Engineering executable agents using multi-context systems*

Jordi Sabater†, Carles Sierra†, Simon Parsons‡, and Nicholas R. Jennings§

†IIIA - Artificial Intelligence Research Institute
CSIC - Spanish Council for Scientific Research
Campus UAB, 08193 Bellaterra, Catalonia, Spain.
{jsabater,sierra}@iiia.csic.es

‡Department of Computer Science
University of Liverpool
Chadwick Building, Liverpool L69 7ZF
United Kingdom
S.D.Parsons@csc.liv.ac.uk

§Department of Electronics and Computer Science
University of Southampton
Highfield, Southampton SO17 1BJ
United Kingdom
nrj@ecs.soton.ac.uk

December 22, 2000

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 specifying logic-based agents, which is based on the use of multi-context systems, and give examples of its use. The resulting specifications can be directly executed, and we discuss an implementation which makes this direct execution possible.

*This is a revised and expanded version of a paper which appeared at the 6th International Workshop on Agent Theories, Architectures and Languages [25].

1 Introduction

Agent-based computing is fast emerging as a new paradigm for engineering complex, distributed systems [16, 34]. 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 [20, 29, 32]). 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, 33], DESIRE [5, 30] and Z [7]). However, these techniques typically fall short in at least one of the following ways: (i) they prescribe a particular means of performing the separation of concerns and limit the type of inter-relationships that can be expressed between the resulting components; (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 [24] the use of *multi-context systems* [13] 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 and component interaction 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 [28]) or they can be used as the basis for specialisation of new

contexts (inheritance in the object-oriented sense [4]).

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 [24] where we use multi-context systems to simplify the construction of a belief/desire/intention (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 mental attitudes and those of other agents. The first is that multi-context systems make it possible [13] 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) while working within the computationally simpler framework of standard predicate logic. Thus the use of multi-context systems makes it easy to directly execute agent specifications where those specifications deal with modal notions. Again this is illustrated in [24]. 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 [3, 6].

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 that currently exists between the theory and the practice of agent-based systems. While the work described here falls some way short of bridging the gap, it does show the way in which such a bridge might be built. To some extent the advantages of multi-context systems were explored in [24]. However, this paper extends the former by further refining the approach, extending the representation and providing additional support for building complex agents. In particular we introduce three new ideas. The first is that of grouping contexts together into modules, giving another level of abstraction in defining agent architectures. The second is the idea of bridge rules which delete formulae from certain contexts (as opposed to just introducing them), an idea which allows the modelling of consumable resources. The third idea is that of introducing a time-delay into the execution of a bridge rule in order to allow inter-context synchronisation. In addition to these three things we also give some details of the implementation of a system for executing multi-context agents.

The remainder of this paper is structured in the following manner. Section 2 introduces the ideas of multi-context systems on which our approach is founded.

Section 3 explains how we have extended the use of multi-context systems to better handle systems of high complexity. Section 4 then illustrates our approach using a specific agent architecture and a specific exemplar scenario, and Section 5 extends this example to include inter-agent communication. Section 6 describes our prototype implementation, and Section 7 compares our approach to other proposals in more or less the same vein. Finally Section 8 draws some conclusions and discusses the future direction of this work.

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. Whether such agents are the best solution depends on the task the agent is to perform. See [35] for a discussion of the relative merits of logic-based and non logic-based approaches to specifying and building agent architectures.

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 [13], before being identified as a methodology for constructing agent architectures [21] where full detail of the components can be found. In brief, the components are the following:

- *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 [24]), 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 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 [21, 24], 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 (for more detail on how this is achieved, see Section 6). 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 units 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 [21] and used in [24]. 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:

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

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).

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 (such as those surveyed in [31]) as a means of giving a semantics to time-outs. As Gabbay [11] discusses, resource logics like linear logic are captured naturally in systems of argumentation[1], and it is also natural to consider extending the predicates we use to have explicit temporal arguments.

It should be noted that the use of consuming conditions is related to the problem of contraction in belief revision. In both, the removal of formulae from a logical theory means that deductions based upon those formulae become invalid and must be retracted. Since systems of argumentation explicitly record the formulae used in every deduction, it is conceptually simple (if computationally complex in general) to identify those deductions invalidated by the consumption of given formulae[2]. When, as is the case in the examples considered here, the theories from which formulae are retracted are small and involve few deductions, establishing the effects of consumption need not be too difficult.

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 [24]). 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*. Essentially we group related units into modules and separate interconnections into those inside modules and those between modules. This abstraction is, of course, one of the main conceptual advantages of object orientation [4].

[1] To be more precise Gabbay discusses how labelled deductive systems can be used to capture linear logic, but the necessary features of labelled deductive systems are shared with systems of argumentation

[2] A naive procedure for doing this in the general case would be to check that every formula in every argument is still present in the theory, labelling those arguments which rely on formulae now missing from the theory as invalid. For n arguments each of which includes m formulae in its grounds this would involve checking at most mn formulae (assuming no duplication). For a theory which contains N formulae, this would, in the worst case (where each of the m formulae in the grounds of the argument included only formulae from the theory rather than deductions from them), involve checking that each of the mn formulae were present in the N. The worst case complexity of this search would be Nnm.

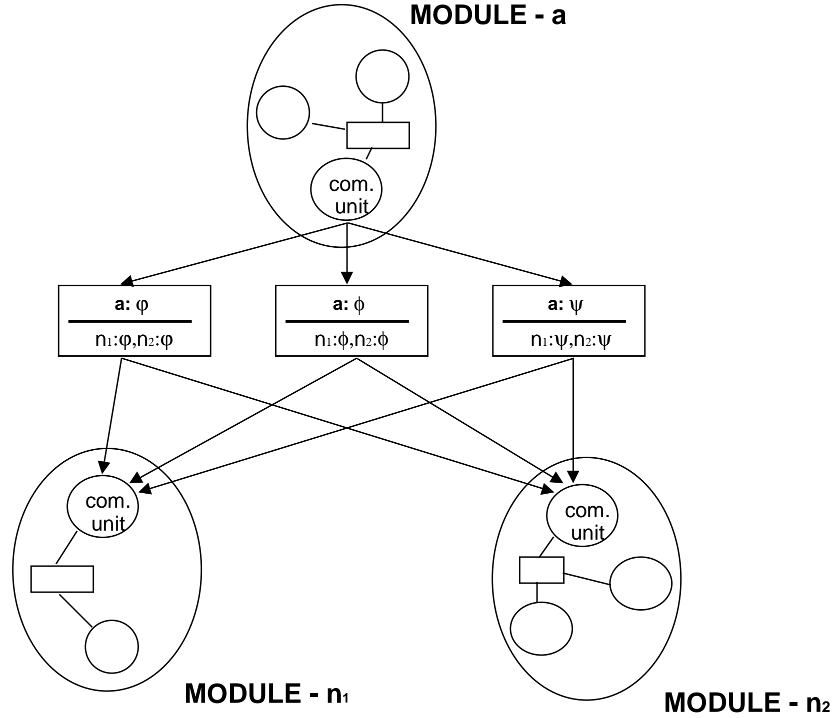

Figure 1: Module inter-connection (from a's perspective only). The bridge rules are enclosed in rectangles.

