# Agents as Clonable Objects with Knowledge Base State

**Keith Clark** and **Nikolaos Skarmeas**
Imperial College
Department of Computing
London
UK
**email:** {klc.ns4}@doc.ic.ac.uk

**Frank McCabe**
Fujitsu Laboratories Ltd.
Japan
**email:** fgm@flab.fujitsu.co.jp

## Abstract

In this paper we introduce a distributed object oriented programming language and we illustrate its use for building Internet wide agent based applications. The language, April++, is a macro implemented OO extension of the distributed symbolic programming language April. The objects in April++ can have knowledge bases comprising sets of facts and recursive definitions of relations which can be queried, both locally and remotely, using Prolog style queries. April++ has one other significant feature. Using the higher order features of the underlying April, objects can be cloned, with the current state of the object encapsulated in a procedure closure. The clone can then migrate to be forked as a new active object running on a new host. This features enables us to implement mobile agents.

## Introduction

In (F.G. McCabe 1995) we introduced the distributed symbolic programming language April (**A**gent **P**rocess **I**mplementation **L**anguage) and showed how it could be used to build simple agent based applications. We also briefly illustrated the powerful macro facilities of the language, which have previously been used to build a simple delegation based OO extension of the language (K.L.Clark & McCabe 1996). April was designed at the outset, as part of the EC funded IMAGINE (Haugeneder 1994) project on multi-agent systems, as an extensible language. Like Prolog it has a extensible operator precedence syntax. Its macros are sets of recursive rewrite rules that operate on the operator parse tree of the program. Using macros we can write complex, and context sensitive, compile time source transformations of the program. With a combination of extra operators and macros one can readily implement sophisticated extensions of the language to support particular classes of applications.

Recently, the use of April has been significantly enhanced by a companion tool, AdB (Frank McCabe 1996). This is a database system system that can be used for persistent storage of any April data value.

Our goal is to use the language extension features and companion tools to incrementally extend April from a process based distributed symbolic programming language - an *agent implementation language* - into something that can be called an *agent oriented programming language*.

This paper describes the first step on that path. An *object oriented extension* of April in which:

- Object classes can be defined using single inheritance, with the usual Self and Super communication to access overriding and overridden methods respectively. Each object has shared, visible and private state components.
- Objects are active, can be given public names, and can be distributed over the Internet. They communicate transparently with each other, irrespective of location. Objects are processes.
- Objects can have new methods added as they are created and after they have been created. This allows us to dynamically modify a class defined prototype object, and it gives the objects a learning and instruction capability.
- Objects can create clones of themselves, encapsulating all their current methods and state into a closure data value. A clone communicated to another object, residing on a different host, can then be forked by the receiving object allowing replication or migration of the original object. We can also store clones in an AdB data base, giving us object persistence.
- In addition to the normal state variables recording object state, objects can also use SQL style sets of records (with field names) to encode part or all of the object state.
- Any of the extensional relations of an object's knowledge base can be optionally declared as `foreign`, `external` or `persistent`. A foreign relation is a shared or visible relation of another object that can be queried as though it was local. An external relation is one that resides on some AdB data base. A persistent relation is one that will be fetched from an AdB data base when an object is created, held as a local relation, but saved on the data base when the object (naturally) terminates, or when it is sent a `sync` message.

**Structure of the paper**

In the next section we review the key features of April.

We will then introduce the OO and KB features of April++ by giving the class definition for a yellow pages server. This is similar to the skill server April program we gave in (F.G. McCabe 1995), but is much more general with respect to the retrieval enquiries it can handle. We discuss modification of the server using traditional inheritance, to define a subclass with extra functionality, *or* by the more unusual runtime addition of extra methods and state components to an instance of the base class.

We then further illustrate the use of the language by looking at a typical application involving a mobile information gathering agent.

We finish by mentioning some current agent based projects using April and April++, and some related languages.

