The operator **~/** is April's pattern match *selector* from tuples/sets. Its second argument, in this case:

```
([handle,symbol[]?sks] :: sk in sks)
```

is the element pattern, together with a test, to be used in selecting from the first argument, the tuple/set of elements. The value of the **~/** expression is all the pairs in the skills database which match the pattern and have the required skill. The test **sk in sks** is, in this case, the real selection criterion since all the records will match the pattern.

Notice also the use of the choice statement:

```
{ rs != [] -> [CanDo,rs^agent,sk] >> replyto
| true -> none >> replyto                               /* no known agent */
}
```

```

skill_entry := [handle?agent,symbol[]?skills];
SkillServer()
{
  skill_entry[]?skillsDB := []; /* The skills database is initially empty */
  repeat{
    [request,symbol?sk]->{ /* Request message received */
      skill_entry[]?rs:=
        skillsDB~/([any,symbol[]?sks]::sk in sks); /* search skills database */
      { rs != [] -> [CanDo,rs^agent,sk] >> replyto
      | true -> none >> replyto /* no known agent with the skill */
      }
    }
  | deregister -> /* deregister an agent */
    skillsDB := skillsDB ^\ [sender,any] /* remove skill record */
  | [register,symbol[]?sks] ->
    skillsDB := (skillsDB ^\ [sender,any]) <> [[sender,sks]] /* new record */
  } until quit;
}

```

Program 5.1: The basic `SkillServer`

This is a *test choice statement*, a choice statement with only test conditions and no message receive patterns in the guards. The guard `true` always succeeds, so if the guard `rs != []` of the first guarded command of the choice fails (remember the guards are tried in sequence), the action of the second guarded command will be executed. The expression `rs^agent` is tuple/set projection. It projects the set `rs` on the `agent` field producing a tuple/set of agent handles.

The operator `^` is `April`'s pattern match deletion operator. For the `deregister` message its effect is to remove any entry in `skillsDB` for the `sender` of the message. Its use for a `register` message guarantees that there is only one entry per agent.

5.3 Retrieving information with user defined patterns

Using the above skill server a `request` message can only ask about one skill even though an agent can register a tuple of skills. Perhaps we should also allow `request` messages that give a tuple of skills and only agent handles registered with all the requested skills will be returned. To do this, we could add an extra guarded command to the choice statement of the message processing loop:

```

[request,symbol[]?rsks] ->
  skill_entry[]?rs:=skillsDB~/([handle,symbol[]?sks] :: rsks subset sks)

```

The `subset` operator is `April`'s built predicate that will test that every element in the tuple `rsks` appears on the registered `sks` tuple.

Note that we can have this additional guarded command in the message processing loop even though we already have a message receive clause for a `request` message pair. This is because the `request` message accepted by the new message receive must have a different type of second component in the message pair, a tuple of symbols rather than just a single symbol. The two patterns will therefore match quite different incoming messages.

But what if we now want to find an agent that has either one pair of skills or another pair. Do we add another message receive guarded command? We could, but there is a better alternative. Rather than have a message receive for each new type of retrieval request, we can allow the request message itself to specify the retrieval test in the message. The requester can send a pattern match procedure, a pattern abstraction, in the request message.

Consider this alternative guarded command for `request` messages.

```

[request,[],()?ret_pattern] ->
  skill_entry[]?rs:=skillsDB~/ [](ret_pattern)

```

This expects the message to include a `ret_pattern` which is a type `[]()`, which is the `April` type descriptor for a pattern that just tests, i.e. which does not bind any variables outside the pattern. (The bracket pair `()` signals that it is a pattern abstraction and the preceding empty tuple `[]` tells us it exports no variable bindings.) The `ret_pattern` is applied using the expression `[](ret_pattern)`. The surrounding round brackets tell us that `ret_pattern` is a pattern abstraction and the preceding `[]` is the list of visible variables, in this case none, that it will bind.

