Content based routing as the basis for intra-agent communication

Nikolaos Skarmeas and Keith L. Clark

Department of Computing
Imperial College
London
{ns4,klc}@doc.ic.ac.uk

Abstract. In this paper, an agent architecture is proposed that can be used to integrate pre-existing components that provide the domain dependent agent functionality. The key integrating feature of the agent is an active message board that is used for inter-component, hence intra-agent communication. The board is active because it automatically forwards messages to components, they do not have to poll the message board. It does this on the basis of message pattern functions that components place on the board using advertisement messages. These functions can contain component provided semantic tests on the content of the message, they can also communicate with any other component whilst they are being applied. In addition an agent management toolkit, called ALFA, is described which offers a set of agent management services. This toolkit consists of a number of servers for storing the code of the components and symbolic descriptions of what agents regarding their component makeup. A third server uses all this information to facilitate launching new agents.

1 Introduction

With the advances in Object Oriented technology, the construction of off-the shelf components that can be used and re-used for the construction of large software systems has become fashionable. The re-use can be employed either at the design level which involves abstract patterns([2]), or at the more concrete level that involves components implemented in some specific language. This leads to a new style of software construction called *component oriented* ([15]).

In parallel, especially for distributed applications, another style of programming is often used, called *agent oriented*. In this style of programming the entities of the application are viewed as agents which are capable of accomplishing complex tasks.

A number of software architectures have been proposed for building agents. A common approach, for the internal architecture of an agent, is the separation of the agent functionalities into two main categories: the domain independent and the domain dependent one. The *domain dependent* part of the agent deals with the (possibly local) problem solving activities of the agent. The *domain*

independent part deals with the communication oriented activities of the agent and other features such as knowledge base management.

The domain dependent part of the agent usually consists of a number of quasi-independent modules, different agents comprising different collections of modules. One way of designing and implementing the domain dependent part is to hardwire into the components the inter-component, hence the intra-agent communication. However, this has the disadvantage of creating inflexible architectures. Deletions and additions of components affect the overall agent functioning and usually result in the need to re-engineer some of the other components. Therefore, what is needed is a way of allowing the components to communicate with each other which allows dynamic deletion, addition and modification of components. A standard approach is the use of a message board, a shared repository of messages which components can use as their communication bus. The advantage of a message board is that the components can be heterogeneous -implemented in different programming languages - and can be written independently. When we code one component C all that we need to assume is a standard language for inter-component communication. When C needs a service S from another component all we need do is program C to place a message requesting S on the message board. We do not need to know the identity of the other component that provides service S. Indeed there may be several, and it may be useful to have them all respond to the service request. Moreover, between requests for the service S some or all of the components offering S may be replaced, allowing dynamic upgrading and reconfiguration of the agent.

A Corba ORB offers a limited message board facility for components for it offers message routing and implementation independence for object components. But its message routing requires a destination address. The destination address can be found by use of an ORB trader or yellow pages directory object, but a more high level message board would automatically route the message to the appropriate component, based on the content of the message.

The rest of this paper explains in detail a proposal for an active message board architecture that can be used to integrate components based on content's routing of messages to appropriate destination components. The message board acts as an intelligent router because it actively forwards messages to one or more of the other agent components based on the content of the message. It does not wait to be polled by the components. As with a KQML match maker, the message board finds out which component or components to send the message to using message pattern advertisements sent to the message board by the components. However, a crucial difference between our proposed message board and a KQML match maker is that the advertised patterns are not themselves KQML messages, they are not even symbolic expressions. Instead they are active patterns - test functions to be applied to each message sent to the message board. If the function succeeds, the message board knows that the component that advertised the active pattern is interested in receiving the message.

The major advantage of sending message patterns as functions is that we can incorporate semantic tests on the content of the message. The semantic tests are

supplied by the advertiser, either as additional code included with the message test, or on demand. It can be provided on demand because in our implementation, using the April programming language ([14]), the active message pattern can communicate with the advertiser to get extra information regarding the acceptability of the message when the test is applied. In effect, each advertiser supplies the semantic information to the message board, but without the need for the message board to be able to understand this information. The message board simply executes the test function. The inability of a KQML style matchmaker to support application specific semantic tests on behalf of advertisers is a recognised shortcoming ([10]).

Each message pattern advertised by each component, C, is applied by the message board to each new message placed on the board. Whether or not the message is forwarded to C now depends on qualifications, concerning the destinations, specified by the sender. For example, the sender can specify one or all destinations (KQML `broker-one` or `broker-all`). In addition, it can attach an optional filter function to be applied to the list of identities of all the potential destinations to prune the set. This destination filter function can embed client semantics regarding appropriate destinations. For example, the filter function can directly query each potential destination, to ask them meta queries about quality of service, current task load etc. Only destinations offering the right quality of service will be left in the pruned destination set. All of these get sent the message.