## Review of the April language

April is a process based distributed symbolic programming language in which process forking is cheap (a single April invocation, which is one Unix process, can readily handle 10,000 April processes). Complex symbolic messages can be passed between processes irrespective of their location. The messages can be any April data value (number, symbol, string, record of any values, list of any values, function, procedure or pattern closure).

Like an actor (Agha & Hewitt 1987), each process has exactly one incoming message queue. It reads messages from the queue by entering a choice statement of guarded commands. Each guarded command is of the form:

*message_pattern* :: *test* -> *action*

In turn, each message in the message queue is tested against the message patterns of each guarded command of the choice statement. (In an actor, only the first message is tested.) If a message matches the *message pattern*, and the optional *test* is true (this can test values in the pattern and/or values of state variables of the process), the message is removed from the queue and the *action* is executed. Typically, this updates state variables of the process, sends messages to other processes, or forks new processes.

If the end of the message queue is reached with no message pattern/test satisfied, the process suspends until the next message arrives. How long the process suspends can be limited by giving a special timeout guarded command at the end of the choice statement. (The April message processing semantics is essentially the same as that of Erlang (Armstrong, Virding, & Williams 1993)).

A process that just iterates over a single choice statement, comprising a guarded command for each type a message it can handle, is a natural representation of an active object.

An April invocation (or Unix process) and all April processes forked within that invocation are time shared by that April invocation. They share a single heap for data values. Communication between processes within an April invocation is very fast, it is just a copy of the heap address of the message from the sender process to the message queue of the recipient process. Communication between processes in different invocations of April, whether on the same machine or anywhere on the Internet, is handled by encoding the message as a byte stream and sending it to the target process via special April communication servers. There is one such communication server for each Internet host machine. The message is sent using the TCP/IP protocol. The byte encoding of the message, and the use of the communication servers, is transparent to the April application. In the April program the same message send operation is used irrespective of the location of the recipient.

## An introduction to April++

An April++ class definition has the following general syntax:

```
class <class name> [isa <Super name>]
  shared{ ... }     /* decl. of shared state variables */
  visible{ ... }    /* declaration of visible variables */
  private{ ... }    /* declaration of private variables */
  shared_db{ ... }  /* declaration of shared relations */
  visible db{ ... }          /* ... visible relations */
  private_db{ ... }          /* ... private relations */
  methods{
    message_pattern ::  optional test --> action
  |
  :
  } initial_actions{
  /* Actions to be performed on creation */ }
}
```

Nearly every division is optional.

When an instance of the class is created, initial values for some or all of the state variables can be given in the create statement. These can override any default initial values. The object can be publically created, and given a public string name, or anonymously created with a system generated identity. If publically created, other objects can send it messages using the public name. Otherwise its identity, or **handle** in April++ parlance, must be passed to any other object that needs to send it a message.

After the instance is created, it will immediately perform any initial actions defined in the class. It then enters a loop waiting for messages invoking its instance methods. These methods are given in the methods section as a choice statement of guarded commands, the only difference from April syntax is that we use --> instead of ->.

## A yellow pages directory server

Below we give the class definition of a **yellow_pages** server object. Other objects/agents can register with such a server, giving a list of skill names using a skills ontology common to all the agents. They can also query its visible data base to find the identities of agents with a particular skill.

```
class yellow_pages isa db_object
private_db {
  has_skills
    schema (handle?agent,symbol[]?skills)
    initial []}
visible_db {
  recommend_for(symbol?Sk) is {
    all of (handle?H) where
      H=skills from has_skills:agent==H and
        Sk in Sks }
    /* declare a set valued function that is a */
    /* 'view' onto the private data base relation */
  }
  methods {
    (register,symbol[]?sks) -->
      assert(has_skills(replyto,sks))>>Self
  | deregister -->
      retractall(has_skills(replyto,?))>>Self
  };
```

The private relation **has_skills** is declared to be a set of records each one of which comprises a handle - an object identity - paired with a list of symbols (the type **symbol[]**). **agent** and **skills** are field names that can be used, as in SQL, to access or express conditions on the fields of these records in queries. They are so used in the definition of the 'view' function **recommend**.

