Multiagent and Grid Systems – An International Journal 5 (2009) 109–131
DOI 10.3233/MGS-2009-0122
IOS Press

Environment support for developing and configuring adaptive agents

Liang Xiao* and Des Greer
School of Computer Science, Queen's University Belfast Belfast, BT7 1NN, UK

Abstract. Adaptivity is very often an important goal for software systems. This paper reviews existing approaches to achieving adaptivity in object oriented systems, particularly those using design patterns, and concludes that there are further opportunities for improving adaptivity in agent systems. The approach described proposes that agents should be coupled with the environment, rules and laws about agent behaviour being externalised in a continuously re-configurable knowledge repository. Tools have been implemented to support the re-configuration. Once new requirements are specified by business experts via the tools, the system automatically adapts its behaviour in the environment, without additional effort from developers. This novel approach pulls together a knowledgebase, configuration tools, and business experts as an integral environment through which the MAS achieves re-configurability ranging from overall infrastructure to individual policy sets. This fosters cost effective software evolution because much of the effort resulting from changes to business strategies and collaborations can be shifted from developers to customers, code change not being required since the environment maintains the dynamics instead.

Keywords: Adaptive agent model, adaptivity, agent, design pattern, environment, requirements engineering, UML, XML

1. Introduction

Business environments are constantly changing. Therefore supporting software systems need to be quickly and easily adapted at frequent intervals. Analysis for new requirements, re-design, re-development, and re-deployment is recurrent but not easy and always costly. The major difficulty in building an adaptive application is that the existing technology and language support usually imposes fixed component functionalities and dependencies. Even in Multi-agent Systems (MAS), Interaction Protocols (IPs) [4], that model agent conversations, have to be manually turned into program code by developers. It is hard to dynamically configure the architecture of MAS or individual agent behaviour rules using current development and runtime environments.

Although precisely defined protocols provide an exact formalism as the basis on which agents are developed without ambiguity, such restrictions impose difficulty in situations where the environment keeps changing and agents have to adapt their behaviour to fit with variations in interaction protocols. Having various goals due to different business needs or even the same goal under varied environmental conditions, agents have to behave dynamically to meet their goal. They may have to seek collaborators opportunistically, use available services immediately, or conform to emerging policies as needed. The changing environment requires agents to act and react with variations. If an adaptive model can be used by agents to adapt to their environment with various protocols formed dynamically, then designing many distinct communication protocols and models alike separately and pre-determinedly can be avoided. In

*Corresponding author. Tel.: +353 1 402 2310; E-mail: liangxiao@rcsi.ie.

ISSN 1574-1702/09/$17.00

this way, the maintenance burden can be relieved. The Adaptive Agent Model (AAM) approach is aimed at developing such an adaptive model. Briefly, our major objective is using agents to adapt system behaviour according to a changing environment, business models being specified by business experts according to the changing business needs.

The problem with the current MAS development practice is largely due to the fact that people view agents as active software units but seldom associate their (dynamic) behaviour with the changing environment. Engineering agents alone, in isolation from their characteristic surroundings, results in few advantages for MAS over traditional OO systems. An "environment" in the context of MAS is often implicitly regarded as an infrastructure that supports direct communication or indirect coordination among agents. Emphasising the importance of the environment and treating it as a first-class abstraction has been advocated in [13], which describes the environment as providing conditions for agents to exist and mediation for agent interaction and resource access. Indeed, it is via the environment that one agent becomes aware of what is happening around it, being then able to change its own belief about the environment and then to react in a possibly different way so impacting on other agents in the environment. These impacted agents will similarly carry out processes and the system as a whole organises itself iteratively. Therefore, agents in MAS must react in accordance with the changing environment, rather like humans react to theirs. Environmental factors must be explicitly modelled so that agents running in that environment can behave appropriately and in a timely fashion, in interaction with other agents and with access to resources. Otherwise, their behaviour will not reflect the current runtime conditions and business needs and they become obsolete.

