Using multi-context systems to engineer executable agents

Jordi Sabater[1], Carles Sierra[1], Simon Parsons[2], Nicholas R. Jennings[2]

[1] IIIA - Artificial Intelligence Research Institute
CSIC - Spanish Council for Scientific Research
Campus UAB, 08193 Bellaterra, Catalonia, Spain.
{jsabater,sierra}@iiia.csic.es
[2] Department of Electronic Engineering
Queen Mary and Westfield College
University of London, London E1 4NS, UK.
{S.D.Parsons,N.R.Jennings}@elec.qmw.ac.uk

Abstract. In the area of agent-based computing there are many proposals for specific system architectures, and a number of proposals for general approaches to building agents. As yet, however, there are comparatively few attempts to relate these together, and even fewer attempts to provide methodologies which relate designs to architectures and then to executable agents. This paper provides a first attempt to address this shortcoming; we propose a general method of defining architectures for logic-based agents which can be directly executed. Our approach is based upon the use of multi-context systems and we illustrate its use through the specification of a simple agent.

1 Introduction

Agent-based computing is fast emerging as a new paradigm for engineering complex, distributed systems [15, 28]. An important aspect of this trend is the use of agent architectures as a means of delivering agent-based functionality (cf. work on agent programming languages [16, 24, 26]). In this context, an architecture can be viewed as a separation of concerns—it identifies the main functions that ultimately give rise to the agent's behaviour and defines the interdependencies that exist between them. As agent architectures become more widely used, there is an increasing demand for unambiguous specifications of them and there is a greater need to verify implementations of them. To this end, a range of techniques have been used to formally specify agent architectures (eg Concurrent MetateM [9, 27], DESIRE [3, 25] and Z [6]). However, these techniques typically fall short in at least one of the following ways: (i) they enforce a particular view of architecture upon the specification; (ii) they offer no explicit structures for modelling the components of an architecture or the relationships between them; (iii) they leave a gap between the specification of an architecture and its implementation.

To rectify these shortcomings, we have proposed [20] the use of *multi-context systems* [12] as a means of specifying and implementing agent architectures. Multi-context systems provide an overarching framework that allows distinct theoretical components to be defined and interrelated. Such systems consist of a set of contexts, each of which can informally be considered to be a logic and a set of formulae written in that logic, and a set of bridge rules for transferring information between contexts. Thus, different contexts can be used to represent different components of the architecture and the interactions between these components can be specified by means of the bridge rules between the contexts. We believe multi-context systems are well suited to specifying and modelling agent architectures for two main types of reason: (i) from a *software engineering perspective* they support modular decomposition and encapsulation; and (ii) from a *logical modelling perspective* they provide an efficient means of specifying and executing complex logics. Each of these broad areas will now be dealt with in turn.

Let us first consider the advantages from a software engineering perspective. Firstly, multi-context systems support the development of modular architectures. Each architectural component—be it a functional component (responsible for assessing the agent's current situation, say) or a data structure component (the agent's beliefs, say)—can be represented as a separate context. The links between the components can then be made explicit by writing bridge rules to link the contexts. This ability to directly support component decomposition offers a clean route from the high level specification of the architecture through to its detailed design. Moreover, this basic philosophy can be applied no matter how the architectural components are decomposed or how many architectural components exist. Secondly, since multi-context systems encapsulate architectural components and enable flexible interrelationships to be specified, they are ideally suited to supporting re-use (both of designs and implementations). Thus, contexts that represent particular aspects of the architecture can be packaged as software components (in the component-ware sense [23]) or they can be used as the basis for specialisation of new contexts (inheritance in the object-oriented sense [2]).

Moving onto the logical modelling perspective, there are four main advantages of adopting a multi-context approach. The first is an extension of the software engineering advantages which specifically applies to logical systems. By breaking the logical description of an agent into a set of contexts, each of which holds a set of related formulae, we effectively get a form of many-sorted logic (all the formulae in one context are a single sort) with the concomitant advantages of scalability and efficiency. The second advantage follows on from this. Using multi-context systems makes it possible to build agents which use several different logics in a way that keeps the logics neatly separated (all the formulae in one logic are gathered together in one context). This either makes it possible to increase the representational power of logical agents (compared with those which use a single logic) or simplify agents conceptually (compared with those which use several logics in one global context). This latter advantage is illustrated in [20] where we use multi-context systems to simplify the construction of a BDI agent.

Both of the above advantages apply to any logical agent built using multi-context systems. The remaining two advantages apply to specific types of logical agent—those which reason about their beliefs and those of other agents. The first is that multi-context systems make it possible [12] to build agents which reason in a way which conforms to the use of modal logics like KD45 (the standard modal logic for handling belief) but which obviates the difficulties usually inherent in theorem proving in such logics. Again this is illustrated in [20]. Thus the use of multi-context systems makes it easy to directly execute agent specifications where those specifications deal with modal notions. The final advantage is related to this. Agents which reason about beliefs are often confronted with the problem of modelling the beliefs of other agents, and this can be hard, especially when those other agents reason about beliefs in a different way (because, for instance, they use a different logic). Multi-context systems provide a neat solution to this problem [1, 5].