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. Note that currently we do not allow modules to be nested inside one another, largely because we have not yet found it necessary to do so. However, in the same way that in object-oriented approaches it is useful to allow objects to be nested inside other objects, 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

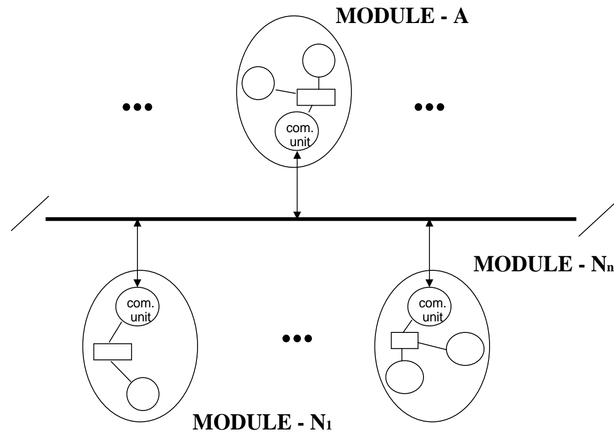

Figure 2: A pictorial explanation of the bus metaphor

in module a and a single conclusion in the modules n_1 and n_2. The use of broadcast communication within the agent was chosen for convenience and simplicity—it is clearly not an essential part of the approach. It does, however, enhance the plug and play approach we are aiming for since when broadcast is used it is not necessary to alter the message handling within an agent when modules are added or removed.

Note that under this scheme *all* modules receive *all* messages, even those messages that are not specially for them. This obviates the need for a central control mechanism which routes messages or chooses which modules should respond to requests from other modules. With this type of connection, adding or removing a module doesn't affect the others (in a structural sense). We can see this communication net as a bus connecting all modules and firing a MBR is the same as putting a message onto this bus. There are as many kinds of messages running along this bus as there are MBRs (see Figure 2).

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, and here we consider one of them which we have found useful as an example. We associate a weight, which we call a "degree of importance", with each message. This value is drawn from the interval $[0, 1]$, where maximum importance is 1 and minimum is 0, and assigned to the message by the communication unit of the module that sends it out. These degrees of importance can be used to resolve contradictory messages, for instance by preferring the message with highest degree of importance. The degrees of importance are discussed further in the next section.

Obviously, the use of modules does not solve every problem associated with altering the structure of an agent. For instance, if the only module which can perform a given task is removed, the agent will no longer be able to perform this task. Similarly, if one module depends on another module to do something and the second is removed,

the first module becomes useless. However, the use of modules does simplify dealing with these kinds of interdependencies by reducing the number of components whose interdependencies have to be considered.

3.2 Messages between modules

We start with a set AN of agent names and a set MN of module names. Our convention is that agent names are upper case letters, and module names are lower case letters. An inter-module message has the form:

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

where

- I is an illocutionary particle that specifies the kind of message. In this paper we use the illocutions Ask and $Answer$.
- S and R both have the form $A[/m]^*$. As elsewhere we use BNF syntax, so that $A[/m]^*$ means A followed by one or more occurrences of $/m$. $A \in AN$ 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. Thus a message with $S = \mathit{Self}/a$ and $R = \mathit{Self}/\mathit{all}$ indicates a message from module a of the agent to all other modules of the agent.
- 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 $\varphi_n = \varphi$.
- $\omega \in [0, 1]$ is the degree of importance associated with the message.

Note that G is exactly the set of grounds of the argument for φ [24]. 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 [11] 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.

A typical intra-agent message for an agent B would thus be:

$$\mathit{ask}(\mathit{Self}/a, \mathit{Self}/\mathit{all}, \mathit{Give}(B, A, \mathit{Nail}), G_1, 0.5)$$

meaning that module a of an agent B is asking all the other modules in B whether B should give an agent called "A" a nail. The reason for doing this is G_1 and the weight a puts on this request is 0.5. Currently we treat the weights of the messages as normalised possibility measures [8], interpreting them as the degree to which the module sending believes that other modules should take the content of the message to be important. Because of the possibilistic semantics we combine the disjunctive support for $\mathit{not}(\mathit{Give}(B, A, \mathit{Nail}))$ using max as is the case for possibility measures [8].

The advantage of using possibility theory to underpin the degrees of importance, as opposed to developing some new measure from scratch, is that it allows us to exploit the large body of work on possibility theory to solve problems arising from

the use of intra-agent messages. Thus, because in possibility theory it is perfectly acceptable for a proposition and its negation to have a degree of possibility of 1 (it just indicates that the opinion about the proposition is equally balanced) we can sidestep some of the problems which would occur if we were, for example, using probability theory to underpin the degrees (a probability of 1 for a proposition and its negation would represent a contradiction). It is also acceptable for a message to have a degree of possibility of 0; this means that its negation is believed to be of maximum importance.

There are three points which need to be made about these messages. Firstly, the degrees of importance in the message, based as they are in possibility theory, have a common meaning across all modules. The fact that different modules assign different degrees is not because they mean different things, but because the various modules have different views of the world. Thus the degrees of possibility they assign represent different preferences, as for example discussed in [1]. Secondly, although the different modules can use different languages, the content of the messages passed must be a common set of terms, a communication language of sorts, which are given the same meaning by all modules. In the interests of generality, we don't specify such a language, leaving that to the designer of the agents, but in later sections we give a set of predicates which we have used for this task. Finally, because the modules can use different logics, it maybe that one module cannot understand the record of the derivation provided by another. In our work we have side-stepped this issue for now by using one common logic, as in the examples we give later. However, we are working on argumentation frameworks in which the inference rules are explicitly denoted (so that the derivations may be traced) and themselves form the basis of discussion between modules (so that those acceptable derivations may be identified) [19]. Indeed, the roots of this work are present in [24].

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 [22] which could be based upon the weight of the relevant intra-agent messages.)