The **register** and **deregister** actions are Prolog style assert and retractall side effects to the **has_skills** data base relation of the object. These **Self** invocations of **assert** and **retractall** actions will invoke methods inherited from the **db_object** class. This is system class of April++.

We should explain the use of the **replyto** as an argument of the asserted and retracted facts. Every April (hence April++) message has an outer wrapper which contains at least two extra fields. One of these is the **replyto** field, the value of which is extracted by using the **replyto** keyword. The other is the **sender** field, which *cannot* be altered by the application program, denotes the identity of the object/process from which the message was sent. By default, the **replyto** field will be set by the communications system to the identity of the sender process, but it can be explicitly set by the application program.

## Creating a named instance of the class

To create a publically named **yellow_pages** object, with the name **"skill_directory"**, some other object, inside the action part of one of its methods, can execute:

```
"skill_directory" public_instance_of yellow pages
```

The creating object, and any other object running on the same host, can communicate with the new object by a message send of the form:

```
message >> handle??"skill_directory"
```

The **handle??** is a type cast that tells the April compiler that the string **"skill_directory"** is the public name of an object created on this host.[1] The above type cast will cause the string **"skill_directory"** to be converted into the appropriate handle at run time.

Any other April++ object running on a different host, anywhere on the Internet, can send a message to the new object with a message send of the form:

```
message >> handle??"skill_directory@machine_name"
```

where **machine_name** is the Internet name of the host machine on which the **"skill_directory"** object is running.

Note that we could create publicly named instances of the **yellow_pages** class, and give them the same public name, **"skills_directory"**, on any number of hosts. A message send to **handle??"skill_directory"** always gets routed to the locally created object with that name.

## Querying the directory server

Another object, on the same host, could access the data base of the server using a query expression such as:

```
{all of handle?A where recommend_for(
  cheap_airfares)(A) } !  handle??"skill_directory"
```

This query expression is an an implicit message send/ get reply that blocks until the answer is returned, the answer becoming the value of the query expression.

Alternatively, the querier can send an explicit labelled query message, and then at any future time enter a choice statement to pick up the reply. The querier uses a query send/ pick up reply sequence such as:

```
{all of handle?A where recommend_for(
  cheap_airfares)(A)} >> handle??"skill_directory";
⋮                                    /* other actions */
  {(answer,q123,handle[]?Ags) ::
    replyto==handle??"skill_directory" --> ...}
```

## More complex queries

Queries with multiple conditions, such as:

```
{all of handle?A where
  recommend_for(cheap_airfares)(A)} and
  recommend_for(bespoke_holidays)(A)} and
  not recommend_for(package_holidays)(A)}
```

can also be sent.

More generally, the query we send can have conditions in it that use visible relations of any other object. For example:

[1] Object handles are not actually strings, handles do not have any literal values. They can only be generated by use of special system functions, such as **creator()** which we shall use later, by use of keywords such as **replyto**, and by the **handle??** type caste applied to a string public name.

```
{all of handle?A where
  recommend_for(cheap_airfares)(A) and
  {abta_registered(A) ! handle??"abta_agent@..."}
  } ! ...
```

which includes a subquery to another publically named object "abta_agent", on some other host, to make sure that only the handles of agents that are ABTA registered[2] are returned as answers to the query. Typically the abta_registered relation will itself be an external relation held on some AdB data base on the remote host.

No extra functionality is required in the yellow_pages program to handle this more complex query. The complexity comes in the macro expansion of the query expression into a set returning function closure that will include code for the remote query. A yellow_pages object just calls this query function, passing in a pointer to its data base as an argument, to evaluate the query on behalf of the client object. It does not even know about the romote subquery.

