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 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 with an example architecture capable of argumentation-based negotiation.

1 Introduction

Agent-based computing is fast emerging as a new paradigm for engineering complex, distributed systems [13, 26]. 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 [14, 22, 24]). 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, demand for unambiguous specifications of architectures increases and there is a greater need to verify implementations of architectures. To this end, a range of techniques have been used to formally specify agent architectures (eg Concurrent MetateM [7, 25], DESIRE [3, 23] and Z [5]). 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 propose the use of *multi-context systems* [10] 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 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 [21]) 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).

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 [10] 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. 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, 4].

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.

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 formal model

Using a multi-context approach, an agent architecture consists of the following components [10, 15]:

– *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 [18]), 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 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 rules of inference 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:

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

[1] See [27] for a discussion of the relative merits of logic-based and non logic-based approaches to specifying and building agent architectures.

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.

2.2 The execution model

All the elements of the model, both units and bridge rules, work concurrently. The execution of each unit is a non-terminating, deductive process. 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.

In order to better handle the dynamics of reasoning, we 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.

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:

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

means that t units of time after theory in unit u_1 gets formula ψ, 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).

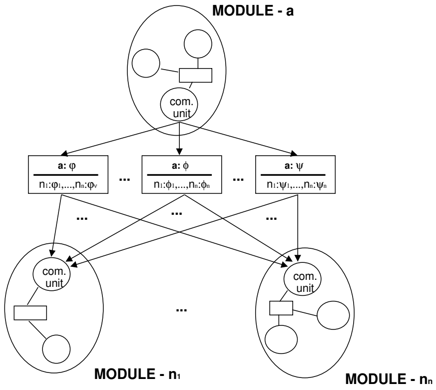

Fig. 1. Module inter-connection (from a's perspective only)

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 [18]). 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 added 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.

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 degree of importance with each message. This value is assigned to the message by the communication unit of the module that sends it out. Degrees of importance belong to $[0, 1]$ (maximum importance is 1 and minimum is 0). The meaning of the degree of importance 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 importance might be preferred, or the different degrees of importance 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 [16]). Note that degrees of importance are used only in *inter-module* messages. If the units within a module need to know the level of importance of a message, the communication unit could, for instance, translate the degree of importance into a degree of belief and vice versa.

3.2 Formalising 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 $a[/m]^{*2}$ 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.
- φ is the content of the message.
- G is the set of *grounds* of φ, a set of formulae which justify it. It has the form: $\{\{\Gamma_1 \vdash \varphi_1\} \ldots \{\Gamma_n \vdash \varphi_n\}\}$ where Γ has the form $\gamma[,\gamma]^*$ (γ being a formula) and φ_i is a formula with $\varphi_n = \varphi$ [3].
- $\psi \in [0, 1]$ is the degree of importance of 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:

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

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

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

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

[3] In other words, G is exactly the set of grounds of the argument for φ [18]. 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 [8] this approach is a generalisation of classical logic—there is nothing to stop the same approach being used when messages are just formuale in classical logic.

and

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

while module *b* sends

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

As the combined degree of importance (by using max and treating the degree of importance as a degree of possibility [6]) of the negated literal is higher than the degree of importance 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 [16].)

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

4 The home improving agents

To illustrate the power of the approach we have advocated, this section details an agent architecture we have designed and built using the modular multi-context approach. This architecture enables the agents to negotiate using argumentation techniques. The particular scenario we consider is that of two home improvement agents [17, 18]. The first agent (named A) wants to hang a picture and the second one (named B) wants to hang a mirror. Both agents have the same architecture, but have distinct initial knowledge and resources (described below). As will become apparent, neither of them can succeed in their goals without the help of the other, so they engage in negotiation in order to exchange knowledge and resources. This negotiation is based on argumentation [17, 18] and so the agents are involved in rebutting each other's arguments (finding arguments which oppose them) and undercutting each other's arguments (finding arguments which attack their grounds). Note that in the example below conflicts between modules do not appear, hence the reference to degrees of importance in the formulae have been removed to improve the readability.