We believe that modelling the environment along with its facilitating functions is essential to keeping the MAS useful when requirements change. In this way, agents are kept up to date with their knowledge about other agents, global constraints, and available services in the system. More importantly, agents can also be aware of the behaviour and contribution the system currently expects from them, if such a model is available to them. Taking both concerns into consideration, the environment should firstly be modelled in an *inter-agent* dimension to capture the collective agent behaviour towards their common goals in the environment. Secondly, an *intra-agent* dimension is required where the services and constraints to individual agents in the environment are modelled. Thus, we view the environment as not just a medium for agent communication and coordination but also a knowledge source that could inform on: what/why to communicate; who to communicate with and under which coordination; and the internal computation for agents before or after collaboration with other agents. Keeping the environment current, therefore, ensures requirements are always adhered to by agents and changing requirements are catered for. Hence the environment is not just a place where agents are situated with functional facilities being supported towards static goals and realising current requirements knowledge (e.g. a directory service and messaging service provided by frameworks such as JADE) but also a provider to agents of new emerging requirements knowledge.

We propose an Adaptive Agent Model (AAM) that provides a mediator environment for achieving adaptive agent behaviour. In one aspect it supports an adaptive intra-agent function pattern and in another aspect an adaptive inter-agent dependency pattern. The actual knowledge that these patterns capture includes policy rules and interaction rules that govern intra-agent and inter-agent behaviour, respectively. In the environment architecture, event perception and handling, computation and processing, action performing, and message delivering which are the constituent components of the rules are supported. Such rules are supplied and maintained by business people who are best placed to know about (changing) business needs, and form a key part of the AAM environment for agent interpretation and performance. The result of applying the approach is the alleviation of the environmental change problems that we

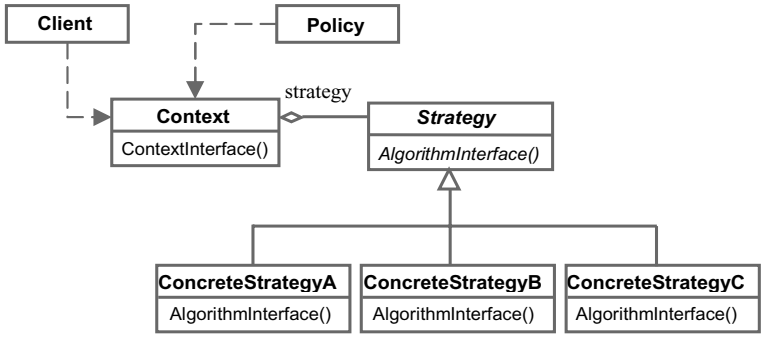

Fig. 1. Strategy Design Pattern [6].

identified at the beginning. Thus, technological changes are minimised, overhead cost should be reduced, and stakeholders can control their own system. The approach is novel but also practical.

The rest of the paper is structured as follows. Section 2 gives an overview of sample design paradigms towards software adaptivity. The investigated design patterns and models use metadata in one form or another attempting to achieve adaptivity but end up with various weaknesses or limitations. Nevertheless, the common characteristics of these inspire the idea that using a dedicated knowledgebase to support agent behaviour in the MAS environment can better manage the dynamics of the environment. In Section 3 we present the use of two types of rules: policy rules and collaboration rules, which form part of our AAM environment and illustrate them using two associated e-commerce case studies. It is demonstrated that adaptivity is achieved in both intra-agent function and inter-agent collaboration for agents running in the environment. Section 4 concludes the contribution of this work and a comparison with related work.

2. Design for adaptive behaviour

Various design approaches have been developed in an attempt to give objects flexible behaviour. Many of these have proved their usefulness and are widely accepted. Since the value of these approaches for object-oriented systems is recognised, before considering agent adaptivity, it is important to review this body of knowledge in order to gain an insight into their rationale that might similarly be applied to agents. Such knowledge helps in providing a complementary adaptivity approach for agent-oriented systems, making use of their higher level of abstraction.