A client can also ask for just N answers, where N is some integer using a query form N of ... where .... If it is not sure how many answers will be needed, the answers can be requested one at a time. For example:

```
handle?QS:= {
  stream of handle?A where
    recommend_for(cheap_airfares)(A) and
    abta_registered(A) ! ...
  } ! handle??"skill_directory" ;
  :
handle?A := next(QS); ...
```

will bind QS to the handle of a special answer stream process that will be forked by the "skill directory" server on receipt of the stream of ... query. This process will find the first answer to the query and then suspend. It returns the first answer to the client when the client executes a next(QS) call on the answer stream handle. It also finds the next answer in anticipation of the next demand. The next call will return the value void when there are no more answers. An any stage the client can terminate the answer stream process with a kill(QS) action. This gives objects the functionality of the KQML stream performatives.

## Importing relations from other objects

In the above example, with the remote subquery, the querier has to know which object holds the definition of the auxiliary relation abta_registered. As an alternative, we could declare that relation, and any other skill certifying relations that a client might want to use, as visible relations of the yellow_pages class *imported* from the objects that maintain them. We just add declaration such as:

```
relation abta_registered
  schema (handle?agent)
  of_type (foreign, handle?? "abta_agent@...")
```

to the visible_db section of the class definition. Now a client object just sends the query:

```
{all of handle?A where
  recommend_for(cheap_airfares)(A)
  and abta_registered(A)}
```

treating abta_registered as a though it were a local relation of the "skill_directory" server. The appropriate subqueries will automatically be sent by the server to the remote object holding the abta_registered relation.

## Modification on and after creation

April++ gives us all the normal OO facilities for describing inheritance between classes: but in addition, it allows us to dynamically add new relation state components, and to dynamically add new methods, to a created instance of a class.

In April++ we can give an existing object extra functionality by dynamically adding extra methods and data base relations when and afterr it has been created.

Every class has automatically added to it an extra list state variable, dynamic_methods, default initialized to the empty list []. Every class also has an an extra method added which is tried before all the class defined methods. This checks if there is any method in the dynamic methods list that applies to the next message. This means that a dynamically added method can override a class defined method.

Dynamic methods can be added to an object when it is created by giving a value for the dynamic_methods variable. Alternatively, if object's class inherits directly or indirectly from the system class, dynamic_object, then it inherits add method and delete_method 'meta' methods that allow it to update its dynamic_methods state variable after creation.

Each dynamic methods has two components: a symbol name, that can be used in a delete_method message to remove the method, and a *method* functions. The function takes as arguments a message M and the current state MyState of the object. It returns a pair (true,NewState), where NewSate is a posible new value for the state of the object if the method can accept the message in the current object state. It returns (false,MyState) if not.

This method function can send messages, update state variables etc. It can do anything that a normal method given in a class definition can do. April++ even allows essentially the same syntax to be used for specifying the method function of a dynamic method, as is used for defining a static method in a class definition. April++ will macro expand an expression of the form:

*ptn* :: *test* \-> *action*

into the appropriate method function.

In the mobile agent application we shall see how an application program can make use of these higher

[2] ABTA is the UK travel agents association.

order features of April++, particular the macro expanded syntax for specifying the method function of a dynamic method, to extend the notion of a single dynamic methods to a set of dynamic methods indexed by a set of conversation states. This indexed set will specify a conversation protocol, as described in (Mihai Barbuceanu & Mark J. Fox 1995a).

## A mobile agent application

We now look at using the facilities of April++ to implement a typical mobile agent application illustrated in figure 1. It involves an Internet wandering information gathering agent.