When the software engineering and the logical modelling perspectives are combined, it can be seen that the multi-context approach offers a clear path from specification through to implementation. By providing a clear set of mappings from concept to design, and from design to implementation, the multi-context approach offers a way of tackling the gap (gulf!) that currently exists between the theory and the practice of agent-based systems. This paper extends the suggestion made in [20] by further refining the approach, extending the representation and providing additional support for building complex agents.

2 Multi-context agents

As discussed above, we believe that the use of multi-context systems offers a number of advantages when engineering agent architectures. However, multi-context systems are not a panacea. We believe that they are most appropriate when building agents which are logic-based and are therefore largely deliberative[1].

2.1 The basic model

Using a multi-context approach, an agent architecture consists of four basic types of component. These components were first identified in the context of building theorem provers for modal logic [12], before being identified as a methodology for constructing agent architectures [17]. The components are[2] :

– *Units*: Structural entities representing the main components of the architecture.
– *Logics*: Declarative languages, each with a set of axioms and a number of rules of inference. Each unit has a single logic associated with it.
– *Theories*: Sets of formulae written in the logic associated with a unit.
– *Bridge rules*: Rules of inference which relate formulae in different units.

Units represent the various components of the architecture. They contain the bulk of an agent's problem solving knowledge, and this knowledge is encoded in the specific theory that the unit encapsulates. In general, the nature of the units will vary between architectures. For example, a BDI agent may have units which represent theories of beliefs, desires and intentions (as in [20]), whereas an architecture based on a functional separation of concerns may have units which encode theories of cooperation, situation assessment and plan execution. In either case, each unit has a suitable logic associated with it. Thus the belief unit of a BDI agent has a logic of belief associated with it, and

[1] See [29] for a discussion of the relative merits of logic-based and non logic-based approaches to specifying and building agent architectures.
[2] For more detail see [17].

the intention unit has a logic of intention. The logic associated with each unit provides the language in which the information in that unit is encoded, and the bridge rules provide the mechanism by which information is transferred between units.

Bridge rules can be understood as rules of inference with premises and conclusions in different units. For instance:

$$\frac{u_1 : \psi, u_2 : \varphi}{u_3 : \theta}$$

means that formula θ may be deduced in unit u_3 if formulae ψ and φ are deduced in units u_1 and u_2 respectively.

When used as a means of specifying agent architectures [17, 20], all the elements of the model, both units and bridge rules, are taken to work concurrently. In practice this means that the execution of each unit is a non-terminating, deductive process[3]. The bridge rules continuously examine the theories of the units that appear in their premises for new sets of formulae that match them. This means that all the components of the architecture are always ready to react to any change (external or internal) and that there are no central control elements.

2.2 The extended model

The model as outlined above is that introduced in [17] and used in [20]. However, this model has proved deficient in a couple of ways, both connected to the dynamics of reasoning. In particular we have found it useful to extend the basic idea of multi-context systems by associating two control elements with the bridge rules: *consumption* and *time-outs*. A consuming condition means the bridge rule removes the formula from the theory which contains the premise (remember that a theory is considered to be a set of formulae). Thus in bridge rules with consuming conditions, formulae "move" between units. To distinguish between a consuming condition and a non-consuming condition, we will use the notation $u_i > \psi$ for consuming and $u_i : \psi$ for non-consuming conditions. Thus:

$$\frac{u_1 > \psi, u_2 : \varphi}{u_3 : \theta}$$

means that when the bridge rule is executed, ψ is removed from u_1 but φ is not removed from u_2.

Consuming conditions increase expressiveness in the communication between units. With this facility, we can model the movement of a formula from one theory to another (from one unit to another), changes in the theory of one unit that cause the removal of a formula from another one, and so on. This mechanism also makes it possible to model the concept of state since having a concrete formula in one unit or another might represent a different agent state. For example, later in the paper we use the presence of a formula in a particular unit to indicate the availability of a resource.

A time-out in a bridge rule means there is a delay between the instant in time at which the conditions of the bridge rule are satisfied and the effective activation of the rule. A time-out is denoted by a label on the right of the rule; for instance:

$$\frac{u_1 : \psi}{u_2 : \varphi}[t]$$

[3] For more detail on exactly how this is achieved, see [21].

means that t units of time after the theory in unit u_1 gets formula ψ, the theory in unit u_2 will be extended by formula φ. If during this time period formula ψ is removed from the theory in unit u_1, this rule will not be applied. In a similar way to consuming conditions, time-outs increase expressiveness in the communication between units. This is important when actions performed by bridge rules need to be retracted if a specific event does not happen after a given period of time. In particular, it enables us to represent situations where silence during a period of time may mean failure (in this case the bridge rules can then be used to re-establish a previous state)[4].

3 Modular agents

Using units and bridge rules as the only structural elements is cumbersome when building complex agents (as can be seen from the model we developed in [20]). As the complexity of the agent increases, it rapidly becomes very difficult to deal with the necessary number of units and their interconnections using bridge rules alone. Adding new capabilities to the agent becomes a complex task in itself. To solve this problem we suggest adding another level of abstraction to the model—the *module*.