4.1 Inter-agent and inter-module messages

This section lists the types of message that are passed between modules and between agents. These are as follows:

– Inter-agent messages
  - **Request:** a query to another agent.
  - **Reply:** an answer to an inter-agent request.

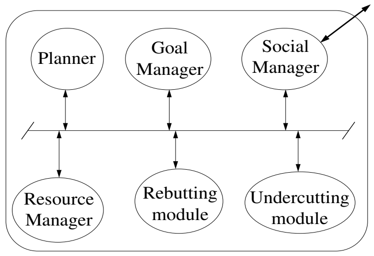

Fig. 2. The topology of the modules for the home improving agents. The arrow in the top-right corner denotes a connection to other agents.

- Inter-module messages
  - **Ask:** a request to another module.
  - **Answer:** an answer to an inter-module request.
  - **Inform:** information volunteered (rather than sent as an answer to a specific request).
  - **Controvert:** a request for something that can be used to refute the content of the message or the argument supporting that content.
  - **Rebut:** Contains an argument which rebuts the content of the message.

We also need to define the kinds of content that messages of these types can have. Given a set of agent names AN, and with $AN' = AN \cup \{Self\}$.

- $Goal(X, Z)$: $X \in AN'$ and Z is a string describing an action. This denotes Agent X has the goal Z.
- $Have(X, Z)$: $X \in AN'$ and Z is the name of an object. This denotes Agent X has possession of Z.
- $Give(X, Y, Z)$: $X, Y \in AN'$ and Z is the name of an object. This denotes that Agent X gives Z to agent Y.
- $GetPlan(\varphi, \psi)$: φ is a formula and ψ a set of formulae. This denotes the fact that the module sending the message is asking for a plan to achieve φ. That plan must satisfy the restrictions ψ[4].

4.2 A module level description

The modules which make up our home improving agents are shown in Figure 2. Note that the multicast bridge rules are drawn as a single bus connecting all the modules.

- *Planner (P):* This module can generate plans in order to achieve a given goal taking into account any restrictions placed upon it.
- *Goal Manager (GM):* This module is responsible for ensuring that the agent's goals are satisfied. It asks the planner module for plans that can achieve the agent's goals, and monitors their execution.

[4] So that, for instance, one agent can look for a plan subject to the restriction that a particular resource is not used.

- *Social Manager (SM):* This module monitors all the agent's exchanges with other agents. It is the dialogical channel between the agent and the environment (i.e. the other agents). Generally speaking, this module tries to help the agent's acquaintances (satisfying their requests if possible) but it also takes into account the agent's own goals and preferences. This module also manages the agent's social relations, keeping track of which agents are reliable [16], which ones have not collaborated after a request has been made, and so on.
- *Resource Manager (RM):* This module keeps track of the agent's resources and its knowledge about the resources of its acquaintances.
- *Rebutting module (RE):* This module finds arguments to rebut a proposition (from the agent's perspective).
- *Undercutting module (UM):* This module examines the grounds of an argument to find incorrect assumptions or beliefs (again from the agent's perspective).

In addition to the generic domain knowledge encoded in these modules, each agent has some specific domain information:

AGENT-A

- Goal Manager: Has the goal *Goal(A,HangPicture)*.
- Planner: Knows how to hang pictures using nails and mirrors using screws.

$$
\begin{aligned}
&Have(X, Picture), Have(X, Hammer),\\
&\qquad Have(X, Nail) \rightarrow Goal(X, HangPicture)\\
&Have(X, Screw), Have(X, Screwdriver),\\
&\qquad Have(X, Mirror) \rightarrow Goal(X, HangMirror)
\end{aligned}
$$

- Resource Manager: Knows that a picture, hammer, screw and screwdriver are available resources for A, and has the information that agent B has a nail.

AGENT-B

- Goal Manager: Has the goal *Goal(B,HangMirror)*.
- Planner: Knows how to hang a mirror using a nail:

$$Have(X, Mirror), Have(X, Hammer), Have(X, Nail) \rightarrow Goal(X, HangMirror)$$

- Resource Manager: Knows that the mirror and the nail are available to B.