This bilateral control of the message routing, by both senders and receivers, provides a powerful mechanism for processing and routing messages, more powerful than the contents routing of a KQML match maker and much more powerful than Linda style tuple spaces, or blackboards, which are passive message repositories that must be continually polled by processes for messages of interest.

The message board does not need to be used for every communication between components, but it is good policy to use it for each communication that opens a transaction. Subsequent communications for that transaction can be direct inter-component communications.

As we mentioned above, our message board is implemented using the April language but its functionality could be realised in any language offering code mobility and communication or call back facilities, for example, Java. The message board is part of an April based agent toolkit that we have used to build two agent based applications [17],[1]. This toolkit, contains also an agent management layer. This layer consists of a collection of servers that offer a set of management services for configuring a network of agents based on our component architecture.

Section 2 gives more detail of our component based agent architecture. Section 3 describes an example agent. Section 4 gives details of the implementation in the April language. Section 5 describes the management layer.

2 The agent architecture

Agent architectures range from quite abstract proposals of what agents should comprise ([5]) to more concrete artifacts designed for specific kinds of applications (viz. [6], [11]). A common approach (nicely summarised in [12] p115 and [7] p86), is an agent comprising a set of interrelated components whose functionality contributes to the overall agent behaviour. Each component can have its own private knowledge base but it is often useful to have an agent wide knowledge base that the components share. The *knowledge base* component is a domain independent component, although its contents will be domain dependent. It is also useful to have two other domain independent components. An agent *head*, or *communicator*, which communicates with other external agents or non-agent applications, and a *meta-component* that allows for dynamic reconfiguration of the domain dependent components. This architecture, which is more fully described in [16] is depicted in Figure 1 and explained below. All the components are concurrently executing components.

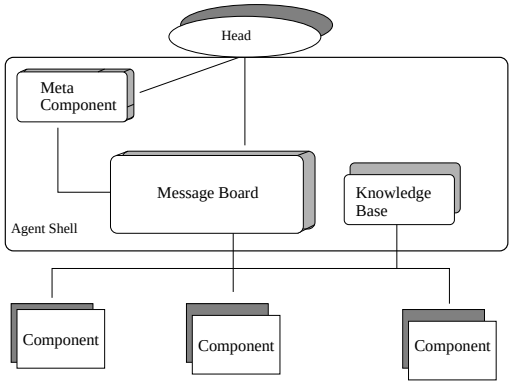

Fig. 1. The agent architecture

2.1 The knowledge base

The knowledge base component keeps information shared by all the other components. It is the global memory for all the components of the agent (each of which may also have private memory) and can be used to store information such as the beliefs, intentions and plans of the agent as well as meta-level information about other agents and the capabilities of the behavioural (the domain dependent) components. This knowledge base can be accessed and updated by all the other agent components. For the current implementation of the knowledge base, a deductive extension to April, called AprilQ ([4]), is used that provides a high level syntax for retrieving and updating information.

2.2 Behavioural components

The specific behaviour that the agent exhibits is implemented by a number of behavioural (domain dependent) components. The behavioural components will generally differ from agent to agent. A behavioural component can be composite, indeed it can itself be another agent. So agents can have a recursive structure. These behavioural components are changeable over the lifetime of the agent. This allows the agent to be reconfigured and gives it the ability to adapt to new requirements that its environment imposes. The manipulation of the behavioural components is the purpose and role of the *meta-component*.

2.3 Message board

The agent components interact with each other in two ways. They store and retrieve information from the shared knowledge base. They also interact via messages. The message interaction is supported by the active message board. Any agent component can place messages intended for one or more other components on the message board. It never explicitly reads from the message board. Components can join and withdraw without disrupting the functionality of the rest of the agent. When a new component is added to the agent it registers itself with the message board using a symbolic name and then sends advertisement messages to the board giving it a set of active message patterns. The message board stores the symbolic name together with the low level process identity of the component (needed for message forwarding) in its 'white pages' directory. It stores the advertisements, linked with the symbolic name of the component, in its 'yellow pages' directory. Because of the advertisements the component will be sent messages in the future. If the new component is pro-active, it may also place messages on the message board requesting services. It just needs to know the required format of the request, not the identities of the other components capable of servicing these requests.