Thus, a message from B to A offering the Nail mentioned above would have the form:

$$inform(A, B, Give(B, A, Nail), G_2)$$

As in the case of intra-agent messages, the content of inter-agent messages must be written in some communication language which has a common meaning across all agents to whom the communication is sent. Once again we do not prescribe such a language, though we do give an example of one we have used later in the paper.

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

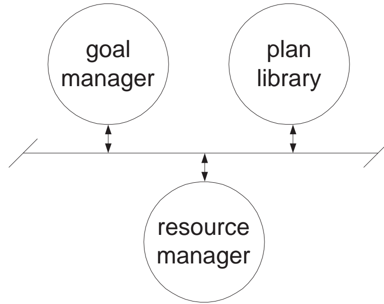

Figure 3: The modules in the agent

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 [23], which is supposed to roam the authors' homes making small changes 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 [24], 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). Subsequent sections build upon this basic definition to produce more sophisticated agents.

4.1 A high-level description

The basic structure of the agent is that of Figure 3. 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 illocution 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.

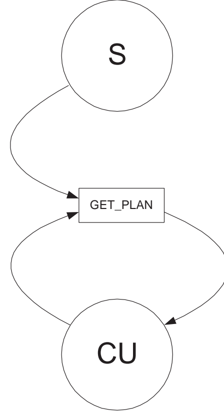

Figure 4: The plan library module

- $have(X, Z)$: $X \in AN'$ 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.

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 plan to achieve the goal of possessing 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 4. 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.

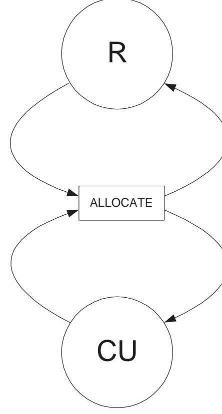

Figure 5: The resource manager module

The bridge rule connecting these units is:

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

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[3].

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 5. 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:

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

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.

[3]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.

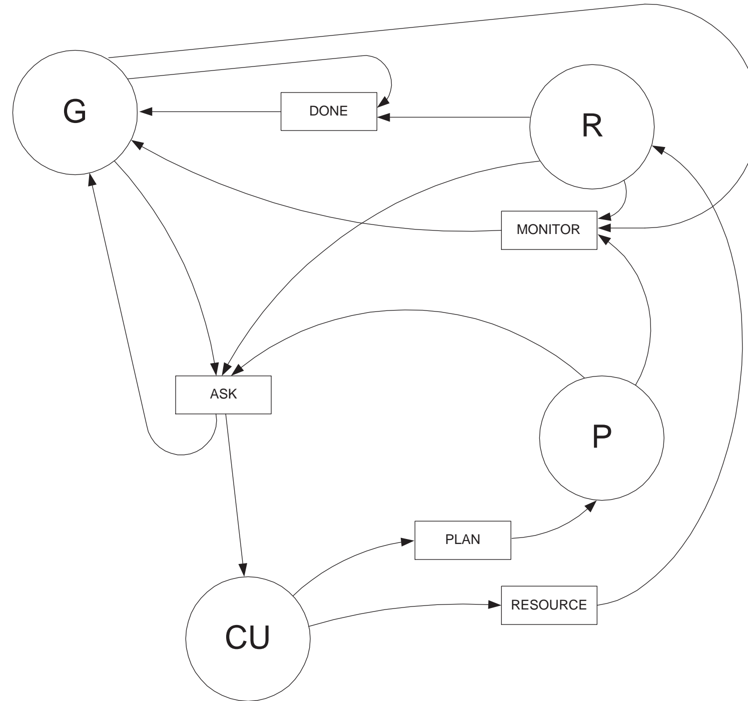

Figure 6: The goal manager module

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 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 6 which shows the units it contains, and the bridge rules which connect them. These units 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:

$$\begin{array}{rcl} monitor(X,P) & \rightarrow & assert_subgoals(P) \\ monitor(X,P) & \rightarrow & prove(P) \\ monitor(X,P) \wedge proved(P) & \rightarrow & done(X) \\ \\ assert_subgoals(\bigwedge_i Y_i) & \rightarrow & \bigwedge_i goal(Y_i) \\ prove(X \wedge \bigwedge_i Y_i) \wedge done(X) & \rightarrow & prove(\bigwedge_i Y_i) \\ \bigwedge_i done(Y_i) & \rightarrow & proved(\bigwedge_i Y_i) \end{array}$$

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[4]. 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:

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

Of course, this is a very rudimentary plan, which only consists of the basic resources needed to achieve the goal of hanging a picture. The resource repository holds the information that the agent has a picture, nail and a hammer:

$$\begin{array}{rcl} R & : & resource(picture, free) \\ R & : & resource(nail, free) \\ R & : & resource(hammer, free) \end{array}$$

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.

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

Table 1: The inter-module messages

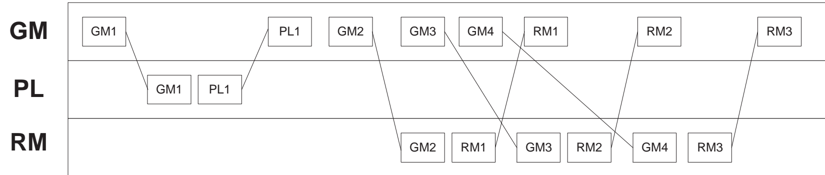

Figure 7: An execution trace for the agent

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 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 7—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.

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

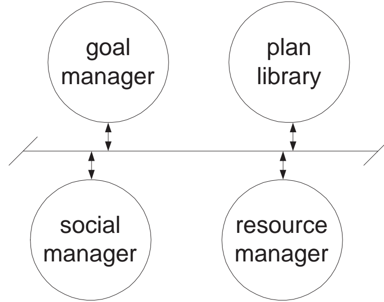

Figure 8: The modules in the agent

5 Specifying more complex agents

This section gives a specification of a pair of agents which build upon those in the previous section. Indeed the agents introduced here are strict extensions of those in the previous section, containing all the components (down to the level of individual units) of the autistic agents and other components besides. The main extension is to reduce the autism of the model by giving each agent mechanisms for interacting with other agents. The resulting agents are thus intermediate in complexity between that described in the previous section and that described in [24]. In comparison with the latter, the main simplification is the absence of mechanisms for argumentation.

5.1 A high-level description

The basic structure of the agent is that of Figure 8. There are four modules connected by multicast bridge rules. These are the plan library (PL), the resource manager (RM), the goal manager (GM) and the social manager (SM). The first three modules carry out the same basic functions as their namesakes in Section 4. The social manager handles interactions with other agents.

The intra-agent messages are exactly the same as for the autistic agent, but there are also two types of inter-agent message, which broadly correspond to the *ask* and *answer* messages. These are:

- **Request:** a request to another agent.
- **Reply:** an answer to an inter-agent request.