4.3 Argumentation in Action

This sub-section describes the operation of the two home improvement agents; more details of the precise interchanges within and between the agents are given in the annex. Table 1 provides a graphical and more abstract view of these interactions. The notation we use should be read as follows: ($GM1_{A,B}$) refers to the message (numbered one) generated by module GM in agents A and B, and ($RM3_A$) refers to a message (numbered three) generated by A's RM module.

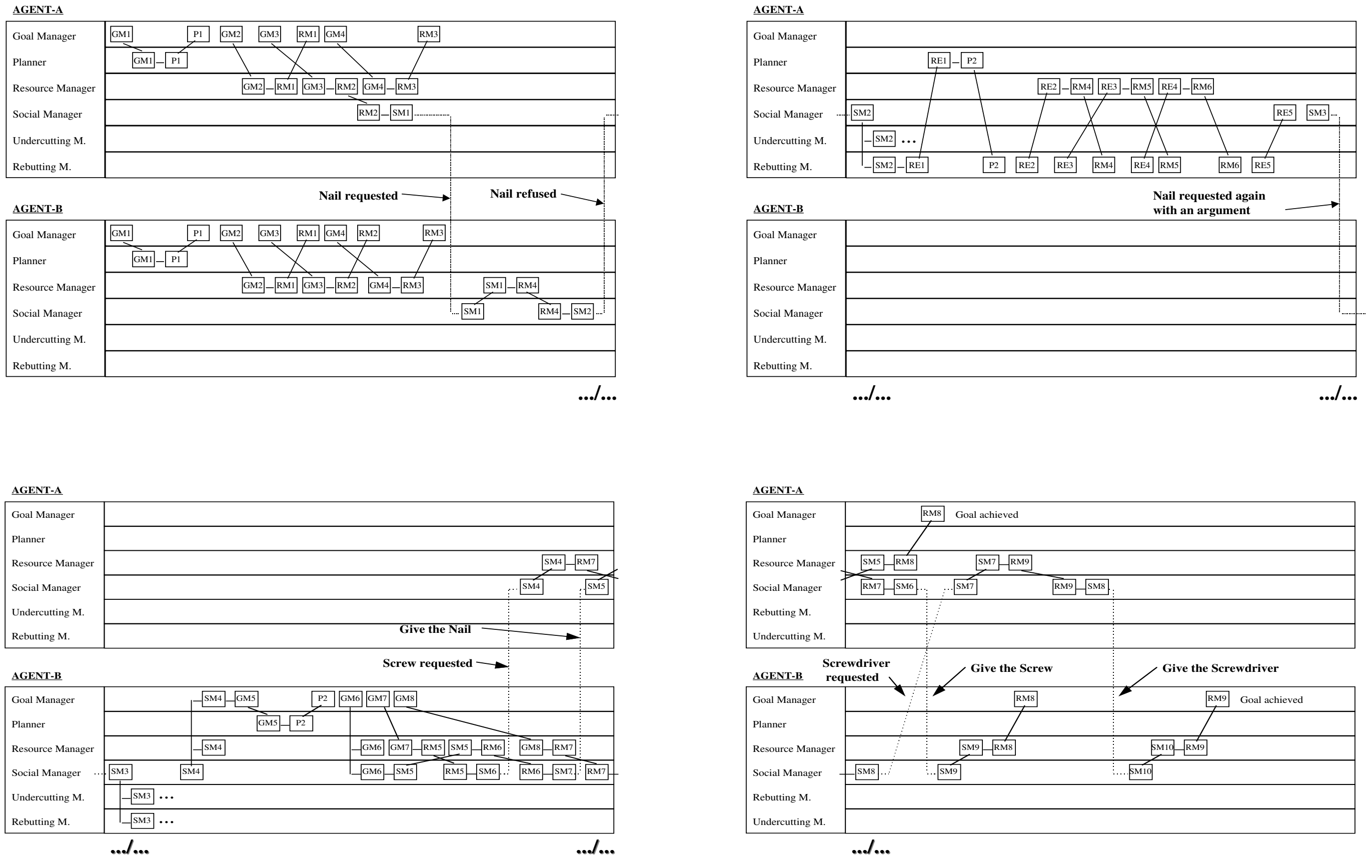

Table 1. A trace of the messages passed between modules and agents.