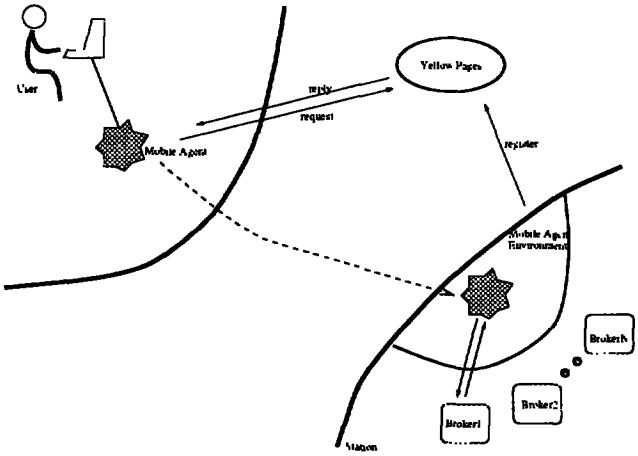


Figure 1: Mobile Agent Scenario

A user, linked to their personal secretary agent, specifies the general topic of the information gathering task for the mobile agent, such as `discount_air_fares`, and gives it a conversation protocol that the mobile agent is to use. This conversation protocol is the mobile agent's user supplied *script*. The protocol will be used by the mobile agent to engage in conversations with agents at sites it will visit. The user also gives the agent a symbolic name. The agent will use that name when it finally reports to the user, via the user's personal agent.

As part of its initial actions, the mobile agent will consult an Internet wide yellow pages server, the identity of which is held as the default value of its `Directory` provate state variable, to find which electronic shopping malls it should visit for the topic it has been given. The yellow pages server will actually return a list of handles of station agents for these malls. We shall assume that a station agent can accept a mobile agent sent to it in a message, and that they also act as local directory agents for their mall. Information broker agents for each mall will register with their local station agent. This will ensure that each topic covered by a local broker gets reflected in an entry for it in the internet wide directory. As a last initial action, the mobile agent will send itself to the first station agent.

At each site, it will begin its conversation by first sending a query to the station agent which will forward it to a suitable broker. Thereafter the agent will continue its conversation with the broker. When it has concluded the its conversation with that broker, it will move to the next station. When it has visited all stations it reports to the user's personal agent.

Space does not allow us to give the program for the personal agent. We will concentrate on the programs for the mobile agent, and the station agent.

### The mobile agent class

A mobile agent must have a negotiation ability, which it inherits from a negotiation object. A negotiation object essentially has the ability to follow the rules of any conversation protocol it is given, whether or not it is mobile. The information brokers would also be negotiators.

```
class mobile_agent isa negotiator
private {
  symbol?MyName; handle?originator;
  handle[]?Stations;
  handle?Directory := handle??"Yellow Pages@...";
  symbol?topic;
}
visible_db {
  replies schema (handle?station, any?answer);
}
methods {
  move_on -->{
    if Stations =!= [] then {
      [?next_station,..?Rest] = Stations;
      Stations := Rest;
      goto next_station;       /* move to next stn */
      conversation_state := start;
      start_conversation >> Super
    } else
      (finished,MyName,
        {all of ?Ans where replies(?,Ans)}!Self
      ) >> originator;
  }
} initial_actions { initialize >> Super;
  originator:=creator();
  Stations := {all of handle?A where
    recommend_for(topic)(A)}!Directory
  });
  move_on >> Self
};
```

In the `move_on` method of the object is the sequence:

```
goto next_station;
conversation_state := start;
start_conversation >> Super
```

This is actually macroed into:

```
(moving_on,MyName,next_station) >> creator();
(mobile_object,MyName,Closure)>>next_station;
die;
```

`Closure` is a procedure closure that encapsulates the current state of the object and code. The code, when

forked by the receiving object, will execute the two actions:

```
conversation_state := start;
start_conversation >> Super
```

and then enter the normal message processing loop of the object. At this point the state of the object will be that which it had before the `goto` further updated by these two actions. This is how the agent moves. It is similar in functionality to the Telescript (James E. White 1994) `goto`. The only aspect of the state of the object that is not transferred is its message queue. The assumption is that a `goto` is executed only when any pending messages can be disregarded.

As we shall see later, a station agent has a method:

```
(mobile_object,symbol?object_name,closure?A)-->{
  handle?H := fork A;
  assert(active_mobile_object(object_name, H));
};
```

for accepting such a message. The type `closure` is a special type of April++, denoting a no argument procedure closure. This `mobile_object` method forks the clone as a local process, using the April `fork` primitive, and then asserts a record into a local `active mobile object` relation which records the names and handles of its currently active mobile objects.