As in the previous section these illocutions are the only actions available to the agents. Thus the agents can talk about passing resources between themselves, but we provide no mechanisms for actually passing the resources.

The only new predicate which these agents employ is:

- $give(X, Y, Z)$: $X \in AN'$ and $Y \in AN'$ are agent names, and Z is a string describing a resource. This denotes X giving Z to Y.

and this is used in conjunction with the *request* and *reply* message types to build inter-agent messages, which are of the form:

$$request(A, B, give(B, A, nail), \{\})$$

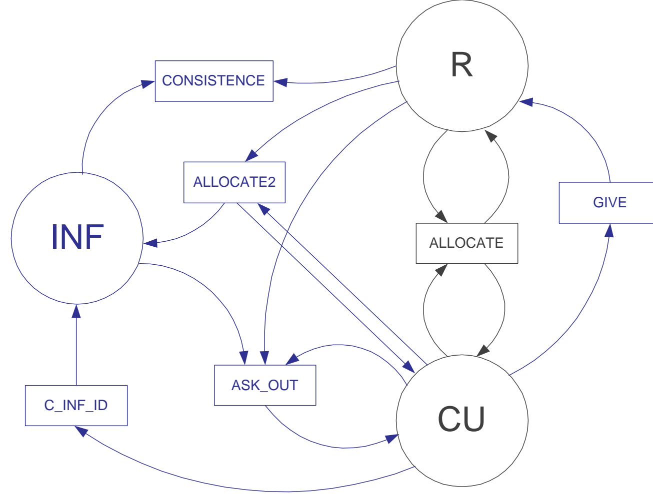

Figure 9: The resource manager module

In this example, A requests that B gives a nail to A. In the following:

$$reply(B, A, give(B, A, nail), \{\})$$

B replies to A that B will give the nail to A.

5.2 Specifications of the modules

Once again, having decided on the overall structure of the agent, we have to specify the internal structure of the individual modules. The plan library module and the goal manager module have exactly the same structure as in the simple agent (see Section 4.2) and are not repeated here.

As can be seen by comparing Figure 5 with Figure 9, the resource manager of our new agents is considerably more complex than that in Section 4. This resource manager contains an extra unit INF which holds information about the resources possessed by agents in contrast with the R unit which simply records whether resources are free or allocated—this is a complication introduced by moving from one agent to several. Because the autistic agent does not deal with any external entities, any resources it considers belong to it, and any resources which do not belong to it do not exist as far as it is concerned. The social agents, in contrast, need to consider two aspects to every resource—whether or not it is free, and who has control over it. The R unit deals with the former, and the INF unit with the latter.

Clearly with more units we have more bridge rules. Of those in Figure 9, only the ALLOCATE rule is familiar from the autistic agent:

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

where $resource(Z, allocated)$ 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. This rule will be used if the agent is dealing with its own need for a resource that it owns, as in the case of the autistic agent.

Because we now have two agents, the resource that one agent requires may be owned by another agent, and this situation is where the INF unit comes into play. There are four bridge rules which relate this unit to R and CU. The first of these is C_INF_ID, which places knowledge about which agent has which resource into INF as a result of an *inform* message:

$$\text{C_INF_ID} \quad = \quad \frac{CU > inform(U, V, have(X, Z), \{W\})}{INF : have(X, Z)}$$

The name indicates that the rule is a kind of identity rule between the CU and INF units. Because in this model resources belong to just one agent, there is a contradiction if a resource is thought to belong two agents at once. The CONSISTENCE rule ensures that this situation does not occur by ensuring that the agent doesn't think another agent has a resource $have(X, Z)$ when in fact the agent has the resource itself $resource(Z, free)$.

$$\text{CONSISTENCE} \quad = \quad \frac{\begin{array}{c} INF > have(X, Z), \\ R : resource(Z, free) \end{array}}{}$$

The bridge rule works by detecting that the resource Z is recorded both as being free and being owned by agent X, and simultaneously deleting the record of the fact that Z is owned by X using a deleting condition. The rule will be fired when, for example, Agent A knows that B has some resource, but is then presented with the information that the resource is now free because B has given it up. Without the bridge rule, A would continue to believe that B has the resource. Because the purpose of this bridge rule is to delete the $have(X, Z)$, and this is achieved by its antecedents, there is no consequent, making the rule unlike others in the agent (and subsequently stressing the operational nature of our use of bridge rules). If one wanted to not only remove the $have(X, Z)$, but also conclude that $not(have(X, Z))$ was subsequently the case, the appropriate rule would be:

$$\text{CONSISTENCE2} \quad = \quad \frac{\begin{array}{c} INF > have(X, Z), \\ R : resource(Z, free) \end{array}}{INF : not(have(X, Z))}$$

If an agent requires a resource it does not have, the ASK_OUT bridge rule allows it to request the resource from another agent, and the GIVE rule makes it possible to accept a resource it is given:

$$\text{ASK_OUT} \quad = \quad \frac{\begin{array}{c} CU : ask(Self/Sender, Self/Receiver, goal(have(X, Z), \{P\}), \\ R : not(resource(Z, free)), \; INF : have(Y, Z) \end{array}}{CU : ask(Self/RM, Self/SM, give(Y, X, Z, \{P\}))}$$

$$\text{GIVE} \quad = \quad \frac{\begin{array}{c} CU > ask(Self/RM, Self/SM, give(X, Y, Z), \{P\}), \\ CU > answer(Self/SM, Self/RM, give(X, Y, Z), \{Q\}) \end{array}}{R : resource(Z, free)}$$

The final resource-related situation an agent may be in is when it has a resource that another agent requires. This situation is handled by the ALLOCATE2 bridge rule,

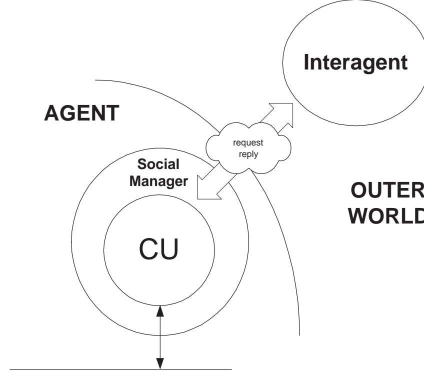

Figure 10: The social manager module

which hands over a resource if it is free and the social manager tells it to, updating the INF unit with information about where the resource is:

$$\mathsf{ALLOCATE2} \quad = \quad \frac{\begin{array}{c} CU > ask(Self/SM, Self/Receiver, give(Self, Y, Z), \{P\}), \\ R > resource(Z, free) \end{array}}{\begin{array}{c} CU : answer(Self/RM, Self/SM, give(Self, Y, Z), \{P\}), \\ INF : have(Y, Z) \end{array}}$$

This completes the description of the resource manager.