3.1 Introducing modules

A module is a set of units and bridge rules that together model a particular capability or facet of an agent. For example, planning agents must be capable of managing resources, and such an agent might have a module modeling this ability. Similarly, such an agent might have a module for generating plans, a module for handling communication, and so on. Thus modules capture exactly the same idea as the "capabilities" discussed by Busetta *et al.* [4]. Unlike Busetta *et al.*, we do not currently allow modules to be nested inside one another, largely because we have not yet found it necessary to do so. However, it seems likely that we will need to develop a means of handling nested hierachies of modules in order to build more complex agents than we are currently constructing.

Each module must have a communication unit. This unit is the module's unique point of contact with the other modules and it knows what kind of messages its module can deal with. All of an agent's communication units are inter-connected with the others using *multicast bridge rules* (*MBR*s) as in Figure 1. This figure shows three MBRs (the rectangles in the middle of the diagram) each of which has a single premise in module a and a single conclusion in each of the modules n_i.

Since the MBRs send messages to more than one module, a single message can provoke more than one answer and, hence, contradictory information may appear. There are many possible ways of dealing with this problem, however here we consider just one of them as an example. We associate a weight with each message. This value is assigned

[4] Both of these extensions to the standard multi-context system incur a cost. This is that including them in the model means that the model departs somewhat from first order predicate calculus, and so does not have a fully-defined semantics. We are currently looking at using linear logic, in which individual propositions can only be used once in any given proof, as a means of giving a semantics to consuming conditions, and various temporal logics as a means of giving a semantics to time-outs.

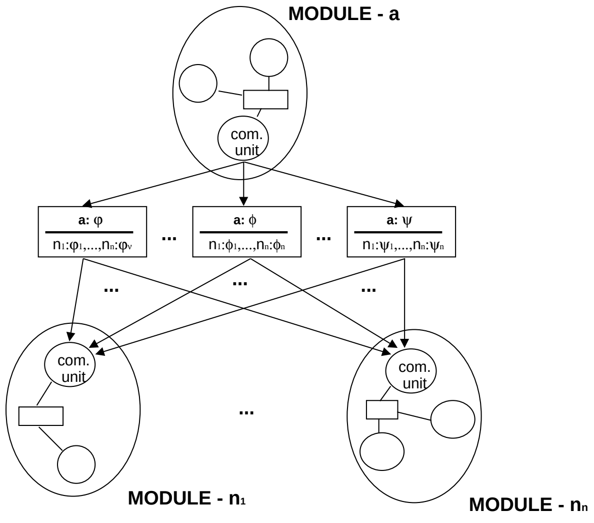

Fig. 1. The inter-connection of modules (from a's perspective only)

to the message by the communication unit of the module that sends it out. Weights belong to $[0, 1]$ (maximum importance is 1 and minimum is 0), and their meaning is the strength of the opinion given in the message, and this can be used to resolve contradictory messages. For instance, the message with highest weight might be preferred, or the different weights of incoming messages could be combined by a communication unit receiving them to take a final decision (for instance using the belief revision mechanism described in [18]). Note that weights are used only in *inter-module* messages.

3.2 Messages between modules

Given a set AN of agent names and a set MN of module names, an inter-module message has the form:

$$I(S, R, \varphi, G, \psi)$$

where

- I is an illocutionary particle that specifies the kind of message.
- S and R both have the form $A[/m]^*$[5] where $A \in AN$ or $A = \mathit{Self}$ (Self refers to the agent that owns the module) and $m \in MN$, or $m = \mathit{all}$ (all denotes all the modules within that agent). S reflects who is sending the message and R indicates to whom it is directed.

[5] As elsewhere we use BNF syntax, so that $A[/m]^*$ means A followed by one or more occurrences of $/m$.

– φ is the content of the message.
– G is a record of the derivation of φ. It has the form: $\{\{\Gamma_1 \vdash \varphi_1\} \ldots \{\Gamma_n \vdash \varphi_n\}\}$ where Γ is a set of formulae and φ_i is a formula with $\varphi_n = \varphi$ [6].
– $\psi \in [0, 1]$ is the weight associated with the message.

To see how this works in practice, consider the following. Suppose that an agent (named B) has four modules (*a*, *b*, *c*, *d*). Module *a* sends the message:

$$Ask(\mathit{Self}/a, \mathit{Self}/all, Give(B, A, Nail), \psi_1, 0.5)$$

This means that module a of agent B is asking all its modules whether B should give A a nail. The reason for doing this is ψ_1 and the weight a puts on this request is 0.5. Assume modules *c* and *d* send the answer

$$Answer(\mathit{Self}/c, \mathit{Self}/a, not(Give(B, A, Nail)), \psi_2, 0.6)$$

and

$$Answer(\mathit{Self}/d, \mathit{Self}/a, not(Give(B, A, Nail)), \psi_3, 0.7)$$

while module *b* sends

$$Answer(\mathit{Self}/b, \mathit{Self}/a, Give(B, A, Nail), \psi_4, 0.3)$$

Currently we treat the weights of the messages as possibility measures [7], and so combine the disjunctive support for $not(Give(B, A, Nail))$ using max. As this combined weight is higher than the weight of the positive literal, the communication unit of module *a* will accept the opinion $not(Give(B, A, Nail))$.