If this is the only request message that `SkillServer` will accept, how does another agent send a request to search for a single skill, or to search for a tuple of skills.

The message that is sent to find all the agents registered with a particular skill, say the `multiply` skill, is now:

```
[request,tau()([any,symbol[]?sks]::multiply in sks)]
```

The message to find all those that have all of a tuple of skills `[sk1,...,skn]` is:

```
[request,tau()([any,symbol[]?sks]::[sk1,...,skn] subset sks)]
```

The message to find agents that have skill `sk1` and not `sk3`, is:

```
[request,tau()([any,symbol[]?sks]::sk1 in sks and not sk3 in sks)]
```

and so on. Expressions of the form `tau(variables)(pattern)` are pattern abstractions. In the above examples there are no *variables* since the pattern abstractions are just tests.

The key advantage of this approach is that the requester defines the retrieval criterion, it is not restricted to a fixed set of types of retrieval inquiry. Plus, the `SkillServer` just uses the retrieval abstraction to scan its data base. It cannot inspect the criterion, which is private to the requester.

The main disadvantage is that the requester needs to know the type of the internal `skillsDB` data base of `SkillServer`. It needs to know that the pattern abstraction it will send will be applied to a `skill_entry` record.

6 Influences and related languages

Many of the features presented in `April` are taken from other languages; in particular much is owed to Parlog [4] and its object oriented extension Polka [6], and to PCN [9], CSP [11], Guarded Commands [8], LISP [17], Prolog [5] and APL[12].

6.1 The actor paradigm

The process style of programming in `April`, and the pattern matching on messages, comes from the Parlog family; although clearly there is a close similarity with the actors concept [1]. We can quite easily simulate actor style programs in `April`. Program 6.1 is the `April` equivalent of the factorial program given in [1].

```
factorial()
{ [doit integer?n] ->
  { n=0 -> {1 >> replyto; factorial()}
  | n>0 -> {[doit n-1] ^^ fork factcust(replyto,n) >> self;
            factorial()}
  }
};
factcust(handle?customer, integer?n)
{
  integer?f -> n*f >> customer
};
```

Program 6.1: An actor style program

We have written it as a recursive program because the actor program is recursive. As with actors, the recursive call to `factorial` inherits the same message buffer as the initial call. This is true even if the call to `factorial` is replaced by a call to some other procedure. (Only if a procedure call is forked as a separate process do we get a new message buffer.) As with actors, this technique of calling a different procedure can be used to implement behaviour change of a process. However, the more normal `April` style for defining a process whose behaviour does not change, is to embed its main message processing choice statement in a loop.

The `factorial` program is a non-terminating program that consumes a stream of messages that are all non-negative integers wrapped up as `doit` messages. It replies with their factorial values. It is a factorial server. When it receives a 0, it replies with the value 1. When it receives an integer `n` greater than 0 it does not reply immediately. To compute the factorial of `n` it needs the factorial of `n-1`. To find this value, it sends itself a `[doit n-1]` message using the message send:

```
[doit n-1] ~~ fork factcust(replyto,n) >> self
```

`self` is an `April` keyword that denotes to the handle of the process in which it is used. In this case `self` is the factorial process. The construct:

```
M ~~ handle >> target
```

is a message send which makes `handle` the `replyto` address associated with the message instead of the default, `self`³. In this case it means that the answer to the `[doit n-1]` message will be sent back, not to the factorial process, but to the newly forked `factcust(replyto,n)` process. This will take the answer, multiply it by `n`, and send the answer to the `replyto` process for the original `[doit n]` message. That is all that it does.