The final module in the new agents is the social manager. As can be seen from Figure 10, here the social manager consists of a single communication unit CU. As well as being connected to the agent's internal modules via multi-cast bridge rules, the social manager module is also connected to the corresponding module of the agent's aquaintances via an "interagent" which handles the transfer of messages between agents (see Section 6). This passes *request* and *reply* on to the communication unit of the module to which they are addressed. The generation of these messages is carried out by the theory in the communication unit.

5.3 Specifications of the units

So far we have described the modules which make up the new agents, and for each module we have identified both the units which composed them and the connections necessary between the units. The next step is to decide what the internal structure of the units will be—which formulae and theories they will contain, and which logics those theories will be written in. Once again, there are not many units which include more than just a few atomic formulae. One of these is the unit G of the goal manager which contains the same theory as in the autistic agent (see Section 4.3):

The other unit which contains more than just atomic formulae is the CU unit in the social manager, which contains:

$$ask(Self/Sender, Self/SM, give(X, Self, Z), \{\}) \rightarrow$$
$$request(Self, X, give(X, Self, Z), \{\})$$
$$reply(X, Self, give(X, self, Z), \{\})$$

$$\wedge\ ask(Self/Sender, Self/SM, give(X, Self, Z), \{\}) \rightarrow$$
$$answer(Self/SM, Self/Sender, give(X, Self, Z), \{\})$$
$$request(X, Self, give(Self, X, Z), \{\}) \rightarrow$$
$$ask(Self/SM, Self/RM, give(Self, X, A), \{\})$$
$$answer(Self/Sender, Self/SM, give(Self, X, Z), \{\}) \rightarrow$$
$$reply(Self, X, give(Self, X, Z), \{\})$$

This theory takes care of the translation from intra-agent messages to inter-agent messages. The first formula takes an incoming *ask* message which contains a request for another agent, X to give a resource, and converts it into a *request* illocution. The second formula handles the reply to that request—if another agent responds positively to a request that the agent has previously made, then an *answer* message is generated and sent to the orginator of the request. The next two formulae handle responses to requests. The first of these takes a request for a resource from another agent and turns it into a message to the resource manager. The second takes a positive response, and converts that into a *reply* message. The logic used in this unit, as in all the units in this agent specification, is classical first order logic.

As in Section 4, the specification up to this point is generic, defining something like a class description for simple non-autistic agents. For the particular scenario we have in mind, that of two agents which co-operate in hanging a picture, it is necessary to instantiate this generic description twice. The first instantiation creates an agent A which is virtually the same as the autistic agent of Section 4, the only difference being that A does not have the nail necessary to hang the picture, knowing instead that an agent B has the nail. A's plan repository holds the same plan as that of the autistic agent:

$$S \ : \ plan(hangPicture(X), have(X, picture) \wedge have(X, nail) \wedge have(X, hammer))$$

A's resource repository holds the information that the agent has a picture and a hammer:

$$R \ : \ Resource(picture, free)$$
$$R \ : \ Resource(hammer, free)$$

while A's INF unit holds the information that B has a nail:

$$INF \ : \ have(B, nail)$$

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

$$G \ : \ goal(hangPicture(A))$$

This completes the specification of A. B is much simpler to instantiate, since it is only necessary to program it with the resource of a nail, by adding the following formula to its resource repository:

$$R \ : \ Resource(nail, free)$$

This completes the specification of the two agents.

Agent A

$ask(Self/GM, Self/all, goal(hangPicture(A)), \{\})$	(GM1)
$answer(Self/PL, Self/GM, goal(hangPicture(A)),$ $\{have(A, picture) \wedge have(A, nail) \wedge have(A, hammer)\})$	(PL1)
$ask(Self/GM, Self/all, goal(have(A, picture)), \{\})$	(GM2)
$ask(Self/GM, Self/all, goal(have(A, nail)), \{\})$	(GM3)
$answer(Self/RM, Self/SM, give(B, A, nail), \{\})$	(RM1)
$ask(Self/GM, Self/all, goal(have(A, hammer)), \{\})$	(GM4)
$request(A, B, give(B, A, nail), \{\})$	(SM1)
$answer(Self/RM, Self/GM, have(A, picture), \{\})$	(RM2)
$answer(Self/RM, Self/GM, have(A, hammer), \{\})$	(RM3)
$answer(Self/SM, Self/RM, give(B, A, nail), \{\})$	(SM2)
$answer(Self/RM, Self/GM, have(A, nail), \{\})$	(RM4)

Agent B

$ask(Self/SM, Self/RM, give(B, A, nail), \{\})$	(SM1)
$answer(Self/RM, Self/SM, give(B, A, nail), \{\})$	(RM1)
$reply(B, A, give(B, A, nail), \{\})$	(SM2)

Table 2: The inter-module messages

5.4 The agents in action

If we execute these two agents, they generate and exchange the messages in Table 2 and Figure 11, which are very similar to those generated by the autistic agent. The main difference in this case concerns the provision of the nail required to hang the picture. In the case of the autistic agent, this nail was the property of the agent and so all the agent had to do to execute its "plan" of owning the nail was to allocate it. In this case when Agent A wants the nail it has to request it from B. Luckily for A, when B receives this request, it immediately agrees.

In more detail, the execution proceeds as follows. The goal manager unit of Agent A, 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 GM1 (detailed in Table 1). When this message reaches A's 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). When GM2 and GM4 reach the resource manager, they cause the same sequence of events as in the autistic agent, firing the ALLOCATE rule, generating the messages RM2 and RM3 and allowing the goal manager to build part of its plan.