The messages we have discussed so far are those which are passed around the agent itself in order to exchange information between the modules which compose it. Our approach also admits the more common idea of messages between agents. Such inter-agent messages have the same basic form, but they have two minor differences:

– S and R are agent names (i.e. $S, R \in AN$), no modules are specified.
– there is no degree of importance (because it is internal to a particular agent—however inter-agent messages could be augmented with a degree of belief [18] which could be based upon the weight of the relevant intra-agent messages.)

With this machinery in place, we are in a position to specify realistic agent architectures.

4 Specifying a simple agent

This section gives a specification of a simple agent using the approach outlined above. The agent in question is a simple version of the home improvement agents first discussed in [19], which is supposed to roam the authors' homes making small changes

[6] In other words, G is exactly the set of grounds of the argument for φ [20]. Where the agent does not need to be able to justify its statements, this component of the message can be discarded. Note that, as argued by Gabbay [10] this approach is a generalisation of classical logic—there is nothing to stop the same approach being used when messages are just formulae in classical logic.

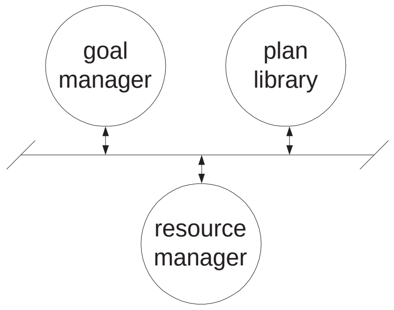

Fig. 2. The modules in the agent

to their environment. In particular the agent we discuss here attempts to hang pictures. As mentioned, the agent is rather simpler than those originally introduced, the simplification being intended to filter out unnecessary detail that might confuse the reader. As a result, compared with the more complex versions of the home improvement agents described in [20], the agent is not quite solipsistic (since it has some awareness of its environment) but it is certainly autistic (since it has no mechanisms for interacting with other agents). For an example of the specification of a more complex agent, see [21].

4.1 A high-level description

The basic structure of the agent is that of Figure 2. There are three modules connected by multicast bridge rules. These are the plan library (PL), the resource manager (RM), and the goal manager (GM). Broadly speaking, the plan library stores plans for the tasks that the agent knows how to complete, the resource manager keeps track of the resources available to the agent, and the goal manager relates the goals of the agent to the selection of appropriate plans.

There are two types of message which get passed along the multicast bridge rules. These are the following:

– **Ask:** a request to another module.
– **Answer:** an answer to an inter-module request.

Thus all the modules can do is to make requests on one another and answer those requests. We also need to define the predicates which form the content of such messages. Given a set of agent names AN, and with $AN' = AN \cup \{Self\}$.

– $Goal(X)$: X is a string describing an action. This denotes the fact that the agent has the goal X.
– $Have(X, Z)$: $X \in AN'$ is the name of an agent (here always instantiated to $Self$, the agent's name for itself, but a variable since the agent is aware that other agents may own things), and Z is the name of an object. This denotes Agent X has possession of Z.

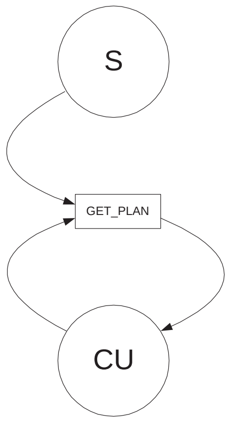

Fig. 3. The plan library module

Note that in the rest of the paper we adopt a Prolog-like notation in which the upper case letters X, Y, Z, P are taken to be variables.

As can be seen from the above, the content of the messages is relatively simple, referring to goals that the agent has, and resources it possesses. Thus a typical message would be a request from the goal manager as to whether the agent possesses a hammer:

$$ask(Self/GM, Self/all, goal(have(Self, hammer)), \{\})$$

Note that in this message, as in all messages in the remainder of this paper, we ignore the weight in the interests of clarity. Such a request might be generated when the goal manager is trying to ascertain if the agent can fulfill a possible plan which involves using a hammer.

4.2 Specifications of the modules

Having identified the structure of the agent in terms of modules, the next stage in the specification is to detail the internal structure of the modules in terms of the units they contain, and the bridge rules connecting those units. The structure of the plan library module is given in Figure 3. In this diagram, units are represented as circles, and bridge rules as rectangles. Arrows into bridge rules indicate units which hold the antecedents of the bridge rules, and arrows out indicate the units which hold the consequents. The two units in the plan library module are:

- The communication unit (CU): the unit which handles communication with other units.
- The plan repository (S): a unit which holds a set of plans.

The bridge rule connecting these units is:

$$\mathsf{GET\_PLAN} = \frac{\begin{array}{c} CU > ask(Self/Sender, Self/all, goal(Z), \{\}), \\ S : plan(Z, P) \end{array}}{CU : answer(Self/PL, (Self/Sender, goal(Z), \{P\})}$$

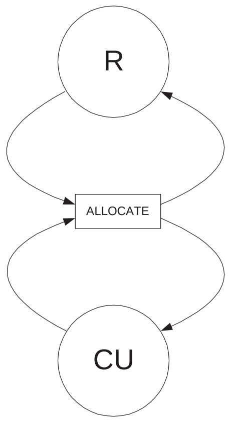

Fig. 4. The resource manager module

where the predicate $plan(Z, P)$ denotes the fact that P, taken to be a conjunction of terms, is a plan to achieve the goal Z[7].