A major difference between the `April` program and the actor program is that the latter immediately forks itself to handle more `doit` messages whilst the `April` program *must* execute the action part of its message receive guarded command *before* recursing to accept more `doit` messages. This is because the action part of the guarded command is a sequence of statements.⁴ The difference is insignificant for this example since the action part for a message containing a positive integer just forks a process and sends a message before recursing. Only if substantial sequential processing was involved would the actor program have more parallelism by doing the recursive call in parallel. Sometimes, being able to delay an actor's continuing behaviour until other actions have terminated, is what one wants to achieve. Because the basic actor concept does not include sequentialized computation, this delaying has to be achieved in a somewhat convoluted style as an actor program. `April` has a much more coarse grained computational model than actors.

Other differences are:

- An actor only accepts messages at the head of its message buffer. So the actor program for factorial will fail if it is sent a message other than a `doit` message containing a non-negative integer. The `April` program will skip over any such rogue message, leaving it in the buffer.
- An actor has just one message processing choice statement, and the action on receipt of a message *cannot* contain another message receive guarded command. In `April` the action can be specified by any `April` statement or group of statements.
- The actor paradigm does not guarantee to preserve order of messages sent from an actor A1 to an actor A2. `April` does.

6.2 ABCL/1

`April` is closer to the ABCL [15] derivative of the actor paradigm. ABCL/1 *preserves* order of messages between a given pair of processes. Like us, the authors of ABCL/1 consider this to be essential for certain

³Remember that the sender of a message can always be identified by the value of the `sender` keyword. Testing whether `sender=replyto` will test if the sender has explicitly set the `replyto` handle for the message to be a different from itself. This can be used to detect possible *passing off* – where one process attempts to mimic the messages of another process. The sender handle associated with a message cannot be changed.

⁴We could not fork the recursive `factorial` call for the forked call would get a new message buffer. In `April` there is no way of transferring ownership of a message buffer from one process to another. A process and its unique message buffer are an inseparable pair.

applications. ABCL/1 also has both a sender and a replyto identity attached to every message. (As in **April**, the replyto identity can be set by the message send in both ABCL/1 and the actors framework.) In addition, in ABCL/1, an action on a message receive can include another message receive. This auxiliary message receive can also search down the message buffer, and will cause a suspension of the process if the message has not yet been received. But as far as we understand, the main message processing loop of an ABCL/1 program can only pick off messages at the front of the message buffer. (In **April** any message receive guarded command or choice statement will search for an acceptable message in the buffer.)

On the surface, ABCL/1 appears to have more ways of sending messages. It has the *past* form, which is the same as in **April**, and the *now* and *future* forms which are not directly supported in **April**. The *past* is asynchronous message send. The *now* form is synchronous send, in which the sender suspends until a reply is received. The *future* form is asynchronous but the sender can later suspend if the reply has not yet been received and placed in some variable associated with the *future* send. However, as shown in [15], the *future* and *now* forms of message send can be implemented using just the *past* form providing the language allows the explicit setting of the `replyto` address, and it can search for messages in its message buffer. Both of these facilities are in **April**. Thus, the extra forms of message send in ABCL/1 can be emulated in **April**. Using the macro processing facility, we can easily extend the language to have extra message send operators with the semantics of the ABCL/1 *now* and *future* forms. (A macro to implement the *now* form of message send is given in [13].)

ABCL/1 also has two *modes* of message sending: normal and express. An express message sent to an ABCL/1 object/process interrupts the current processing of any normal message, suspending the processing. The action associated with the express message is then executed. After this is completed the normal message processing is usually resumed.

```

all_object(){
  handle?n_mess_proc := fork normal_object();
  while true do
    { messages() > 0 ->
      {
        express_message1 -> {suspend(n_mess_proc);...;resume(n_mess_proc)}
        .
        | timeout 0 secs -> {any?normal_mess -> normal_mess >>> n_mess_proc}
      }
    | true -> wait_for_message()
  }
};
normal_object(){
  while true do
    { normal_mess1 -> ....
      .
    }
};