The problem, of course, is with GM3 which is requesting a nail. Since this is not a resource that A owns, the ASK_OUT rule is fired, generating RM1 which in turn

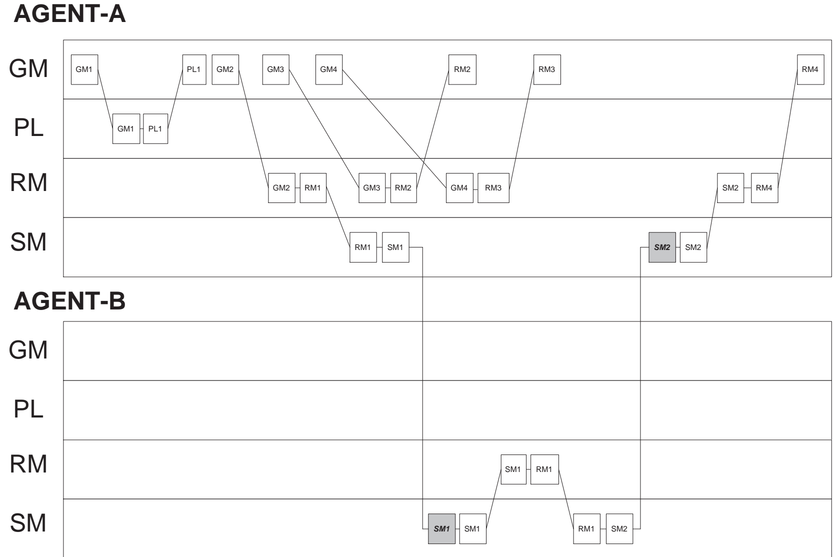

Figure 11: An execution trace for the agents

sparks off activity in the social manager resulting in SM1. This *request* is passed to B, where the social manager generates $SM1$. This goes to B's resource manager, triggering the ALLOCATE2 rule and then RM1 which confirms that B is happy to give a nail to A. The message passes back through B's social manager as SM2, is received by A's social manager becoming A's SM2 message. This activates the GIVE rule in A's resource manager, which updates its resource list finally allowing it to alocate the nail. The message RM4 is sent to the goal manager which now has all the resources it requires. Consequently 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.

Both of the examples we have given are of rather simple agents. However, we can use the framework to specify and execute more complex agents (indeed we have already done so), so it should not be inferred that the examples given illustrate the limits of what is possible using this approach. The simplicity of these examples is purely for ease of presentation and understanding.

6 Implementation

Our aim in this work is to be able to directly execute the kinds of specifications developed in previous sections. To do this we clearly need some kind of computational infrastructure which will make it possible to define units, modules, and their interconnections, and then execute the various theories within these components. This section describes our prototype implementation of this infrastructure.

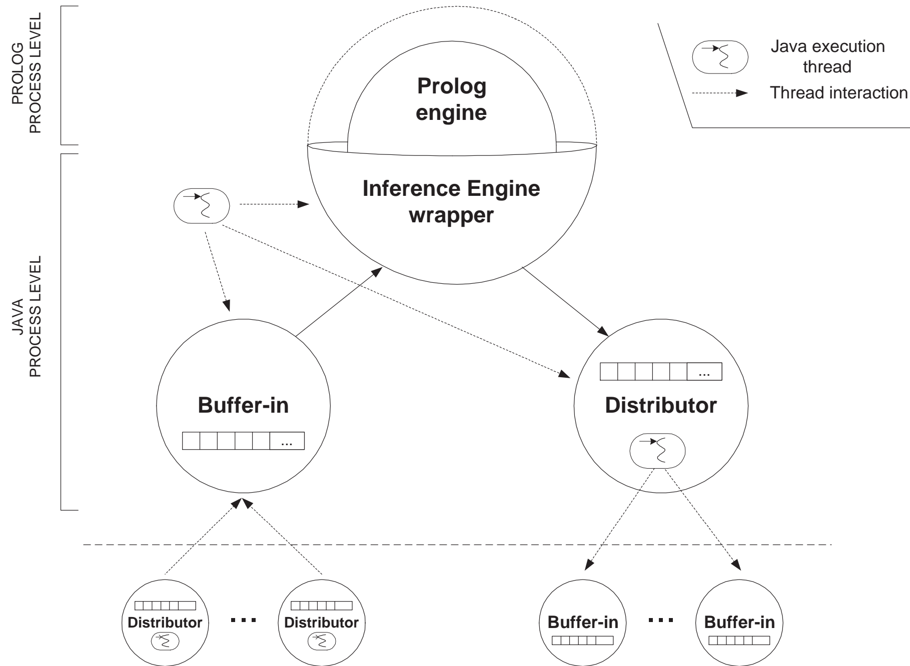

Figure 12: How a component is implemented

6.1 Units and bridge rules

To support the execution of the kinds of specifications developed above, we need to be able to handle two kinds of component—units, and bridge rules (multi-cast bridge rules between modules are just bridge rules which connect every social managers to every other social manager). We treat both these components in the same way, giving them the structure pictured in Figure 12. Thus every component has, at its core, a theorem prover written as a meta-interpreter in Prolog. Thus every unit is just a Prolog-based meta-interpreter, as is every bridge rule. Now, because we want these units and bridge rules to run concurrently, we wrap the Prolog engine which executes the meta-interpreter in a Java wrapper and this wrapper is set running in its own thread (from outside the wrapper it looks just like any other Java object).

These components need to be connected, and we want these connections to be asynchronous. The idea is that each component runs its meta-interpreter on the current contents of its theory, computes the closure of this, outputs the relevant results, looks at what new input has arrived, updates its theory, and then starts computing the closure of the theory once again. To get this asynchronicity we provide each component with an input buffer by using a Java object called *buffer-in*. This stores incoming information until the meta-interpreter finishes running. To ensure that the right information is sent to the right other components we provide each component with a Java *distributor* which provides this routing capability.

The operating cycle of this structure can be split in four stages:

Update: All actions (`asserts` and `retracts`) that have been accumulated during

the inference stage in the buffer-in object are applied to the working memory of the relevant Prolog process through the wrapper class.

Inference: A new inference cycle is launched in the Prolog engine.

Distribution: All formulae deduced during the inference stage are sent to the distributor object as soon as they are deduced.

Actualization: The changes notified during the inference process, and accumulated in the distributor, are transmitted to the buffer-in objects of the relevant components.

The update, inference and distribution stages are carried out sequentially in a single execution thread while the actualization stage has its own execution thread to allow the changes performed during the inference stage (`asserts` and `retracts` in the Prolog engine) to be transmitted to the other components without delaying the execution of the other stages.

6.2 From components to modules

Taking the general structure of a component, we obtain units and bridge rules by specialization[5]. Usually, each structure implements a single component, that is, one unit, or one bridge rule. There is only one exception to this: if bridge rules Br_i and Br_j have a consumption condition from the same unit, then they must be placed in the same component. The reason for doing this is to avoid a situation in which several bridge rules with a common consumption condition fire at the same time, and to understand why this might occur is it is necessary to understand in more detail how the bridge rules are implemented.

In order to reduce the messages between units and bridge rules, each bridge rule has a partial image of the units related with it, in the sense that it only has access to those literals which appear in its antecendents (since these may easily be indentified from the bridge rule itself, this is easy to achieve). This image is updated when there is a change in the unit, and the update is via the messages from the unit's distributor to the bridge rule's buffer-in. Each distributor "knows" which formulae of the theory could match the premises of the bridge rules to which it is connected, and only those formulae will be sent. This implementation is efficient. However, consumption bridge rules can cause unwanted inconsistencies between different images of the same unit.