2.1. Strategy design pattern

The concept of design patterns has been around for some time now, and addresses reoccurring problems by reusing existing proven design solutions, the best known patterns having been documented in [6]. Some patterns have been discovered that describe ways of adapting behaviour, without the need to rewrite them. One such pattern is the *Strategy Pattern*. This pattern is used where a client is served by one of many alternative mechanisms, depending on policy criteria. Shown in Fig. 1, the pattern allows different algorithm implementations to be interchangeable transparently from the client.

2.2. Other patterns

Dynamic selection of algorithms is not the only means of achieving adaptivity. Also important are the capabilities of creation and use of object types with particular behaviour, addition of structuring attributes to those object types; and flexible composition. Each of these provides different facets of adaptivity. Major design patterns that attempt to address these facets have been investigated. The *Abstract Factory Pattern* encapsulates families of related objects and shields the client from the creation process. Different concrete factories are used to create particular product objects with different implementations, exchanging product families being supported by configuration. The *Adapter Pattern* allows the reuse of an existing class, the interface of which does not match the one the client expects, by using an adapter class. The *Composite Pattern* defines a class hierarchy made up of primitive objects and composite objects, the latter being composed recursively by primitive and composite objects into arbitrarily complex structures. New components can be accommodated into the pattern and the client treats composite objects and individual ones uniformly. The *Decorator Pattern* adds responsibilities to individual objects dynamically.

Generally speaking, design patterns help developers to use common solutions to common problems in given contexts. Fundamentally, design patterns encapsulate certain changeable constructs such as system structure or behaviour into configurable metadata (for example, a policy for Strategy Pattern or an adapter class for Adapter Pattern) to provide adaptive solutions, so reducing the required maintenance effort. Individual patterns each address one aspect or another of the adaptivity issue by using a specific pattern, suitable in a certain context. However, when many aspects of adaptivity are required at once, which is the normal situation, the use of all required design patterns at the same time could be almost impossible and impractical. Even combinational use of a few of them demands highly creative and subjective design skills and might lead to excessive extra complexity. Three design patterns, namely Observer, Strategy, and Facade have been used together to try to achieve adaptivity. The Observer Pattern allows a set of objects to be notified on a given state change. The Strategy Pattern describes the implementation of an object that can dynamically change its behaviour. The Facade Pattern provides an interface to encapsulate a set of objects. Combining these three patterns could mean that when a state change occurs, notification is generated (Observer Pattern) and a dynamic change of behaviour is imposed under control (Strategy Pattern), the functionality of both being encapsulated using a proper and unified interface (Facade Pattern) [2]. Though a system designed in this way has improved adaptivity, the resulting design is very complex due to the increased number of classes and the increased communication overhead.

2.3. The Adaptive Object Model (*AOM*)

Design patterns are not designed exclusively to cope with the adaptivity issue but provide common design solutions to common issues. The building of several patterns fully dedicated to adaptivity must address adaptivity better. The *Adaptive Object Model* (AOM) is such an example. The AOM is described in [17] as a framework that models business units with metadata, which will be interpreted at runtime. This is achieved using the *TypeObject*, *Property* and *Strategy* patterns [18]. The TypeObject Pattern uses an EntityType-Entity structure to replace the Class-Subclass structure, making unknown subclasses simple instances of a generic class. This avoids the problem where there are an unpredicted number of new classes by allowing their creation at runtime from generic types in the same way as objects are instantiated from classes. New business entities therefore can be dynamically defined for the system. The Property Pattern in AOM uses an Entity-Property structure to separate an object from its property which encapsulates a collection of attributes. This allows objects of different types but the same class to have