```

Program 6.2: Emulating ABCL/1 express messages

In **April** we can emulate having both express and normal messages with a program structure as depicted in Program 6.2. `all_object` is the main process that handles all messages. It forks another process to handle just the normal messages. The outer choice statement of its loop waits until there is a message in the buffer. `messages()` is an **April** primitive that finds the number of current pending messages. `wait_for_message()` is a built in procedure that will suspend until a message is placed in the buffer of the process. When the message buffer is not empty the process searches the buffer looking for an express message. If this search fails, the `timeout` guarded command:

```
{timeout 0 secs -> {any?normal_mess -> normal_mess >>> normal_mess_proc}}
```

is executed. The `0 secs` timeout means that the action of this command will be executed as soon as the search of the buffer for an express message has failed. All the messages in the buffer must be normal

messages. The action part picks up the first of these and forwards it to the normal message process. >>> is **April**'s message forwarding operator. The statement:

```
normal_message >>> normal_mess_proc
```

is equivalent to:

```
normal_message ^^ replyto >> normal_mess_proc
```

so the receiver of the normal message sees the original **replyto** value attached to the message.

Using **April** macros, we can extend the language to allow the packing up of such a two procedure emulation in one procedure definition. A suitable syntactic sugar for Program 6.2 is Program 6.3. In this program express messages are separated from the normal messages and further distinguished by having their own message/action operator -->.

```
abcl object
express_messages
  { express_message1 --> {.....}
  .
  }
normal_messages
  { normal_mess1 -> ....
  .
  };
```

Program 6.3: An ABCL/1 style program

```
#macro abcl ?Name express_message ?E normal_messages ?N ->
  {(all_##Name)(){
    handle?n_message_proc := fork (normal_##Name)();
    while true do
      { messages() > 0 ->
        {E
          |
          timeout 0 -> {any?M -> M >>> n_message_proc}
        }
        | true -> wait_for_message()}
      };
    (normal_##Name)(){
      while true do N
    }
  }
};
#macro (?P --> ?A) -> {P -> {suspend(n_mess_proc);A;resume(n_mess_proc)}};
```

Program 6.4: Macros for ABCL/1 style objects

The two macros in Program 6.4 will expand Program 6.3 into Program 6.2. The first macro massages the given program into two procedure declarations. The left hand side of the macro:

```
abcl ?Name express_message ?E normal_messages ?N
```

matches the structure of Program 6.3, binding the macro variable **Name** to **object**, and the macro variables **E** and **N**, respectively, to the choice statements for the express messages and normal messages. The replacement program is constructed by evaluation of the right hand side of the macro. In this, **##** is used as an infix operator. As such, it is the **April** string concatenator. It is used to construct

two new procedure names out of the given `Name`. The macro also adds the `timeout` guarded command, which forwards the normal messages, as a last alternative to the express message choice statement `E`. The second macro expands each express message receive action to explicitly suspend and resume the normal message process. In addition, we would need suitable operator declarations making `express_messages`, `normal_messages` and `-->` into operators.

The reader may have noticed that we have glossed over the issue of the state variables of the object and what will happen if both the normal and express message actions need to manipulate the same state variables. Since, in `April`, processes cannot share variables, two processes can only share information if it is held in a third process that accepts access and update messages for its components. So, the `April` emulation of ABCL/1 objects with express message actions that share state variables with the normal message actions, has to have all these shared variables as local variables of a third process. To generate this emulation using macros, we will need to have explicit declaration of the state variables that are shared by the actions of the two kinds of message at the `abcl` syntax level. Then, every use and update of such a variable in the action associated with either a normal or express message has to be replaced by explicit message communications to the third process which holds their values.

But this handling of the state updates by a separate process also has a positive aspect. It means that we could allow concurrent execution of actions. That is, on receipt of a message we could fork a process to execute the associated action. In this framework, suspending the normal express message processing on receipt of an express message more appropriately becomes giving the process that is forked on receipt of the express message highest priority. Or equivalently, suspending the processes currently executing the tasks triggered by the normal messages until it has terminated. All the messages can then be handled by one process which keeps a data base of the processes it has forked, just as our server program of Program 3.3.