It's easy to see the problem with an example. Consider the two bridge rules in Figure 13 (BR1 and BR3) implemented using two different components and suppose z is in Unit2. From this Unit1 deduces x. Bridge rules BR3 and BR1 are notified about this change and BR3 is fired. Meanwhile, Unit1 deduces r and BR1 is alerted. Unit1 receives a message from BR3 telling it to remove x, but by this time BR1 has already been sent r and has fired. The solution to this problem is to make those bridge rules with coincident consumption conditions (in our example BR1 and BR3) share a common vision of the units related with them (in our example Unit1 and Unit2). In our implementation this is done including those bridge rules into the same

[5]As a result, our approach has an even more object-oriented flavour than might be first thought. Not only is everything made up of units and bridge rules, but the units and bridge rules themselves are basically variations on the same thing.

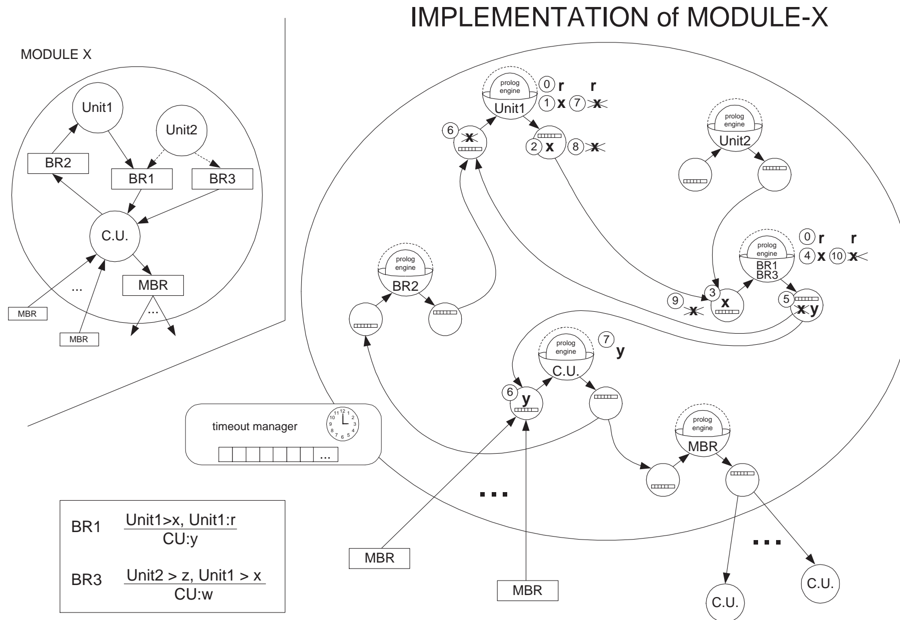

Figure 13: How a module is implemented

structure and passing the update message to that structure just once. This ensures that only one bridge rule with a consuming condition will respond to a given update, and has proved sufficient for our purposes so far. However, it does not provide a perfect solution in that it does not seek to mediate between two bridge rules which compete to consume the same literal[6].

The last thing that needs describing is the *timeout manager*. This is the entity responsible for monitoring timeout bridge rules. When the preconditions of a bridge rule with a timeout hold, the bridge rule tells the timeout manager. The timeout manager has a list with pairs:

$$(\langle timer \rangle, \langle bridge\ rule\ reference \rangle)$$

The timers in the list are initialized to the values in the timeout bridge rules and decrease synchronously. When a timer becomes 0, the timeout manager informs the corresponding bridge rule structure that it is authorised to fire if the conditions still hold.

These components can then be slotted together to make a module as shown in Figure 13. This shows a module made up of three units—a communication unit and two units named Unit1 and Unit2, three bridge rules—named BR1, BR2 and BR3,

[6] As the implementation stands it codes the bridge rules into this common structure in a given order and then passes the message to the rules in that order, thus ensuring that the first rule with a consuming condition always has precedence. One might argue that the message should be passed to a randomly selected bridge rule, but equally one might argue that it is wrong to build agents in such a way that several units are competing for the same literal.

and a multicast bridge rule named MBR. The timeout manager is also depicted. As an example of how the module works, consider the following:

0. Unit1 contains **r**. BR1-BR3 have been notified.
1. The Prolog engine in Unit1 deduces **x**.
2. The distributor in Unit1 is notified about the deduction of **x**.
3. The distributor in Unit1 sends **x** to the BR1-BR3 buffer-in.
4. The buffer-in of BR1-BR3 sends **x** to the Prolog engine
5. As a result of firing BR1, the distributor is notified that **y** must be added to the Communication Unit and that **x** must be removed from Unit1.
6. The buffer-in of Unit1 receives a request to remove **x** and the buffer-in of the Communication Unit is notified about **y**
7. **x** is removed from the Prolog engine of Unit1 and **y** is added to the Prolog engine of the Communication Unit

8, 9 and 10. The Prolog engine of BR1-BR3 is notified about the elimination of **x**.

6.3 From modules to agents

Little more needs to be added to this picture to get a complete agent. The main addition to a collection of modules, each as described above, is a set of multicast bridge rules which connect modules together. These are implemented in exactly the same way as the intra-module bridge rules. As discussed earlier in the paper, each agent includes a social manager module which, while structured exactly like every other module, handles the interactions with other agents—the detail of this, which clearly changes from agent to agent, is encoded in the units that make up the social manager and the bridge rules which connect them. Building social managers with different units or different theories in the same units makes it possible to implement different social conventions since it is these units and theories which define the agent communication protocol. The protocol we use here is based on that described in [21, 27], and the inter-agent messages are transferred by autonomous software entities known as *interagents*. The interagents we use in our current implementation are similar to the system JIM described in [17, 18], and are built on top of JADE [2], an implementation of the mandatory elements contained within the FIPA [10] specification for agent interoperability.

The implementation described here is the beginning of a bridge between specification and execution. The modular multi-context approach gives us a means of specifying agents. The implementation, as it stands, gives us a means of executing these specifications, albeit once the theories in each unit have been translated into a form which can be executed on the appropriate Prolog engine, and it is this translation which is the missing section of the bridge. Once each Prolog engine is equipped with a suitable meta-interpreter for the logic employed by that unit, it will be possible to take the specification, enter the theory in each unit into the meta-interpreter, and directly execute it.

7 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 [14], Agent0 [26] and GRATE* [15], 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 promising to allow the direct execution of the specification as discussed above. This relation to direct execution also distinguishes our work from that on modelling agents in Z [7], since it is not yet possible to directly execute a Z specification[7].