When the communication unit sees a message on the inter-module bus asking about the feasibility of the agent achieving a goal, then, if there is a plan to achieve that goal in the plan repository, that plan is sent to the module which asked the original question. Note that the bridge rule has a consuming condition—this is to ensure that the question is only answered once.

The structure of the resource manager module is given in Figure 4. The two units in this module are:

– The communication unit (CU).
– The resource respository (R): a unit which holds the set of resources available to the agent.

The bridge rule connecting the two units is the following:

$$\mathsf{ALLOCATE} = \frac{\begin{array}{c} CU > ask(Self/Sender, Self/Receiver, goal(have(X,Z)), \{\}), \\ R > resource(Z, free) \end{array}}{\begin{array}{c} CU : answer(Self/RM, Self/Sender, have(X,Z), \{\}), \\ R : resource(Z, allocated) \end{array}}$$

where the $resource(Z, allocated)$ denotes the fact that the resource Z is in use, and $resource(Z, free)$ denotes the fact that the resource Z is not in use.

When the communication unit sees a message on the inter-module bus asking if the agent has a resource, then, if that resource is in the resource repository and is currently free, the formula recording the free resource is deleted by the consuming condition, a new formula recording the fact that the resource is allocated is written to the repository, and a response is posted on the inter-module bus. Note that designating a resource

[7] Though here we take a rather relaxed view of what constitutes a plan—our "plans" are little more than a set of pre-conditions for achieving the goal.

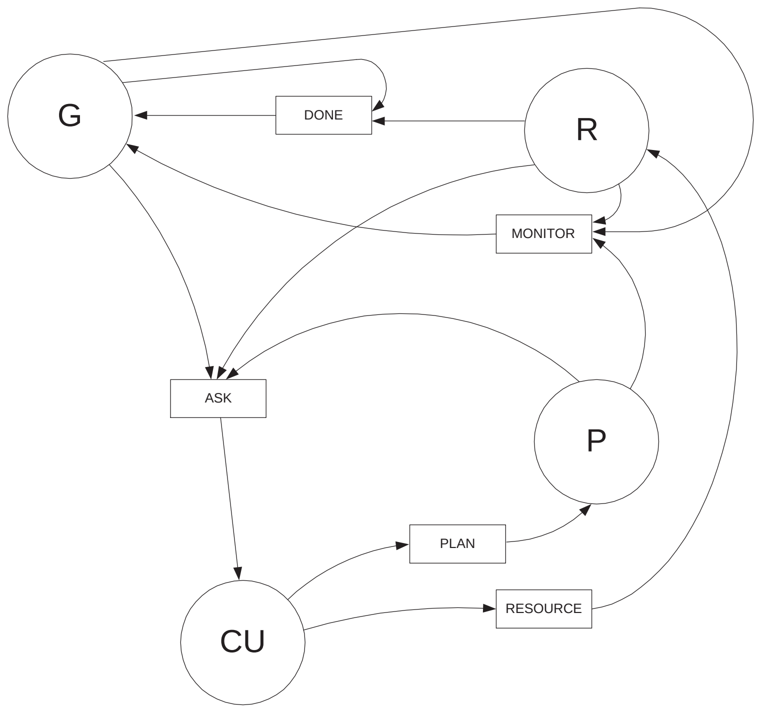

Fig. 5. The goal manager module

as “allocated” is not the same as consuming a resource (which would be denoted by the deletion of the resource), and that once again the bridge rule deletes the original message from the communication unit.

The goal manager is rather more complex than either of the previous modules we have discussed, as is immediately clear from Figure 5 which shows the modules it contains, and the bridge rules which connect them. These modules are:

– The communication unit (CU).
– The plan list unit (P): this contains a list of plans the execution of which is currently being monitored.
– The goal manager unit (G): this is the heart of the module, and ensures that the necessary sub-goaling is carried out.
– The resource list module (R): this contains a list of the resources being used as part of plans which are currently being executed.

The bridge rules relating these units are as follows. The first two bridge rules handle incoming information from the communication unit:

$$\mathsf{RESOURCE} = \frac{CU > answer(Self/RM, Self/GM, have(Self, Z), \{\})}{R : Z}$$

$$\mathsf{PLAN} = \frac{CU > answer(Self/PL, Self/GM, goal(Z), \{P\})}{P : plan(Z, P)}$$

The first of these, RESOURCE, looks for messages from the resource manager reporting that the agent has possession of some resource. When such a message arrives, the goal manager adds a formula representing the resource to its resource list module. The second bridge rule PLAN does much the same for messages from the plan library reporting the existence of a plan—such plans are written to the plan library. There is also a bridge rule ASK which generates messages for other modules:

$$\mathsf{ASK} = \frac{\begin{array}{c} G : goal(X), \\ G : not(done(X)), \\ R : not(X), \\ P : not(plan(X, Z)) \\ G : not(done(ask(X))), \end{array}}{\begin{array}{c} CU : ask(Self/G, Self/all, goal(X), \{\}), \\ G : done(ask(X)) \end{array}}$$

If the agent has the goal to achieve X, and X has not been achieved, nor is X an available resource (and therefore in the R unit), nor is there a plan to achieve X, and X has not already been requested from other modules, then X is requested from other modules and this request is recorded. The remaining bridge rules are:

$$\mathsf{MONITOR} = \frac{\begin{array}{c} G : goal(X), \\ R : not(X), \\ P : plan(X, P) \end{array}}{G : monitor(X, P)}$$

$$\mathsf{DONE} = \frac{\begin{array}{c} G : goal(X), \\ R : X \end{array}}{G : done(X)}$$

The MONITOR bridge rule takes a goal X and, if there is no resource to achieve X but there is a plan to obtain the resource, adds the formula $monitor(X, P)$ to the G unit, which has the effect of beginnning the search for the resources to carry out the plan. The DONE bridge rule identifies that a goal X has been achieved when a suitable resource has been allocated.