When the two agents start, they both have a goal to achieve: agent A wants to hang a picture and agent B wants to hang a mirror. To determine how to achieve these goals, the respective Goal Managers send queries to their respective Planning modules (GM$1_{A,B}$). Agent A's planner indicates it needs a hammer, a nail and a picture to hang a picture (P1_A), while agent B's plan indicates it needs a hammer, a nail and a mirror to hang a mirror (P1_B). Following these plans, the two Goal Managers attempt to obtain the necessary objects by asking the other modules for them (GM$2_{A,B}$, GM$3_{A,B}$, GM$4_{A,B}$). Only the respective Resource Managers know whether the objects are available. B's Resource Manager can allocate the nail (RM1_B) and the mirror (RM3_B), but it has no information about the hammer (RM2_B). At this point, B cannot continue with its plan and it has no alternatives available. Meanwhile, A's Resource Manager allocates the hammer (RM1_A) and the picture (RM3_A). Agent A does not have the nail, but its Resource Manager knows that agent B has one. Hence, A's Resource Manager suggests asking B for the nail (RM2_A). A's Social Manager actually sends out the request (SM1_A).

Agent B's Social Manager receives A's request and tries to satisfy it by asking the other modules if A can have the nail (SM1_B). The Resource Manager responds and indicates that A cannot have the nail (RM4_B) because it is already part of one of its plans (this information comes from the argument GM2_B). The Social Manager sends a reply to agent A (SM2_B) indicating that it cannot have the nail; it also sends the accompanying argument by means of an explanation.

When A's Social Manager receives B's reply, it asks the other modules if they can generate a counter argument against B's decision (SM2_A). This sets the Undercutting and Rebutting modules to work. The Undercutting module finds nothing wrong (B's argument is correct from A's point of view). However, the Rebutting module finds a plan to hang a mirror without using a nail (RE1_A, P2_A). A now has to determine whether this new plan is acceptable to B. In order to do this, the Rebutting module asks the Resource Manager if the screw, screwdriver and mirror can be allocated to B (RE2_A, RM4_A, RE3_A, RM5_A, RE4_A). The screw and screwdriver can be allocated, however the mirror cannot (RM6_A) (because A does not have it). This shortcoming is not serious enough to invalidate the plan, so the Rebutting module sends the new plan to the social manager (RE5_A). Since this is the only counter argument available to A, the Social Manager sends it to agent B (SM3_A).

Upon receipt of A's new proposal, B sees if it can construct a counter argument (SM3_B). However neither the Undercutting nor the Rebutting module can find anything wrong with this new argument. Hence, after a suitable delay (implemented using time out bridge rules) the Social Manager accepts the argument as valid and informs the other modules about its content (SM4_B). The Goal Manager checks the new plan with the planner (GM5_B) and, seeing that the old plan is blocked, changes it for the new one (P2_B). This change involves retracting the nail allocation (GM6_B) and asking A for the screw (GM7_B, RM5_B, SM6_B) and the screwdriver (GM8_B, RM7_B, SM8_B). Meanwhile, the Social Manager repeats A's request for the nail (SM5_B) and this time no counter arguments are generated (RM6_B). Hence the request is granted. It is important to note that that this second nail request is made only after the Social Manger sees a change in the agent's state.

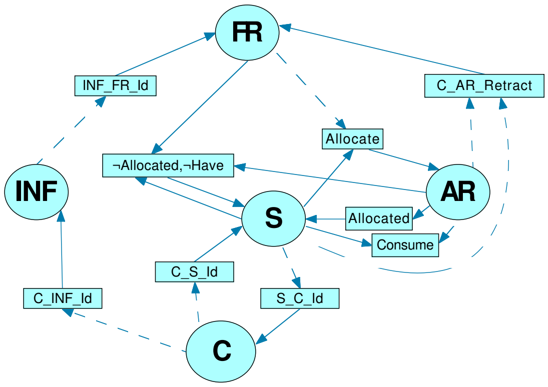

Fig. 3. The units in the Resource Manager module.