The effect of the dispatch of the closure to a station agent, the fork of this as a new active object/process at the destination, and the immediate termination of the active object that dispatched the closure. All the current values of the state components of the object, including the cached replies it has received as a result of its conversations to date, will be transfered as part of the closure.

Before the mobile agent moves on, as a courtesy it informs its (`creator()` that it is moving with a message send:

```
(moving_on, MyName,next_station) >> creator()
```

`creator()` is an April primitive function that will return the handle of the object that forked the agent. Initially this will be the personal agent of the user which really did create the object. Thereafter, it will be the the station agent which last forked it, i.e. the station agent of the electronic mall where it is currently located. As we shall see, the station agent will delete it from its active agents relation and record where it has moved to in another relation. This will allow the originator, the user's poersonal agent, to track it down if need be. Alternatively, whenever it moves, the agent could send a `moving_on` message to its originator, to keep it posted.

If the agent did not die after dispatching the closure, it would have genuinely cloned itself onto the new host. It can do this using the action:

```
copy_me_to D;
```

Such a clone will continue with its own independent life sharing only the history of the original object up to the point of cloning. We can use this technique to implement parallel search by a mobile object. The clones can still talk to one another, allowing for co-operative search as described in (Oates, Prasad, & Victor Lesser 1996).

The mobile agent has other methods that it inherits from the negotiator class, and all the dynamic methods of the conversation protocol held as a relation state component. These dynamic methods will assert facts into the private `replies` relation of the mobile agent. The conversation rules will also determine when it is time for the agent to move to the next site. They will do this sending a `move_on` message to `Self`. When the agent gets this message, and there are no more sites to visit (its `Stations` state variable has value []), it will send all its cached answers to its originator, the user agent that created it. It will then wait for any auxiliary queries to its visible `replies` relation that the originator might like to send it, for the originator will now have its current handle `H`, obtained as the value of the `replyto` for the `answers` message it received. Finally, the agent can be killed by the originator with a `kill(H)` action.

When a mobile station is created, it executes the initial actions that are specified in the `initial_actions` section of the class definition. Its first actions are to invoke its `Super initialize` method and to save the identity of its creator (the user secretary agent) in its `originator` state variable. It then queries the Internet yellow pages server, the public name of which it holds as the default value of its state variable `Directory`. This gives it the handles of all the station agents that it should visit. Its last initial action is to tell itself to `move_on`.

## The Negotiator

Conversations (Mihai Barbuceanu & Mark J. Fox 1995b) are essentially state indexed sets of dynamic methods for an object/agent. For each conversation there is a set of conversation state labels, and for each state label there is a set of dynamic methods that can used when the conversation is in that state. When a message is received, the dynamic methods for the current state are tested one at a time to see if the message test function successfully applies to the message and the current complete state of the object (not just the conversation state). If it does, the action function of the dynamic method is applied to the message and object state. Finally, the conversation moves to a new specified state. The action function can send messages, and update any other other state variable of the object.

We can represent the rules of a conversation as records comprising five fields. A symbol acting as an identifying label of the rule, a symbol acting as the label of the conversation state in which the rule can be used, a dynamic message test function, a dynamic method action function (the two componets of an April++ dynamic method which we described ear-

lier), and a fifth field which is the label of the next conversation state. These are the of records the public relation **neg_table** of the negotiator class.

```
class negotiator isa db_object
shared {
  symbol?conv_state := start;
}
private_db {
  relation neg_table
    schema (symbol?id,symbol?current_state,
      method_function?method,
      symbol?next_state);}
methods {
  any?M ::
    (method_function?Method,symbol?NS) satisfies
    neg_table(?,conv_state,Method,NS)!Self
    and (true,MyStateType?NewState)=
      Method(M,MyState) -->{
    MyState := NewState;
    conv_state := NS}
  }
```

The current conversation state of the agent is stored in the state variable **conv_state** which defaults to **start** on object creation. (The mobile agent also always resets when it moves.)

The single method of a negotiator is a 'meta-method' for trying to apply the methods for the current state of the conversation to any received message **M**.