4.3 Specifications of the units

Having identified the individual units within each module, and the bridge rules which connect the units, the next stage of the specification is to identify the logics present within the various units, and the theories which are written in those logics. For this agent most of the units are simple containers for atomic formulae. In contrast, the G unit contains a theory which controls the execution of plans. The relevant formulae are:

$$\begin{aligned} monitor(X, P) &\rightarrow assert\_subgoals(P) \\ monitor(X, P) &\rightarrow prove(P) \\ monitor(X, P) \wedge proved(P) &\rightarrow done(X) \end{aligned}$$

$$assert\_subgoals(\bigwedge_i Y_i) \rightarrow \bigwedge_i goal(Y_i)$$

$$prove(X \wedge \bigwedge_i Y_i) \wedge done(X) \rightarrow prove(\bigwedge_i Y_i)$$

$$\bigwedge_i done(Y_i) \rightarrow proved(\bigwedge_i Y_i)$$

The $monitor$ predicate forces all the conjuncts which make up its first argument to be goals (which will be monitored in turn), and kicks off the "proof" of the plan which is its second argument[8]. This plan will be a conjunction of actions, and as each is "done" (a state of affairs achieved through the allocation of resources by other bridge rules), the proof of the next conjunct is sought. When all have been "proved", the relevant goal is marked as completed.

The specification as presented so far is generic—it is akin to a class description for a class of autistic home improvement agents. To get a specific agent we have to "program" it by giving it information about its initial state. For our particular example there is little such information, and we only need to add formulae to three units. The plan repository holds a plan for hanging pictures using hammers and nails:

$$\begin{aligned} &S : plan(hangPicture(X),\\ &\quad have(X, picture) \wedge have(X, nail) \wedge have(X, hammer)) \end{aligned}$$

The resource repository holds the information that the agent has a picture, nail and a hammer:

$$\begin{aligned} &R : Resource(picture, free)\\ &R : Resource(nail, free)\\ &R : Resource(hammer, free) \end{aligned}$$

Finally, the goal manager contains the fact that the agent has the goal of hanging a picture:

$$G : goal(hangPicture(Self))$$

With this information, the specification is complete.

4.4 The agent in action

When the agent is instantiated with this information and executed, we get the following behaviour. The goal manager unit, which has the goal of hanging a picture, does not have the resources to hang the picture, and has no information on how to obtain them. It therefore fires the ASK bridge rule to ask other modules for input, sending message

[8] Given our relaxed view of planning, this "proof" consists of showing the pre-conditions of the plan can be met.

$ask(Self/GM, Self/all, goal(hangPicture(Self)), \{\})$	(GM1)
$answer(Self/PL, Self/GM, goal(hangPicture(Self)),$ $\quad \{have(Self, picture) \wedge have(Self, nail) \wedge have(Self, hammer)\})$	(PL1)
$ask(Self/GM, Self/all, goal(have(Self, picture)), \{\})$	(GM2)
$ask(Self/GM, Self/all, goal(have(Self, nail)), \{\})$	(GM3)
$answer(Self/RM, Self/GM, have(Self, picture), \{\})$	(RM1)
$ask(Self/GM, Self/all, goal(have(Self, hammer)), \{\})$	(GM4)
$answer(Self/RM, Self/GM, have(Self, nail), \{\})$	(RM2)
$answer(Self/RM, Self/GM, have(Self, hammer), \{\})$	(RM3)

Table 1. The inter-module messages

GM1 (detailed in Table 1). When this message reaches the plan library, the bridge rule GET_PLAN is fired, returning a plan (PL1). This triggers the bridge rule PLAN in the goal manager, adding the plan to its P unit. This addition causes the MONITOR bridge rule to fire. This, along with the theory in the G unit, causes the goal manager to realise that it needs a picture, hammer and nail, and to ask for these (GM2, GM3, GM4). As each of these messages reaches the resource manager, they cause the ALLOCATE rule to fire, identifying the resources as being allocated, and generating messages back to the goal manager (RM1, RM2, RM3). These resources cause the RESOURCE bridge rule in the goal manager to fire and the resources to be added to the resource list, R. The addition of the resouces is all that is required to complete the plan of hanging a picture, and the bridge rule DONE fires, adding the formulae $done(have(Self, picture))$, $done(have(Self, hammer))$ and $done(have(Self, nail))$ to the G unit. The theory in G then completes execution.

The messages passed between modules are represented in pictorial form in Figure 6—each row in the diagram identifies one module, time runs from left to right, and the diagonal lines represent the transfer of messages between modules.