From this stage, all subsequent requests for resources are granted and the two agents proceed to execute their plans and ultimately achieve their goals.

4.4 A module in detail

This sub-section examines in more detail the operation of the Resource Management module of our agent architecture. This module is composed of five units (figure 3):

- Communication unit (C): the point of contact with the other modules. It also knows what kinds of message this module can deal with.
- Information unit (INF): contains all the information about the resources owned by the other agents. It is also capable of extracting this kind of information from the various arguments that the module may be sent.
- Server unit (S): acts as an interface to translate requests from the communication unit to the Free and Allocated resource units (and vice-versa).
- Free resource unit (FR): container for all the agent's free resources.
- Allocated resources unit (AR): container for all the agent's allocated resources.

These units are linked by a number of bridge rules:

$$C\_INF\_Id = \frac{C > Inform(Self/x, Self/RM, Content(\varphi), G)}{INF : Inform(Self/x, Self/RM, Content(\varphi), G)}$$

$$C\_S\_Id = \frac{C > Ask(Self/x, Self/RM, \varphi, G)}{S : Ask(Self/x, \varphi, G)}$$

$$S\_C\_Id = \frac{S > Answer(Self/x, \varphi, G)}{C : Answer(Self/RM, Self/x, \varphi, G)}$$

$$INF\_FR\_Id = \frac{INF > B(Self, Have(Self, Z))}{FR : Have(Self, Z)}$$

$$C\_AR\_Retract = \frac{\begin{array}{c} AR > Have(X, Z), \\ S > Ask(Self/x, Self/RM, Retract(Have(X, Z)), G) \end{array}}{FR : Have(X, Z)}$$

$$Allocate = \frac{S : Allocate(Z, Y, G), FR > Have(X, Z)}{AR : Allocated(Z, Y, G)}$$

$$Allocated = \frac{AR : Allocated(Z, Y, G)}{S : Allocated(Z, Y, G)}$$

$$Consume = \frac{S : Consume(Z, Y)}{AR > Allocated(Z, Y, G)}$$

$$\neg Have = \frac{\begin{array}{c} S : Allocate(Z, Y, G1), RF : \neg Have(X, Z), \\ AR : \neg Allocated(T, Z, G2) \end{array}}{S : \neg Have(X, Z)}$$

To see how this module operates, we will examine what happens in A's Resource Manager module between the times when the communication unit receives the SM4 message and it generates RM7. At this time, FR has the formulae $Have(A, Screw)$ and $Have(A, Screwdriver)$ and AR has the formulae $Allocated(Hammer, A, A^{GM2})$ and $Allocated(Picture, A, A^{GM4})$. The event trace is as follows:

1. **C** receives the SM4 message.
2. Bridge rule **S_C_Id** is fired: (*Ask(Self/SM,Give(A,B,Screw),*$\{\}$*)* is added to **S**).
3. The inference engine of **S** generates the formula: *Allocate(Screw,B,*$\{\}$*)*.
4. As **FR** contains the formula *Have(A,Screw)*, bridge rule **Allocate** is fired; *Allocated(Screw,B,*$\{\}$*)* is added to **AR** and *Have(A,Screw)* is removed from **FR**. Notice how we use a consuming condition to model the state change of an object from free to allocated.
5. Bridge rule **Allocated** is fired: *Allocated(Screw,B,*$\{\}$*)* is added to **S**.
6. The inference engine of **S** generates: *Consume(Screw,B)* and *Answer(Self/SM,Give (A,B,Screw))*.
7. Bridge rules **C_S_Id** and **Consume** are fired: *Allocated(Screw,B,*$\{\}$*)* is removed from **AR** and *Answer(Self/RM,Self/SM,Give(A,B,Screw),*$\{\}$*)* is added to **C**.
8. A multicast bridge rule distributes the message from **C** (RM7).