When the message arrives, a **satisfies** query is used to find a method function **Method**,and a next state label **NS**, in a conversation rule of the current conversation state, such that the application of the function **Method** to the messaage and the current state of the object (the call **Method(M,MyState)**) returns a pair **(true,MyStateType?NewState)**. **MyState** is a macro expanded reserved identifier of April++. It will be replaced by the tuple of names of all the implementation level state variables of the object[3]. This state tuple has the April++ pre-defined type, **MyStateType**. If this test succeeds, the second returned value, **NewState**, is used to update the implementation level object state variables. Finally, the conversation state is updated.

## A conversation rule

Assume that the user wants the mobile agent to get information about a business class airplane ticket to Athens, which costs no more than 300.

The following schematic conversation rule, for initiating a conversation that might go on to haggle about the fare, might be used.

```
( start,  /* so, to be used in conversation start state */
  {start_conversation \-> {
  [ask_one, (about, topic:Self),
    (content,
      (best_business_fare, {all of ?Price where
        Price = fare from customer_interface:{
          class = business and
          destination = Athens and fare < 300
        }})),
    (language, April),
  ] >> creator();
              /* message send to host station agent */
},
wait_reply                /* next conversation state */
)
```

When a **start_conversation** message is received, the rule requires that the negotiator send a query message to the station agent (its creator). The expectation is that this will be forwarded to the appropriate broker. The message format is KQML in April syntax. Notice that the query message will include the value of state variable of the mobile object, accessed using **topic:Self**.

## The Stations

Stations are the entities that receive mobile agents and foreard their opening conversation message to an appropriate broker. A station offers the environment for mobile agent execution, and house keeping for the brokers that register with it.

```
class station
private {
  (symbol,handle)[]?Brokers := [];
  string?Directory := "directory_server@...";
}
private_db{
  active_mobile_objects
    schema (symbol?name, handle?address);
  moved_to
    schema (symbol?name, handle?next_station);
}
methods {
  (new_broker, symbol?topic) --> {
    if not (topic,?)  in Brokers then
      (register,topic) >> handle??Directory;
    Brokers := [(topic,replyto),..Brokers]
  }
| (mobile_object,symbol?object_name,closure?A) --> {
    handle?H := fork A;
    assert(active_mobile_object(object_name,H))!Self
  }
| (moving_on,symbol?name,handle?destination) --> {
    retract(active_mobile_object(name,replyto))!Self;
    assert(moved_to(name,destination))!Self
  }
| [ask_one, ?R] ::  { (topic, ?T) in R
      and (T,handle?B) in Brokers} -->
    M >>> B                  /* forward request to B */
| any?M --> cannot_handle_message >> replyto
};
```

The brokers that the station is supporting are stored in the **Brokers** state variable. When a broker wants to be included in the station, it sends a **(new_broker**

[3] These are variables names, such as **dynamic_methods**, that are system added during macro expansion and which are the same for every object. They are actually the names of the argument variables of the process that implements an April++ object at the April level. The object state variables of a class definition are not directly reflected as implementation level state variables.

message to the station with a topic name. We shall assume that all station agents are created with the same local public name "station_agent". If this is a new topic, the station agent sends a register message to the Internet wide directory server so that it will be visited in the future by mobile agents with this topic. The topic name and the brokers handle are then added to the list of brokers (`Brokers := [(name,replyto),..Brokers]`).

The station class also has two relation tables, one for keeping track of the active agents (`active_mobile_objects`), and the other for keeping track of where agents moved have gone next (`moved_to`).

Finally, the last two methods are for any messages that the station receives from its mobile agents. When a mobile agent is re_activated, it sends the first message of its conversation to the station. The station finds a broker who can deal with that topic forwards the message. Thereafter the broker and the mobile agent will continue the conversation.

If the station agent knows of no broker for the topic, or the message has the wrong format, the message (`cannot_handle_message`) is sent as a reply.