5 Related Work

There are two main strands of work to which ours is related—work on executable agent architectures and work on multi-context systems. As mentioned above, most previous work which has produced formal models of agent architectures, for example dMARS [13], Agent0 [22] and GRATE* [14], has failed to carry forward the clarity of the specification into the implementation—there is a leap of faith required between the two. Our work, on the other hand, maintains a clear link between specification and implementation through the direct execution of the specification as exemplified in our running example. This relation to direct execution also distinguishes our work from that on modelling agents in Z [6], since it is not yet possible to directly execute a Z specification. It is possible to animate specifications, which makes it possible to see what would happen if the specification were executed, but animating agent specifications is some way from providing operational agents. Our work also differs from that which aims to describe the operational semantics of agent architectures using the π-calculus [8], since our models have a declarative rather than an operational semantics.

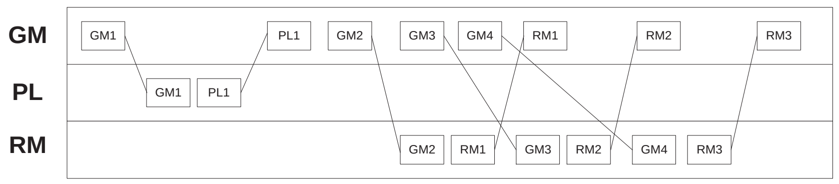

Fig. 6. An execution trace for the agent

More directly related to our work is that on DESIRE and Concurrent MetateM. DESIRE [3, 25] is a modelling framework originally conceived as a means of specifying complex knowledge-based systems. DESIRE views both the individual agents and the overall system as a compositional architecture. All functionality is designed as a series of interacting, task-based, hierarchically structured components. Though there are several differences, from the point of view of the proposal advocated in this paper, we can see DESIRE's *tasks* as modules and *information links* as bridge rules. In our approach there is no an explicit task control knowledge of the kind found in DESIRE. There are no entities that control which units, bridge rules or modules should be activated nor when and how they are activated. Also, in DESIRE the communication between tasks is carried out by the information links that are wired-in by the design engineer. Our inter-module communication is organized as a bus and the independence between modules means new ones can be added without modifying the existing structures. Finally the communication model in DESIRE is based on a one-to-one connection between *tasks*, in a similar way to that in which we connect units inside a module. In contrast, our communication between modules is based on a multicast model.

Concurrent MetateM defines concurrent semantics at the level of single rules [9, 27]. Thus an agent is basically a set of temporal rules which fire when their antecedents are satisfied. Our approach does not assume concurrency within the components of units, rather the units themselves are the concurrent components of our architectures. This means that our model has an inherent concurrent semantics at the level of the units and has no central control mechanism. Though our exemplar uses what is essentially first order logic (albeit a first order logic labelled with arguments), we could use any logic we choose—we are not restricted to a temporal logic as in MetateM.

There are also differences between our work and previous work on using multi-context systems to model agents' beliefs. In the latter [11], different units, all containing a belief predicate, are used to represent the beliefs of the agent and the beliefs of all the acquaintances of the agent. The nested beliefs of agents may lead to tree-like structures of such units (called *belief contexts*). Such structures have then been used to solve problems like the three wise men [5]. In our case, however, any nested beliefs would typically be included in a single unit or module. Moreover we provide a more comprehensive formalisation of an autonomous agent in that we additionally show how capabilities other than that of reasoning about beliefs can be incorporated into the architecture. In this latter respect this paper extends the work of [20] with the idea of modules which links the approach more strongly with the software engineering tradition.

6 Conclusions

This paper has proposed a general approach to defining agent architectures. It provides a means of structuring logical specifications of agents in a way which makes them directly executable. This approach has a number of advantages. Firstly it bridges the gap between the specification of agents and the programs which implement those specifications. Secondly, the modularity of the approach makes it easier to build agents which are capable of carrying out complex tasks such as distributed planning. From a software engineering point of view, the approach leads to architectures which are easily expandable, and have re-useable components.

From this latter point of view, our approach suggests a methodology for building agents which has similarities with object-oriented design [2]. The notion of inheritance can be applied to groups of units and bridge rules, modules and even complete agents. These elements could have a general design which is specialized to different and more concrete instances by adding units and modules, or by refining the theories inside the units of a generic agent template. However, before we can develop this methodology, there are some issues to resolve. Firstly there is the matter of the semantics of the comsuming conditions and time-outs in bridge rules. Secondly, there is the question of how to handle nested hierachies of modules—something which is essential if we are to develop really complex agents.

Acknowledgments This work has been supported by the Spanish CICYT project SMASH, TIC96-1038-C04001 and the UK EPSRC project Practical Negotiation for Electronic Commerce GR/M07076. Jordi Sabater enjoys a CSIC scholarship “Formación y Especialización en Líneas de Investigación de Interés Industrial” in collaboration with C.S.M (Consorci Sanitari de Mataró).