An active message pattern is of the form

```
filter_pattern :: auxiliary_test_code
```

It is applied by the message board to each new message M placed on it. If the filter pattern matches the message, the auxiliary test code is executed. This usually further processes and tests parts of the message extracted as a result of the successful match with the filter pattern to make sure the message really is of interest to the component C that lodged it. However, it can also do arbitrary processing on behalf of C. As part of this processing it can communicate with other agent components. It can even communicate with C, or the component that sent the message, to determine whether or not M is really of interest to C *at this time*. If this test code succeeds, C is a *potential destination* for the message.

If the message M was communicated as a `broker-all` message, all the potential component destinations are found by applying each of the message board's active message patterns to the message. Then, the destination filter function which the sender may have attached to the message is applied to prune this set

of potential destinations. The message M is then forwarded to each destination in the pruned set, if there are any. Finally, the message is retained on the message board until its sender specified expiry time. This is in case a component C' subsequently advertises an active pattern that successfully applies to the message, between the time of its receipt and its expiry time. If so, providing this component also satisfies the destination filter function, the message is forwarded to C'. The default expiry time for a `broker` message is immediate, in which case the message is not usually retained by the message board. It will only be retained if no suitable destination was found at the time of receipt.

There is a similar scenario for a message M' communicated to the message board as a `broker-one` single destination message. The main difference is that M' will be forwarded to at most one component destination and, after forwarding, is not retained by the message board even if its expiry time has not lapsed. It will only be retained if its expiry time has not been reached *and* no suitable destination component has yet been found.

Once the message has been forwarded, the receiver can reply directly to the component that placed the message on the board. We assume that the sender field of the message contains either the sender's symbolic name, as registered with the message board, or its low level process identity. If the former, the reply can be sent via the message board as a `forward` message which only requires white pages lookup by the message board. If the latter, the message is communicated directly, bypassing the message board, using the direct process to process communication.

2.4 Agent head

Agents also have a process that deals with incoming messages from other agents. This is the purpose of the head communicator which is the external communication interface of the agent. The head communicator is also a security wall to the outside world. Incoming messages arrive first at it, and are then put on the the message board in order to be forwarded to the appropriate agent components, based on their content. Therefore, outside agents do not have direct access to the message board which is the internal agent backbone. The head may discard messages that are from unknown agents, or which are not syntactically well formed.

3 An example agent

In order to illustrate the above concepts, consider an agent that implements a travel agency which can be contacted for a ticket reservation. This agent can offer tickets for European and Asian flights. The agent has a broker component that deals with customer requests and two airline interface components dealing with the interaction with the airlines for price information and reservations. Each airline interface component deals with a particular geographical area (Figure 2). The agent also maintains a database with information regarding frequent customers, containing their address details and other personal information.

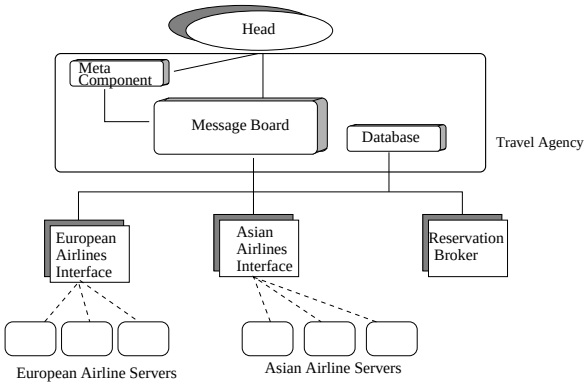

Fig. 2. A travel agent scenario

Assume that a message for a price quote for a fare between London and Tokyo arrives with date constraints and a constraint on the fare (fare < 700). This message is forwarded to the Broker because it is the only component to advertise an interest in requests for price quotes. The broker then consults the data base to see if there is information about the sender, perhaps the software agent for some customer, such as the airline preferences of the customer. Let us assume that either the customer is unknown, or it is known that they do not insist on non-stop flights. Because of this, and because of the low fare constraint, the Broker places a broker-all message on the message board for a flight reservation and fare quote without the constraint 'non-stop-direct'. This way, it will perhaps get replies from the European Airlines Interface component as well as the Asian Airlines Interface component, with the former offering cheaper quotes involving a change of flight.

4 The implementation platform

As an implementation platform for the above framework, we use the April language ([14],[13]). April offers a number of significant features such as: concurrency, symbolic list based computation, pattern matching, TCP/IP based communication between processes on different hosts, higher order features, advanced macro processing. These features can be extremely useful for the implementation of agent based applications. Our agent platform is implemented as a macro extension of April augmented with a set of predefined functions and processes.