## Concluding remarks

The KNOS language/system (Tsichritzis & et. al. 1987) is an OO symbolic language with dynamic methods and object migration via object cloning. However it does not have the equivalent of knowledge base state with remote querying of object knowledge bases. Both Orient84/K (Tokoro & Ishikawa 1984) and DK-Parlog++ (K.L.Clark, Skarmeas, & Wang 1995) have objects with normal methods and modifiable local knowledge bases, but neither language supports dynamic addition of mehtods or object cloning.

April and April++ are curently being used for several agent based applications (Nikolaos Skarmeas 1996),(Wada et. al 1996). A major use is enterprise modeling, recasting in April++ the approach we previously adopted using DK-Parlog++ (K.L.Clark, Skarmeas, & Wang 1995).

We are continuing to extend the language to enrich its agent implementation capabilities. Current efforts involve giving better syntactic support for KQML style messages and further exending April++ to an agent language similar to AgentSpeak (Weerasooriya et. al. 1995).

## References


Agha, G., and Hewitt, C. 1987. Concurrent programming using actors. In Yonezawa, A., and Tokoro, M., eds., *Object Oriented Concurrent Programming*. MIT Press.

Armstrong, J.; Virding, R.; and Williams, M. 1993. *Concurrent Programming in Erlang*. Prentice-Hall International.

F.G. McCabe, K. L. C. 1995. April - agent process interaction language. In N. Jennings, M. W., ed., *Intelligent Agents, LNAI, vol 890*. springer LNAI.

Frank McCabe. 1996. AdB reference manual. Technical report, Fujitsu Laboratories Ltd., Japan.

Haugeneder, H., ed. 1994. *IMAGINE Final Report*. Munich: Siemens AG.

James E. White. 1994. Telescript Technology: the Foundation for the Electronic Marketplace. Technical report, General Magic, Inc., 2465 Latham Street, Mountain View, CA 94040.

K.L.Clark, and McCabe, F. 1996. Distributed and object oriented symbolic programming in april. In Briot, J.-P.; Geib, J.-M.; and Yonezawa, A., eds., *Object-Based Parallel and Distributed Computation*. Lecture Notes in Computer Science (LNCS). Springer-Verlag, to appear.

K.L.Clark; Skarmeas, N.; and Wang, T. 1995. Distributed object oriented logic programming as a tool for enterprise modelling. In *Proceedings of IFIP TC5 Working Conference on Modelling and Methodologies for Enterprise Integration*.

Mihai Barbuceanu, and Mark J. Fox. 1995a. COOL: A Language for Describing Coordination in Multi-Agent Systems. *First-International Conference on Multi-Agent Systems, (ICMAS95)* 17 24.

Mihai Barbuceanu, and Mark J. Fox. 1995b. The Architecture of An Agent Building Shell. In Wooldridge, M.; Jorg P. Muller; and Millind Tambe, eds., *Intelligent Agents II : Agent Theories, Architectures, and Languages*. Springer Verlag. 235 250.

Nikolaos Skarmeas. 1996. A Role Based Organizational Framework. Technical report, Imperial College, London, UK.

Oates, T.; Prasad, M.; and Victor Lesser. 1996. Cooperative Information Gathering: A Distributed Problem Solving Approach. Technical report, Umass Computer Science Techical Report, 94-66.

Tokoro, M., and Ishikawa, Y. 1984. Orient/K: A Laangnage with Multiple Paradigms in the Object Framework. *Proceedings of the International Conference on the Fifth Generation Computer System*.

Tsichritzis, D., and et. al. 1987. Knos: Knowledge acquisition, dissemination and manipulation of objects. *ACM Transactions on Office Information Systems* 5(1):96 112.

Wada et. al. J. 1996. An Agent Oriented Schedule Management System Intellidiary. *The Practical Application of Intelligent Agents and Multi-Agent Technology, PAAM96*.

Weerasooriya et. al., D. 1995. Design of a Concurrent Agent-Oriented Language. In Wooldridge, M., and Jennings, N., eds., *Intelligent Agents*. Springer-Verlag.