Similar descriptions of the other modules which make up the agents may be found in [19]. This complete specification can be directly executed, broadly speaking by considering each unit to be a concurrent theorem prover and the bridge rules to be rules of inference connecting these theorem provers (again the implementation which makes this possible is described in [19]).

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 [11], Agent0 [20] and GRATE* [12], 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 [5], since it is not yet possible to directly execute a Z specification[5].

More directly related to our work is that on DESIRE and Concurrent MetateM. DESIRE [3, 23] 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 [7, 25]. 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 [9], 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 [4]. 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 [18] 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 di-

[5] 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.

rectly 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. Initial thoughts on this methodology are contained in [19] which also contains details of our implementation of these ideas.

Acknowledgments This work has been supported by the Spanish CICYT project SMA-SH, 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. 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.
5. 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.
6. D. Dubois and H. Prade. *Possibility Theory: An Approach to Computerized Processing of Uncertainty*. Plenum Press, New York, NY, 1988.
7. 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.
8. D. Gabbay. *Labelled Deductive Systems*. Oxford University Press, Oxford, UK, 1996.
9. F. Giunchiglia. Contextual reasoning. In *Proceedings of the IJCAI Workshop on Using Knowledge in Context*, 1993.
10. F. Giunchiglia and L. Serafini. Multilanguage hierarchical logics (or: How we can do without modal logics). *Artificial Intelligence*, 65:29–70, 1994.
11. 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.

12. N. R. Jennings. Controlling cooperative problem solving in industrial multi-agent systems using joint intentions. *Artificial Intelligence*, 75:195–240, 1995.
13. N. R. Jennings. Agent-based computing: Promise and perils. In *Proceedings of the 16th International Joint Conference on Artificial Intelligence*, 1999.
14. 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.
15. 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.
16. 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.
17. 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.
18. 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.
19. J. Sabater, C. Sierra, S. Parsons, and N. R. Jennings. An approach to engineering executable agents. Technical report, IIIA, 1999.
20. Y. Shoham. Agent-oriented programming. *Artificial Intelligence*, 60:51–92, 1993.
21. C. Szyperski. *Component Software*. Addison Wesley, Wokingham, UK, 1998.
22. 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.
23. J. Treur. On the use of reflection principles in modelling complex reasoning. *International Journal of Intelligent Systems*, 6:277–294, 1991.
24. 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.
25. 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.
26. M. Wooldridge. Agent-based software engineering. *IEE Proceedings on Software Engineering*, 144:26–37, 1997.
27. M. J. Wooldridge and N. R. Jennings. Intelligent agents: Theory and practice. *The Knowledge Engineering Review*, 10:115–152, 1995.

ANNEX

This annex contains a description of all the messages depicted in Table 1. Note that:

1. Inter-agent messages are emboldened.
2. A^{P1} refers to the grounds supporting message $P1$ of agent A (and similarly for agent B);
3. A_i^{P1} refers to the ith step of the grounds in message $P1$ of agent A (and similarly for agent B);
4. The predicate "Retract" denotes that an action (in this example an illocutionary action) is retracted; and
5. The predicate "Content" is a mechanism by which an agent passes the information within a message to all its modules—it is thus a way of referring to the content of a message rather than the message itself.