A KQML message of the form:

```
"(request
   :language April
   :ontology ticket_reservation
   :reply_with id1
   :sender ‘‘klc_agent@zeus.ic.ac.uk’’
```

```
   :content "[(destination, Tokyo), ((From, London),
              (departure_date,"7/2/97"), (return_date,"14/2/97")]"
)"
```

can be represented as the April data value:

```
(request, [(language,April), (ontology,ticket_reservation),
           (reply_with ,id1), (sender, ``klc_agent@zeus.ic.ac.uk'')
           (content, [(destination, Tokyo), (From, London),
               (departure_date,"7/2/97"),(return_date,"14/2/97")])])
```

The active patterns that components advertise via the message board are represented as boolean function closures that take as argument a data value which is the April representation of a KQML message. The Broker component might send the message board something like:

```
(advertise, [(pattern_name,asian_tickets),
             (sender, Broker),
             (content, {(request,?LabelValuePairs) ::
                 (content, any?Content) in LabelValuePairs  and
                 (destination, any?Destination) in Content and
                 is_true{ asian_city(Destination) or
                          european_city(Destination) } ! kb }) ])
```

which is macroed into a message of the form:

```
(advertise,
    [(pattern_name,asian_tickets),
     (sender, Broker),
     (content,{lambda(?M) ->
          M={(request, ?LabelValuePairs) ::
             (content, any?Content) in  LabelValuePairs and
             (destination, any?Destination) in Content and
             valof{
                 (ask_if, {lambda(?DB)-> {
                     Destination in lookup(asian_city,DB) or
                     Destination in lookup(european_city,DB)}}
                 ) >> kb;

                 (reply, yes) :: sender == kb -> valis true
                 | (reply, no) :: sender == kb -> valis false}
       }}) ])
```

in which the content is a function closure (the `lambda`) to be applied to a message `M`. We will not explain the macro expanded code here. A detailed discussion of the use of April macros and the code generated is given in [16].

The above message, when received by the message board, will cause it to store the symbolic name of the component, `Broker`, the `pattern name` and the `code`. It stores this triple in its table of current active patterns. Notice the test condition

```
is_true { asian_city(Destination) or european_city(Destination) } ! kb
```

of the active pattern.

This is a direct query to the knowledge base component of the agent whose identity is held in the variable `kb` (it does not go via the message board). This is expanded into an explicit communication of a query function to the knowledge base (`>>` is April's message send operator). The query function will be applied to the data `DB` in the knowledge base. If the query function returns true, a `(reply,yes)` message will be returned as response. This is also an example of the power of the active patterns, which not only do local processing but can include calls to arbitrary test functions and communication to other components. This communication can be back to the component that sent the active pattern. The test functions could have been defined in and be private to the sender of the active pattern. Their code is automatically packaged up in the function closure that is the active pattern.

4.1 The message board

We adopt the KQML approach of having messages sent to the message board wrapped in an outer message with a performative indicating the senders intent. The message board takes appropriate action according to the performative of the wrapper message.

As well as broker messages the message board also accepts `forward` messages that give symbolic names for the destination. When a component is added to an agent, the first thing it does is register with the message board giving a symbolic name. The table of symbolic names paired with the process identity of the component that registered the name is the white pages directory of the message board. (The process identity is something like a CORBA inter-orb object reference. The process identity of the sender of any message is provided by the April communications system, on request, when a message is received). The table of registered active patterns is its yellow pages directory. Neither can be directly consulted. They are implicitly consulted by sending either a `forward` or `broker` message to the message board. On receipt of a:

```
(forward, [(content,M), (to,R)])
```

message, the board will forward `M` to the component that registered with the name R. For a

```
(forward_to_all,[(content,M), (to, [List of Component Names])])
```

it will send `M` to the list of the named components. If the destination name is not known, then a broker message:

```
(broker, [(content,M)])
```

can be sent to the message board. The message board will apply all the current active patterns to the message `M` and will forward `M` to the component that lodged the first pattern function it finds that successfully applies to `M`.

There is also a `broker_all` message and in both broker messages there can be a filter field giving a filter function which expresses the senders preferences with respect to the destinations. A destination filter of the form:

```
lambda(?D) -> is_true { has_capability(....)!kb
                        and current_no_of_tasks(N)!D and N<4}
```

can be used to filter out any destination `D` which the knowledge base does not record as having a particular capability, or which has a current number of task in excess of 3. This information is obtained by querying `D` (`current_no_of_tasks(N)!D`). The assumes that D is itself multi-threaded and is able to answer the query whilst concurrently executing these tasks. The query will be answered using `D`'s private knowledge base.