7 On Implementing a DAI platform on top of April

In this section we shall indicate some of the ways that the features of `April` could be used to implement a DAI or agent based platform similar in functionality to AgentSpeak [7]. This is a recently proposed concurrent agent based programming language designed to extend concepts of object based concurrent programming, which `April` definitely embodies.

An agent would be represented as a publically named process. This process would itself have internal parallelism. At start up it would typically spawn several non-public processes, processes that are local to the agent. One would be the main message receiving *interface* process, another would be the *knowledge handler*, a process that holds the state, or knowledge, of the agent. This knowledge would be represented as several sets of records, each set of records being the current facts for some relation used by the agent to encode its knowledge, as in AgentSpeak[7]. The knowledge needs to be held in a separate process, that accepts messages to update and access the knowledge, if we want to allow the agent to be concurrently executing several tasks or plans. Each task/plan would be executed by a forked process internal to the agent, and each of these would access or update the shared knowledge by sending messages to the knowledge handler.

The interface process would typically accept messages in a standard format for inter-agent communication, perhaps formats based on particular speech acts such as *inform*, *request* etc. The agent could offer services, and the actions linked to these services could be held in a data base associating a pattern of use of the service with a plan, defined as a procedure [7]. In `April` we can implement such a data base as a set of pairs comprising a pattern abstraction and a procedure abstraction which is held within the interface process. To retrieve a plan corresponding to an incoming *request* message, the agent would find a pattern abstraction/procedure abstraction pair within the set such that the pattern abstraction successfully applied to the incoming request. Successful application indicates that the plan represented by the paired procedure abstraction is appropriate for the request. The procedure abstraction is then applied to values extracted by the successful application of the pattern abstraction to the message, and forked as a separate process within the agent. Notice that this way of holding plans allows for the plan paired with some pattern abstraction to be updated by an *inform* message.

Agent groups, and message sending to agent groups, can be implemented by having a publically named *membership* process for each group. Agents then register with the membership processes for the groups to which they want to belong, in a manner similar to the registering of agents with the skill server described above. Now, an agent that wants to send a message to an agent group sends it to the group process. This, in turn, can multi-cast forward the message to every registered member of the group,

or to just those that indicate a willingness to accept messages of a certain form. This last feature will depend on how elaborate we want to make the registration action.

7.1 Mobile agents

There is considerable interest at the moment in the concept of mobile agents migrating from environment to environment accessing and abstracting information local to the environment before moving on.

Let us assume, for simplicity, that such an agent has a single thread of execution and so can be represented as a single process in **April**. How can we let this process migrate, and, if it does, how can it access information in a new environment.

Remember that non-public processes within an **April** invocation can be given program assigned names. So, we can have a network of publically processes, `mobile_agent_station@lg.doc.ic.ac.uk` etc., all of which are executing within **April** invocations that have forked processes with standard, but non-public names, such as `skill_server` and `file_server`. A mobile agent is then sent to a station by being sent to one of the publically named station processes, for example `mobile_agent_station@lg.doc.ic.ac.uk`, as a procedure abstraction in a message. When it arrives at a new mobile agent station, it sends out messages using the standard but locally assigned names `skill_server` etc., and, by so doing, plugs into the local environment.