AGENT-A

Ask(Self/GM,Self/P,GetPlan(Goal(Self,HangPicture),{}),{})	(GM1)
Answer(Self/P,Self/GM,Goal(Self,HangPicture),A^{P1})	(P1)
A^{P1} = { {Have(A,Hammer), Have(A,Nail), Have(A,Picture), Have(X,Hammer), Have(X,Nail), Have(X,Picture) $\rightarrow$ Goal(X,HangPicture)} $\vdash$ Goal(A,HangPicture) }	
Ask(Self/GM,Self/All,Have(Self,Hammer),A^{GM2})	(GM2)
A^{GM2} = { $\vdash$ Goal(A,HangPicture); A_1^{P1}; {A_1^{GM2},A_2^{GM2}} $\vdash$ Have(A,Hammer) }	
Ask(Self/GM,Self/All,Have(Self,Nail),A^{GM3})	(GM3)
A^{GM3} = { $\vdash$ Goal(A,HangPicture); A_1^{P1}; {A_1^{GM3},A_2^{GM3}} $\vdash$ Have(A,Nail) }	
Answer(Self/RM,Self/GM,Have(Self,Hammer),{})	(RM1)
Ask(Self/GM,Self/All,Have(Self,Picture),A^{GM4})	(GM4)
A^{GM4} = { $\vdash$ Goal(A,HangPicture); A_1^{P1}; {A_1^{GM4},A_2^{GM4}} $\vdash$ Have(A,Picture) }	
Ask(Self/RM,Self/All,Give(B,Self,Nail),{})	(RM2)
Request(A,B,Give(B,A,Nail),{})	**(SM1)**
Answer(Self/RM,Self/GM,Have(Self,Picture),{}))	(RM3)
Controvert(Self/SM,Self/All,¬Give(B,A,Nail),B^{RM4})	(SM2)
Ask(Self/RE,Self/P,GetPlan(Goal(B,HangMirror),{Have(B,Nail)},{}))	(RE1)
Answer(Self/P,Self/RE,Goal(B,HangMirror),A^{P2})	(P2)
A^{P2} = { { Have(B,Screw), Have(B,Screwdriver), Have(B,Mirror), Have(X,Screw), Have(X,Screwdriver), Have(X,Mirror) $\rightarrow$ Goal(X,HangMirror)} $\vdash$ Goal(B,HangMirror) }	
Ask(Self/RE,Self/All,Have(Self,Screw),{})	(RE2)
Ask(Self/RE,Self/All,Have(Self,Screwdriver),{})	(RE3)
Answer(Self/RM,Self/RE,Have(Self,Screw),{}))	(RM4)
Answer(Self/RM,Self/RE,Have(Self,Screwdriver),{}))	(RM5)
Ask(Self/RE,Self/All,Have(Self,Mirror),{})	(RE4)
Answer(Self/RM,Self/RE,¬Have(Self,Mirror),{}))	(RM6)
Rebut(Self/RE,Self/SM,¬Give(B,A,Nail),A^{RE5})	(RE5)
A^{RE5} = { A_1^{P2}; {Have(A,Screw), Have(A,Screwdriver), Give(X,Y,Z) $\rightarrow$ Have(Y,Z), Give(A,B,Screw), Give(A,B,Screwdriver)} $\vdash$ Have(B,Screw) $\wedge$ Have(B,Screwdriver); $\Gamma_1^{RE5} \cup \Gamma_2^{RE5} \vdash \neg$ Have(B,Nail) }	
Request(A,B,Give(B,A,Nail),A^{RE5})	**(SM3)**
Ask(Self/SM,Self/All,Give(A,B,Screw),{})	(SM4)
Answer(Self/RM,Self/SM,Give(B,A,Screw),{}))	(RM7)
Answer(Self/SM,Self/RM,Give(B,A,Nail),{}	(SM5)
Answer(Self/RM,Self/GM,Have(A,Nail),{})	(RM8)
Reply(A,B,Give(A,B,Screw))	**(SM6)**
Ask(Self/SM,Self/All,Give(A,B,Screwdriver),{})	(SM7)
Answer(Self/RM,Self/SM,Give(B,A,Screwdriver),{}))	(RM9)
Reply(A,B,Give(A,B,Screwdriver))	(SM8)