A component's message patterns are sent to the message board as a

```
(advertise, [(pattern_name,N),(sender,S),(content,<function defn>)])
```

message. Here, `S` is the symbolic name that the component has registered with the message board.

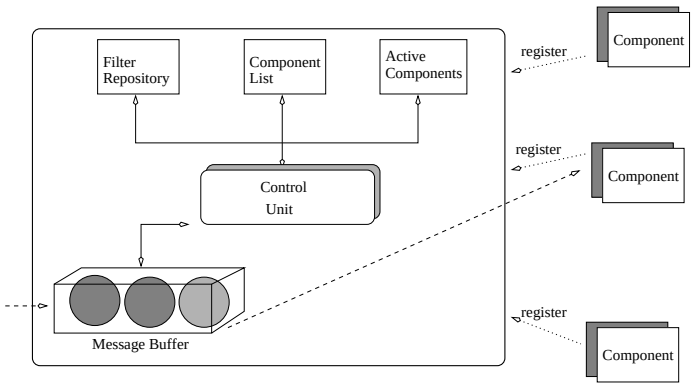

Fig. 3. The message board internal structure

The internal architecture of the message board is depicted in Figure 3. The `Component List` is a list of records that contain the symbolic name of each registered component and its low level process identity (needed for forwarding messages). This name indirection allows a component with a particular symbolic name `N` to be replaced by another process offering the same message interface. All that is needed is a protocol which first deregisters the old process with that name and then registers the new process using the same name `N`. Since all forward messages and advertisements (held in the `Filter repository`) use the symbolic name and not the process identity, the new process will transparently replace the old as far as the other components and message board is concerned. This possibility of component upgrade is the reason why the message board should

always be used for the opening communication of a conversation or transaction between components, using either a broker or forward message.

The `Active components` list is a sublist of all the registered components. After registration a component must send an explicit `activate` message to be put on the active list. It can also send a `deactivate` message to get itself removed. Only active components get sent messages. This mechanism allows a component to temporarily suspend its availability.

When a message is placed in the message buffer it is processed by the control unit which accesses the three lists: `component list`, `active list`, `filter repository`.

4.2 Behaviour components

Components are implemented as April processes, which when they start executing immediately try to attach to the agent. The attachment involves sending a "registration" message to the agent head and consequently to the message board, and (optionally) the registration of active patterns on the message board. The implementation of the `reservation_broker` of the previous example is sketched below.

```
reservation_broker()
{
  (init_input, (string?agent_name,
                string?plug_in_name, input_type[]?Input) -> {
     /* Get information about the agent and
        specific inputs for the component. */

    /* Attaching to the Agent.  Get message board (mboard)
       and database (DB) handles form the agent.*/
    (handle?mboard,
     handle?DB) = attach_to_agent(plug_in_name, MyHandle, agent_name);

    (advertise, ...) >> mboard;   /* Register active patterns */

    ... /* Rest of code */
}
```

All the components are implemented as no argument procedures in order to maintain a uniform view of them. This has proven useful for their persistent storage (see also Section 5.1). When the component is forked as a process it is sent a message containing the name of the agent that it will attached to (`agent_name`), the name under which it will get attached (`plug_in_name`) and other initial data that needs to be given to the component (`Input`). The `plug_in_name` will be the symbolic name which will be used by the message board to identify the component. The `Input` is a list of attribute/value pairs. The component picks up this information using the message receive statement:

```
(init_input, (string?agent_name,
              string?component_name, input_type[]?Input) -> ...
```

which is the first statement executed by the component. For the specific example, the creation of the `reservation_broker` component would be achieved using:

```
handle?H = fork reservation_broker;

(init_input, ("travel_agent", "reservation_broker",
              [(destinations, ["asia", "europe"])])) >> H;
```

This will fork the "reservation broker" process and will send all the "bootstrapping" information via the `init_input` message to the process, identified by the handle `H`, i.e. the `reservation_broker`. After this information is received by the reservation broker, the call `attach_to_agent` will perform the appropriate steps for attaching the component to the agent. Essentially, this means asking the agent for the handle identifier of its message board and database process components storing these in local variables `mboard` and `DB` of the broker, and registering its `plug_in_name` with the message board. After getting hold of the message board address, it then registers its patterns before executing the rest of the component code.

5 The agent management toolkit

The agent architecture described above is supported by an agent management toolkit[1]. This consists of a set of server that offer agent management services. A server for persistently storing components is supplied. Components implemented as closures can be stored in this server. They can be dynamically fetched from this server and attached to agents. Descriptions of agents can be provided and stored in a separate *agent library*. These descriptions specify the components an agent consists of. The descriptions can be retrieved, and based on them, the appropriate components can be fetched from the code server and attached to the agent. An integrated mechanism for creating agents is provided by a management unit (illustrated in figure 4) which provides the glue between these servers.