References

1. M. Benerecetti, A. Cimatti, E. Giunchiglia, F. Giunchiglia, and L. Serafini. Formal specification of beliefs in multi-agent systems. In J. P. Müller, M. J. Wooldridge, and N. R. Jennings, editors, *Intelligent Agents III*, pages 117–130. Springer Verlag, Berlin, 1997.
2. G. Booch. *Object-oriented analysis and design with application*. Addison Wesley, Wokingham, UK, 1994.
3. F. M. T. Brazier, B. M. Dunin-Keplicz, N. R. Jennings, and J. Treur. Formal specification of multi-agent systems. In *Proceedings of the 1st International Conference on Multi-Agent Systems*, pages 25–32, 1995.
4. P. Busetta, N. Howden, Ronnquist R, and A. Hodgson. Structuring BDI agents in functional clusters. In N. R. Jennings and Y Lespérance, editors, *Intelligent Agents VI*. Springer Verlag, Berlin, 1999.
5. A. Cimatti and L. Serafini. Multi-agent reasoning with belief contexts: The approach and a case study. In M. J. Wooldridge and N. R. Jennings, editors, *Intelligent Agents*, pages 62–73. Springer Verlag, Berlin, 1995.
6. M. d’Inverno, D. Kinny, M. Luck, and M. Wooldridge. A formal specification of dMARS. In M. P. Singh, A. S. Rao, and M. Wooldridge, editors, *Intelligent Agents IV*, pages 155–176. Springer Verlag, Berlin, 1998.

7. D. Dubois and H. Prade. *Possibility Theory: An Approach to Computerized Processing of Uncertainty*. Plenum Press, New York, NY, 1988.
8. J. Ferber and O. Gutknecht. Operational semantics of a role-based agent architecture. In N. R. Jennings and Y Lespérance, editors, *Intelligent Agents VI*. Springer Verlag, Berlin, 1999.
9. M. Fisher. Representing abstract agent architectures. In J. P. Müller, M. P. Singh, and A. S. Rao, editors, *Intelligent Agents V*, pages 227–242. Springer Verlag, Berlin, 1998.
10. D. Gabbay. *Labelled Deductive Systems*. Oxford University Press, Oxford, UK, 1996.
11. F. Giunchiglia. Contextual reasoning. In *Proceedings of the IJCAI Workshop on Using Knowledge in Context*, 1993.
12. F. Giunchiglia and L. Serafini. Multilanguage hierarchical logics (or: How we can do without modal logics). *Artificial Intelligence*, 65:29–70, 1994.
13. F. F. Ingrand, M. P. Georgeff, and A. S. Rao. An architecture for real-time reasoning and system control. *IEEE Expert*, 7(6):34–44, 1992.
14. N. R. Jennings. Controlling cooperative problem solving in industrial multi-agent systems using joint intentions. *Artificial Intelligence*, 75:195–240, 1995.
15. N. R. Jennings. Agent-based computing: Promise and perils. In *Proceedings of the 16th International Joint Conference on Artificial Intelligence*, pages 1429–1436, 1999.
16. J. J. Meyer. Agent languages and their relationship to other programming paradigms. In J. P. Müller, M. P. Singh, and A. S. Rao, editors, *Intelligent Agents V*, pages 309–316. Springer Verlag, Berlin, 1998.
17. P. Noriega and C. Sierra. Towards layered dialogical agents. In J. P. Müller, M. J. Wooldridge, and N. R. Jennings, editors, *Intelligent Agents III*, pages 173–188, Berlin, 1996. Springer Verlag.
18. S. Parsons and P. Giorgini. An approach to using degrees of belief in BDI agents. In B. Bouchon-Meunier, R. R. Yager, and L. A. Zadeh, editors, *Information, Uncertainty, Fusion*. Kluwer, Dordrecht, 1999.
19. S. Parsons and N. R. Jennings. Negotiation through argumentation—a preliminary report. In *Proceedings of the International Conference on Multi Agent Systems*, pages 267–274, 1996.
20. S. Parsons, C. Sierra, and N. R. Jennings. Agents that reason and negotiate by arguing. *Journal of Logic and Computation*, 8(3):261—292, 1998.
21. J. Sabater, C. Sierra, S. Parsons, and N. R. Jennings. Engineering executable agents using multi-context systems. Technical report, IIIA, 1999.
22. Y. Shoham. Agent-oriented programming. *Artificial Intelligence*, 60:51–92, 1993.
23. C. Szyperski. *Component Software*. Addison Wesley, Wokingham, UK, 1998.
24. S. R. Thomas. The PLACA agent programming language. In M. J. Wooldridge and N. R. Jennings, editors, *Intelligent Agents*, pages 355–370. Springer Verlag, Berlin, 1995.
25. J. Treur. On the use of reflection principles in modelling complex reasoning. *International Journal of Intelligent Systems*, 6:277–294, 1991.
26. D. Weerasooriya, A. Rao, and K. Rammamohanarao. Design of a concurrent agent-oriented language. In M. J. Wooldridge and N. R. Jennings, editors, *Intelligent Agents*, pages 386–402. Springer Verlag, Berlin, 1995.
27. M. Wooldridge. A knowledge-theoretic semantics for Concurrent MetateM. In J. P. Müller, M. J. Wooldridge, and N. R. Jennings, editors, *Intelligent Agents III*, pages 357–374. Springer Verlag, Berlin, 1996.
28. M. Wooldridge. Agent-based software engineering. *IEE Proceedings on Software Engineering*, 144:26–37, 1997.
29. M. J. Wooldridge and N. R. Jennings. Intelligent agents: Theory and practice. *The Knowledge Engineering Review*, 10:115–152, 1995.