More directly related to our work is that on DESIRE and Concurrent MetateM. DESIRE [5, 30] 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. We could, of course, simulate the kind of control found in DESIRE by building a central controlling module, if this were required.

Concurrent MetateM defines concurrent semantics at the level of single rules [9, 33]. 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 [12], 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 [6]. In our case, however, any nested beliefs

[7]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 of the kind possible using our approach.

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.

8 Conclusions and future work

This paper has proposed a general approach to defining agent architectures which extends the work of [24, 25] with the idea of modules and, as a result, links the approach more strongly with the software engineering tradition. This approach provides a means of structuring logical specifications of agents in a way which makes them directly executable, and we have described an implementation which can carry out this execution. The approach has a number of advantages over other work on defining agent architectures. 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 [4]. 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. The development of such a methodology is our long term goal in this work, but the outline of such a methodology is, we believe, already visible in the resuse of the simple autistic agents of Section 4 as the basis of the more complex agents of Section 5

However, before we can develop this methodology, there are some open issues to resolve. The first thing that we need to do is to extend both the environment for building agents and the range of agents we can build. Our aim with the environment is to turn it into a graphical tool for the rapid prototyping of declarative agents, with a library of different units and bridge rules which can be connected together and edited as required. To be useful this, of course, requires us to first develop specifications for agents which have greater capabilities than the ones described in this paper—agents which are capable of a wider range of dialogues, and which have expertise in areas other than home improvement. The second thing we need to do is to ensure that the module-based approach we have described scales up for more complex agents, something that can only be ensured by building such agents. Experience gained here might well lead to some modifications in our approach, for instance allowing nested hierachies of modules. The last thing we need to do is to resolve the question of the semantics of the comsuming conditions and time-outs in bridge rules. Though last in the list, this is possibly the most important in terms of ensuring that we can build precisely specified agents.

Acknowledgments

This work has been supported by the Spanish CICYT project SMASH, TIC96-1038-C04001, the UK EPSRC project Practical Negotiation for Electronic Commerce GR/M07076 and the EC IST project Sustainable Lifesycles for Information Ecosystems IST-1999-10948. 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). We are grateful to the anonymous referees for their observations on the initial version of this paper.

References

[1] L. Amgoud, S. Parsons, and L. Perrussel. An argumentation framework based on contextual preferences. In *Proceedings of the International Conference on Formal and Applied and Practical Reasoning*, pages 59–67, 2000.

[2] F. Bellifemine, A. Poggi, and G. Rimassa. JADE: A FIPA-compliant agent framework. Technical report, CSELT S.p.A, 1999. Part of this report has been also published in the Proceedings of the Conference on Practical Applications of Agents and Multi-Agents, 1999, 97–108.

[3] 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.

[4] G. Booch. *Object-oriented analysis and design with application.* Addison Wesley, Wokingham, UK, 1994.

[5] 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.

[6] 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.

[7] 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.

[8] D. Dubois and H. Prade. *Possibility Theory: An Approach to Computerized Processing of Uncertainty.* Plenum Press, New York, NY, 1988.

[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] Foundation for Intelligent Physical Agents. `http://www.fipa.org`.

[11] D. Gabbay. *Labelled Deductive Systems.* Oxford University Press, Oxford, UK, 1996.

[12] F. Giunchiglia. Contextual reasoning. In *Proceedings of the IJCAI Workshop on Using Knowledge in Context*, 1993.

[13] F. Giunchiglia and L. Serafini. Multilanguage hierarchical logics (or: How we can do without modal logics). *Artificial Intelligence*, 65:29–70, 1994.

[14] 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.

[15] N. R. Jennings. Controlling cooperative problem solving in industrial multi-agent systems using joint intentions. *Artificial Intelligence*, 75:195–240, 1995.

[16] N. R. Jennings. Agent-based computing: Promise and perils. In *Proceedings of the 16th International Joint Conference on Artificial Intelligence*, pages 1429–1436, 1999.

[17] F. J. Martin, E. Plaza, J. A.Rodríguez:Aguilar, and J. Sabater. Java interagents for multi-agent systems. In *Proceedings of the AAAI Workshop on Software Tools for Developing Agents*, Madison, WI, 1999.

[18] F. J. Martin, E. Plaza, J. A.Rodríguez:Aguilar, and J. Sabater. JIM: A Java Intergent for Multi-agent systems. In *Proceedings of the 1st Catalan Conference on Artificial Intelligence (1er Congrés Català d'Intel·ligència Artificial)*, pages 163–171, Tarragona, Spain, 1999.

[19] P. McBurney and S. Parsons. Tenacious tortoises: a formalism for argument over rules of inference. In G. Vreeswijk, editor, *Workshop on Computational Dialectics, Fourteenth European Conference on Artificial Intelligence (ECAI2000)*, pages 77–84, Berlin, Germany, 2000. ECAI.

[20] 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.

[21] 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.

[22] 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.

[23] S. Parsons and N. R. Jennings. Negotiation through argumentation—a preliminary report. In *Proceedings of the 2nd International Conference on Multi Agent Systems*, pages 267–274, 1996.

[24] 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.

[25] J. Sabater, C. Sierra, S. Parsons, and N. R. Jennings. Using multi-context systems to engineer executable agents. In N. R. Jennings and Y Lespérance, editors, *Intelligent Agents VI*, pages 277–294. Springer Verlag, Berlin, 1999.

[26] Y. Shoham. Agent-oriented programming. *Artificial Intelligence*, 60:51–92, 1993.

[27] C. Sierra, N. R. Jennings, P. Noriega, and S. Parsons. A framework for argumentation-based negotiation. In M. P. Singh, A. Rao, and M. J. Wooldridge, editors, *Intelligent Agents IV*, pages 177–192. Springer Verlag, Berlin, 1997.

[28] C. Szyperski. *Component Software*. Addison Wesley, Wokingham, UK, 1998.

[29] 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.

[30] J. Treur. On the use of reflection principles in modelling complex reasoning. *International Journal of Intelligent Systems*, 6:277–294, 1991.

[31] L. Vila. *On temporal representation and reasoning in knowledge-based systems.* PhD thesis, Institut d'Investigació en Intelligencia Artficial, 1995.

[32] 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.

[33] 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.

[34] M. Wooldridge. Agent-based software engineering. *IEE Proceedings on Software Engineering*, 144:26–37, 1997.

[35] M. J. Wooldridge and N. R. Jennings. Intelligent agents: Theory and practice. *The Knowledge Engineering Review*, 10:115–152, 1995.