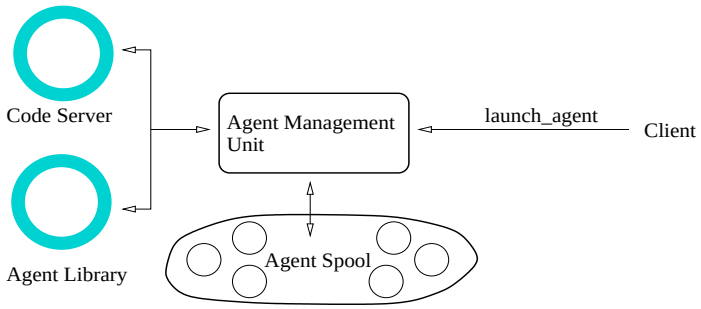

Fig. 4. The Agent Management component

[1] The agent architecture and this layer are called ALFA (Agent Layer For April).

It controls a spool of agents and can be requested to create a new agent and add it into the spool.

5.1 Code server

The component-oriented approach, allows agents to be easily customised. Components can be added or deleted from an agent description, so that it has different set of capabilities when launched. Alternatively, after being launched new components can be added and old ones replaced.

The code for agent components can be stored in a special process, called the *code server*. Other processes can contact the code server in order to retrieve the code they need. When the agent is started up, for example, the code for each component is retrieved from the code server, before being launched as described in section 4.2 The following program sketches the implementation of this code server:

```
code_server()
{
   /* Table for storing component closures */
   definition CodeTable schema (string?code_name, (){}?code);
   ... /* Load any pre-registered closures */

   repeat {
     ({add_code}, (string?code_name, (){}?CodeAbstraction)) -> {
         ... /* Adding a new closure */ }
     | (retrieve_code, string?code_name) -> {
         ... /* Retrieving a code abstraction */ }
     | (delete_code, string?code_name) -> {
         ... /* Deleting a registered closure */ }
   } until quit;
};
```

The code server maintains a database table (`CodeTable`) where the closures are stored. They are stored as pairs with type `(string, {}())`, which contain a symbolic name for the code abstraction and the abstraction[2]. The symbolic name is used for retrieving and deleting the code. This code table is persistently backed-up to the file system.

When a process wants to register a new piece of code that can be used later either by itself or by other processes, it sends a `add_code` message to the code server. In this message it includes the name of the closure and the closure as a procedure abstraction that takes no input arguments. For example, in order to register the code for the `reservation_broker` process the following message needs to be sent to the code server:

```
(add_code, ("reservation_broker",
            reservation_broker)) >> handle??"CodeServer"
```

For deleting code the `delete_code` message is provided. The name of the closure has to be specified. For retrieving a specific closure now, the message:

[2] `{}()` is the April type for a no argument procedure.

```
(retrieve_code, string?code_name) -> ...
```

is provided. When it is received the code is fetched from the `CodeTable` and it is sent back in the form of a message like the following:

```
(retrieved_code, (code_name, CodeAbstraction)) >> replyto;
```

If for some reason the code is not found, either because it does not exist or a wrong name was specified, an error message is generated.

In order, for example, to retrieve the code for the `"reservation_broker"` process the following statements should be used:

```
(retrieve_code, "reservation_broker") >> handle??"CodeServer";
(retrieved_code, ("reservation_broker", (){}?componentCode)) -> {
    spawn_component(componentCode, "travel_agent", "reservation_broker",
                                   [(destinations, ["asia", "europe"]);
};
```

And after the `retrieved_code` message is received the `componentCode` can be executed. A special procedure, `spawn_component` is provided that will fork the component process and send the input arguments as a message, in the manner described in section 4.2.

```
spawn_component((){}?componentCode, string?agent_name,
                string?plug_in_name, input_type[]?Input) {
    /* fork component */
    handle?H = fork componentCode;

    /* supply component with initial input information.
       It is the first message received by the component. */
    (init_input, (agent_name, plug_in_name, Input)) >> H;
}
```

The `spawn_component` procedure is passed the identity of the agent to which the component is being attached (`agent_name`), the name under which it will be known (`plug_in_name`) and input information (`Input`). Now as soon as the component is forked, it will attach itself to the agent.