AGENT-B

Ask(Self/GM,Self/All,GetPlan(Goal(Self,HangMirror),$\{\}$),$\{\}$)	(GM1)
Answer(Self/P,Self/GM,Goal(Self,HangMirror),B^{P1})	(P1)
B^{P1} = { {Have(B,Nail), Have(B,Hammer), Have(B,Mirror), Have(X,Nail), Have(X,Hammer),Have(X,Mirror) $\rightarrow$ Goal(X,HangMirror)} $\vdash$ Goal(B,HangMirror) }	
Ask(Self/GM,Self/All,Have(Self,Nail),B^{GM2})	(GM2)
B^{GM2} = { $\vdash$ Goal(B,HangMirror); B_1^{P1}; $\{B_1^{GM2}$,$B_2^{GM2}\}$ $\vdash$ Have(B,Nail) }	
Ask(Self/GM,Self/All,Have(Self,Hammer),B^{GM3})	(GM3)
B^{GM3} = { $\vdash$ Goal(B,HangMirror); B_1^{P1}; $\{B_1^{GM3}$,$B_2^{GM3}\}$ $\vdash$ Have(B,Hammer) }	
Answer(Self/RM,Self/GM,Have(Self,Nail),$\{\}$)	(RM1)
Ask(Self/GM,Self/All,Have(Self,Mirror),B^{GM4})	(GM4)
B^{GM4} = { $\vdash$ Goal(B,HangMirror); B_1^{P1}; $\{B_1^{GM4}$,$B_2^{GM4}\}$ $\vdash$ Have(B,Mirror) }	
Answer(Self/RM,Self/GM,$\neg$Have(Self,Hammer),$\{\}$)	(RM2)
Answer(Self/RM,Self/GM,Have(Self,Mirror),$\{\}$))	(RM3)
Ask(Self/SM,Self/All,Give(B,A,Nail),$\{\}$)	(SM1)
Answer(Self/RM,Self/SM,$\neg$Give(B,A,Nail),B^{RM4})	(RM4)
B^{RM4} = { B_1^{GM2}; { Give(X,Y,Z) $\rightarrow$ $\neg$ Have(X,Z), Have(B,Nail) } $\vdash$ $\neg$ Give(B,A,Nail) }	
Reply(B,A,$\neg$Give(B,A,Nail),B^{RM4})	**(SM2)**
Controvert(Self/SM,Self/All,Give(B,A,Nail),A^{RE5})	(SM3)
Inform(Self/SM,Self/All,Content(A^{RE5}),$\{\}$)	(SM4)
Ask(Self/GM,Self/P,Goal(B,HangMirror),$\{A_1^{P2}\}$)	(GM5)
Answer(Self/P,Self/GM,Goal(Self,HangMirror),B^{P2})	(P2)
B^{P2} = { { Have(B,Screw), Have(B,Screwdriver), Have(B,Mirror), Have(X,Screw), Have(X,Screwdriver),Have(X,Mirror) $\rightarrow$ Goal(X,HangMirror)} $\vdash$ Goal(B,HangMirror) }	
Ask(Self/GM,Self/All,Retract(Have(B,Nail),$\{\}$)	(GM6)
Ask(Self/GM,Self/All,Have(B,Screw),$\{\}$)	(GM7)
Ask(Self/SM,Self/All,Give(B,A,Nail),$\{\}$)	(SM5)
Ask(Self/GM,Self/All,Have(B,Screwdriver),$\{\}$)	(GM8)
Ask(Self/RM,Self/All,Give(A,Self,Screw),$\{\}$)	(RM5)
Request(B,A,Give(A,B,Screw),$\{\}$)	**(SM6)**
Answer(Self/RM,Self/SM,Give(B,A,Nail),$\{\}$)	(RM6)
Reply(B,A,Give(B,A,Nail),$\{\}$)	**(SM7)**
Ask(Self/RM,Self/All,Give(A,Self,Screwdriver),$\{\}$)	(RM7)
Request(B,A,Give(A,B,Screwdriver),$\{\}$)	**(SM8)**
Inform(Self/SM,Self/RM,Content(Give(A,B,Screw)),$\{\}$)	(SM9)
Answer(Self/RM,Self/GM,Have(B,Screw),$\{\}$)	(RM8)
Inform(Self/SM,Self/RM,Content(Give(A,B,Screwdriver)),$\{\}$)	(SM10)
Answer(Self/RM,Self/GM,Have(B,Screwdriver),$\{\}$	(RM9)