To migrate, the process sends itself to another of the publically named stations wrapped up as a procedure abstraction. Whatever state information it wants to take with it must be held as arguments of the procedure that defines the agent. Program 7.1 gives one possible structure for such a procedure.

```
mobile_agent1(type1?S1,...,typek?Sk){
.
/* initial communication with the environment, sending messages
to the standardly named servers */
.
/* receive replies and get needed information */
.
/* optionally report to some process at the home station
and determine which station, NewStation, to visit next */
.
[migrating_agent, mu(){mobile_agent1(S1,..,Sk)}] >> NewStation;
/* last action of the process is to send a no argument procedure
abstraction, comprising a recursive call to itself with arguments
the current values of its state variables, to the next station */
};
```

Program 7.1: Structure of a mobile agent

To be able to accept and start running such a mobile agent the program for a `mobile_agent_station` just has to contain the message receive:

```
[migrating_agent, []]?Agent -> Mbs := [fork Agent()] <> Mbs
```

This records the handle of the forked agent for purposes of monitoring, and possible killing. For added security, we are considering having a fork which does not allow ancillary forking within the forked process. So a received mobile agent would not be able to fork processes unknown to the mobile agent station.

8 Conclusion

April is a small programming language which combines many of the features needed for implementing multi-agent systems and programming DAI applications.

April is efficiently implemented on top of C, using the standard TCP/IP protocol for message passing between processes in different **April** invocations. It gives a very fast and portable bottom layer

for building a DAI platform. Then, since we expect that many of the features of DAI platforms can be implemented as predefined processes and language extensions that are macro processed, each platform would itself be just a thin layer on top of **April**.

In our future research we intend to build several DAI platforms on top of **April**.

A full specification of **April** is given in [14]. Please contact the second author if you are interested using the language.

References

- [1] G. Agha and C. Hewitt. *Concurrent Programming using Actors*. MIT Press, 1987.
- [2] J. Armstrong, R. Virding, and M. Williams. *Concurrent Programming in Erlang*. Prentice-Hall International, 1993.
- [3] D. Chu. Ic-prolog ii: a language for implementing intelligent distributed systems. In *Proceedings of the 1992 Workshop on Cooperating Knowledge Based Systems*. Dake centre, University of Keele, UK, 1993.
- [4] K.L. Clark and S. Gregory. Parlog:parallel programming in logic. *ACM Toplas*, 8(1):1-49, 1986.
- [5] W.F. Clocksin and C.S. Mellish. *Programming in Prolog*. Springer-Verlag, 1981.
- [6] A. Davison. Polka: a parlog object oriented language. Internal report, Dept. of Computing, Imperial College, London, 1988.
- [7] D. Weerasooriya et al. Design of a concurrent agent oriented language. In M. Woodridge and N. Jennings, editors, *Pre-Proceedings of ECAI94 Workshop on Agent Theories, Architectures and Languages*, 1994.
- [8] e.W. Dijkstra. *The discipline of programming*. Prentice-Hall International, 1977.
- [9] I. Foster and S. Tuecke. Parallel programming with PCN. Internal report anl-91/32, Argonne National Laboratory, 1991.
- [10] L. Gasser and J-P. Briot. Object based concurrent programming and DAI. In N. A. Avouris and L. Gasser, editors, *Distributed Artificial Intelligence: Theory and Praxis*. 1992.
- [11] C.A. Hoare. *Communicating Sequential Processes*. Prentice-Hall International, 1985.
- [12] K. E. Iverson. *A Programming Language*. Wiley, New York, 1962.
- [13] K.L.Clark and F.G.McCabe. Distributed and object oriented symbolic programming in april. Technical report, Dept. of Computing, Imperial College, London, 1994.
- [14] F.G. McCabe. **April** - agent process interaction language. Internal report, Dept. of Computing, Imperial College, London, 1994.
- [15] E. Shibayama and A. Yonezawa. *Distributed computing in ABCL/1*. MIT Press, 1987.
- [16] Reid G. Smith. The contract net protocol: High-level communication and control in a distributed problem solver. *IEEE Transactions on Computers*, pages 1104-1113, 1980.
- [17] G. L. Steele and et. al. An overview of common lisp. In *ACM Symposium on Lisp and Functional Programming*, August 1982.
- [18] Donald Steiner, Alastair Burt, Michael Kolb, and Christelle Lerin. The conceptual framework for mail: An overview. Internal report, DFKI, 1992.