The code retrieval and component launch have been wrapped up under a single routine called `launch_component`, the implementation of which is:

```
launch_component(string?agent_name, string?component_name,
                 string?plug_in_name, input_type[]?Input)
{
    (retrieve_code, component_name) >> handle??"CodeServer";
    /* Retrieving component code */
    (retrieved_code, (component_name, (){}?componentCode)) -> {
        /* Forking component */
        spawn_component(componentCode, agent_name, plug_in_name, Input);
    };
};
```

5.2 The agent library

The agent is constructed by attaching a set of pre-defined components to its message board. These components are implemented as April processes and are stored in the code server. If we had to describe the agent in some symbolic form, that form would basically include the list of the agent components. An agent description in general has the format:

```
(agent_name,
   [ (component_name, plug_in_name, [ ... /* Input information */])
     ... /* Other components */ ]
)
```

The description contains the agent name, and the list of names of the components. For the travel agent example, the description would be similar to:

```
("travel_agent", [ /* Component descriptions */
  ("reservation_broker", "reservation_broker",
                         [(destinations, ["asia","europe"])]),
  ( "airline_interface", "european_airline_interface", [
              (destination, "europe"),
              (european_airlines, ["lufthansa", "ba"]) ]),
  ( "airline_interface", "asian_airline_interface", [
              (destination, "asia"),
              (asian_airlines, ["jal", "ana", "korean_airlines"]) ])
])
```

This is the description of the agent with name `"travel_agent"` which consists of three components described above, namely a central `"reservation_broker"` and two components that handle the interface with airline servers (`"european_airline_interface"`, `"asian_airline_interface"`). For the airline interface components the same component code is used, customised with different information regarding the destination of the flights and the airlines that it deals with.

Ideally, what we would like, is to have a library of such agent descriptions which can be re-used. Other entities can retrieve symbolic agent descriptions and use them to launch (possibly customised) instances of them. We have developed such a server and we call it the *agent library*. It is an April server which maintains a table of agent descriptions. This server is publically forked under the name `"AgentLibrary"`. It can be sent description, asked to delete old ones, and asked to provide description based on its symbolic name. For example, a client process can send the message

```
(new_instance_description, "travel_agent",
     ... /* The travel agent description */) >> handle??"AgentLibrary";
```

which includes the description of the travel agent given above. Later we can retrieve this description by name and launch the agent using the procedure:

```
launch_agent(string?agent_name) {
   /* Fork generic agent components, i.e. meta-component,
```

```
      message board, head, database. */
    fork mu(){agent_proc(agent_name)};

    /* Retrieve the description of the agent */
    (supply_agent_description, agent_name) >> handle??"AgentLibrary"
    (description_is, agent_name, ?agentDescription) -> {

      /* Retrieve the list of components from the agent description */
      comp_type[]?componentDescription = agentDescription.components;

      /* Launch the individual components */
      for (string?component, string?plug_in_name,
                          input_type?Input) in componentDescription do {
          launch_component(agent_name, component, plug_in_name, Input);
       };
    }
 };
```

The result of a `launch_agent` call is to create a new instance of the `agent_proc` process which will automatically launch the domain independent components of the agent (meta-component, message board, database, head). As soon as the generic components are launched, it will get hold of the agent description from the agent library (`supply_agent_description` message); then extract the list of component descriptions (`componentDescription`) and launch them one at a time (`for ... do ...` loop).

5.3 The management platform

The above servers and procedures offer a collection of tools that provide quite a powerful and open platform for building agents. The agent management platform provides the top level interface. It provides a high level, easy to use interface to external clients for creating and managing agents. An external client, can launch an agent simply by sending a message such as:

```
(launch_agent, "travel_agent") >> handle??"ManagementPlatform"
```

to the `"ManagementPlatform"`. The management platform also provides the capability of suspending and killing agents. There is also a graphical interface for visualising the current state of the multi-agent system and for invoking the services offered by the platform.

6 Discussion and Conclusions

Software construction increasingly follows a component oriented style, where components are constructed and used off the shelf for the construction of complex software systems. The construction of agent systems could also follow this philosophy ([8], [15]).

In this paper, an agent architecture is proposed that can be used to integrate pre-existing components. Even components written in a conventional language such as C can be integrated by providing a wrapper that accepts our KQML style messages, mapping them into internal procedure calls, and which sends the required advertisements. April has a C and Java API, which allows Unix processes written in these languages to read and write April messages. Using it, we can implement components whose main functionality is not programmed in April.

The integrating component of the agent is the message board. Attached components provide a symbolic name which is used when messages need to be forwarded to a specific component. This is used to identify the component in all other communications with the message board. It allows for dynamic component replacement. Furthermore, a content based approach to message routing has been also adopted. Components can specify active patterns in the form of code abstractions that the message board uses in order to route messages between components, when the name of a suitable destination component is not known.

Other agents do not have direct access to the message board. Messages from the outside reach the message board via a communication module, the agent head, that acts as a communication mediator between the agent components and the external entities. A network of agents can be integrated using an inter-agent message board of the functionality as the internal message board. This allows for a recursive agent structure, in which components themselves are agents.

In addition to the infrastructure for constructing a single agent, a set of additional servers are supplied which comprise a complete agent management toolkit. For persistently storing components a special persistency server called *code server* is provided. Also, the description of the components that comprise a single agent can be stored in another server, called the *agent library*. A central agent management server is used as the glue. It receives requests for launching agents and by interacting with the other two servers takes all the appropriate steps to launch the agents.

This infrastructure has been used to develop two concrete agent based applications. One is an agent layer for managing telecommunication networks ([17]). The other ([1]), communicates with a Java-based control software to control the environmental conditions in an office in accordance with the location and preferences of the occupants.

A similar approach, starting from an underlying agent architecture on which layers are built that can be used for the development of domain specific agent based applications is presented in [9]. The architecture assumed there is a BDI based architecture. Also, in [3] the issue of semantics of inter-agent communication based on speech acts is addressed, something that is not explicitly dealt in the work presented in this paper.

References

1. M. Boman, P. Davidsson, N. Skarmeas, K. L Clark, and R. Gustavvson. Energy

Saving and Added Customer Value in Intelligent Buildings. *PAAM'98*, 1998.
2. F. Buschmann and et al. *A System of Patterns: Pattern Oriented Software Architecture*. Wiley and Sons, 1996.
3. B. Chaib-draa and D. Vanderveken. Agent communication language: Toward semantics based on success and satisfaction. In J. P. Müller, M. P. Singh, and A. S. Rao, editors, *Intelligent Agents V — Proceedings of the Fifth International Workshop on Agent Theories, Architectures, and Languages (ATAL-98)*, Lecture Notes in Artificial Intelligence. Springer-Verlag, Heidelberg, 1999. In this volume.
4. K. L. Clark and N. Skarmeas. A Harness Language for Cooperative Information Systems. In M. Papazoglou and G. Schlageter, editors, *Cooperative Information Systems: Trends and Directions*. Academinc Press, 1998.
5. Y. Demazeau and Jean Pierre M'uller. Decentralised Artificial Intelligence. In Y. Demazeau and Jean Pierre M'uller, editors, *Decentralised Artificial Intelligence*, pages 3–13. Elsevier Science Publisher, 1990.
6. H. Heugeneden, editor. *IMAGINE final report*. Siemens, Munich, Germany, 1994.
7. L.P. Kaelbling. A situated automata approach to the design of embedded systems. *SIGART Bulletin*, 1991.
8. E. A. Kendall and M. T. Malkoum. Design Patterns for the Development of Multiagent Systems. In C. Zhang and D. Lukose, editors, *Multi-Agent Systems: Methodologies and Applications*. Springer Verlag, 1997.
9. D. Kinny. The AGENTIS agent interaction model. In J. P. Müller, M. P. Singh, and A. S. Rao, editors, *Intelligent Agents V — Proceedings of the Fifth International Workshop on Agent Theories, Architectures, and Languages (ATAL-98)*, Lecture Notes in Artificial Intelligence. Springer-Verlag, Heidelberg, 1999. In this volume.
10. D. Kuokka and L. Harada. On using KQML for Matchmaking. *Proceedings of the First International Conference on Multi-Agent Systems*, pages 239–245, 1995.
11. V. R. Lesser and Daniel D. Corkill. The Distributed Vehicle Monitoring Testbed: A Tool for Investigating Didtributed Problem Solving Networks. *The AI Magazine*, pages 15–33, Fall 1994.
12. P Maes. The agent network architecture (ANA). *SIGART Bulletin*, 1991.
13. F. McCabe. April reference manual. Technical report, Fujitsu Laboratories Ltd., Japan, 1996.
14. F. McCabe and K. L. Clark. April – Agent PRocess Interaction Language. *Intelligent Agents*, 1994.
15. T. D. Meijler and O. Nierstrasz. Beyond Objects: Components. In M. Papazoglou and G. Schlageter, editors, *Cooperative Information Systems: Trends and Directions*. Academinc Press, 1998.
16. Nikolaos Skarmeas. *Agents as Objects with Knowledge Based State*. Imperial College Press, December 1998.
17. N. Skarmeas and K. L. Clark. Intelligent Agents for Telecoms Applications (IATA'96). *IATA'96 Workshop of the European Conference on Artificial Intelligence*, August 1996.