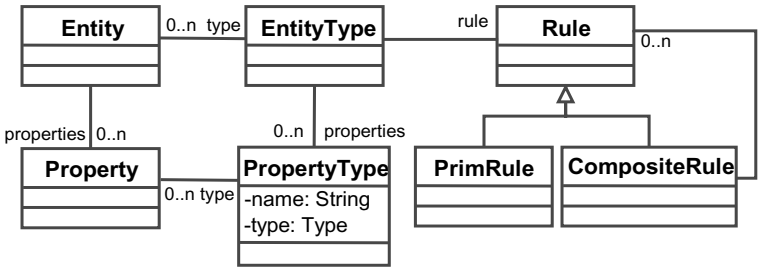

Fig. 2. Architectural Patterns of the Adaptive Object Model.

different sets of attributes. AOM also uses the established Strategy Pattern to allow dynamic behaviour to be configured for classes as described in Section 2.1. Further, the Composite Pattern is applied here to evolve strategies, including composed primitive and composite rules. Dynamic association of rules with entity types at runtime can provide the system with the required behaviour. Figure 2 shows the AOM architecture with the TypeObject Pattern applied twice with the Property Pattern and then Strategies Pattern added. The power of the AOM lies in its separation from the system the metadata using a set of patterns that can later reconfigure the system.

2.4. Agent patterns and other agent-oriented approaches towards adaptivity

Using Agent Patterns [3] is one way for better code encapsulation and reuse. In support of Agent Patterns, it is argued that much research work emphasises only the design of basic elements like goals, roles, communications, and so on, whereas the reuse of patterns, which are observed as recurring agent tasks appearing in similar agent communications, can reduce repetitive code. However, the chance that a pattern can be reused without change is low and reuse of patterns in different contexts is not straightforward. In addition, this approach is not adaptive since any system requirements change means that models need to be changed, patterns need to be re-written and agent classes re-generated.

State machines have also been suggested for agent behaviour modelling [1] and the Extensible Agent Behaviour Specification Language (XABSL) has been specified [9] to replace native programming language and to support behaviour modules design. Intermediate code can be generated from XABSL documents and an agent engine has been developed to execute this code. The language is good at specifying individual agent behaviour, but cannot express behaviour that involves inter-agent collaboration. Moreover, although agent behaviour is modelled in XABSL, it must be compiled before being executed by the agent engine. Thus, changing the XABSL document always requires re-compilation.

Agent behaviour is modelled as workflow processes in [8] and a Behaviour Type Design Tool is described for constructing behaviour. This approach provides a convenient way to compose agent behaviour visually. However, its use of Agent Behaviour Representation Language (ABRL) to describe agent interaction scenarios and “guard expressions” to control the behaviour execution order does not facilitate the modelling of systems as a whole. Further, the approach does not offer an agent system generation solution.

All of the above approaches promote module reuse but do not build an architecture suitable for the reuse of modules using metadata abstracted appropriate to agents. Usually code change is still required, other complexities introduced, and the abstraction of agent over object not fully exploited. Nevertheless, the idea of modelling reusable modules and forming metadata is recognised, being potentially useful for engineering MAS environment towards adaptivity.

3. The adaptive agent model – Supporting system adaptation under environmental changes

Design patterns can make object-oriented systems more adaptive. To date MAS has received little attention in the use of equivalent techniques such as where metadata is abstracted and models are provided for the dynamic management of requirement changes through the offered environment. Our work concentrates on building an adaptive agent model in which agent behaviour and interaction patterns are fully flexible. Adaptivity is addressed in two main aspects: computational functionalities and dependencies. We will demonstrate the development of adaptive agents based on JADE [7] and Java using the Adaptive Agent Model (AAM) in the rest of this section. Before that, we will give an overview of the primary elements of the framework. These define the AAM environment and the entities involved in its environment along with their rationale.

Agent: A conceptual unit for capturing requirements and a software unit for realising responsibilities related with the relevant requirements. Agents interact with one other by passing messages. Agents use knowledge in rules to process incoming messages and produce outgoing messages, contributing to goals and objectives they are expected to meet.

Rule: A captured functional requirement that is configurable at runtime. Rules constitute externalised agent knowledge. Agents use rules to understand and respond to messages, make decisions, and collaborate with each other. A collection of rules compose and define agent interaction protocols. An agent chooses various rules to play various roles in interactions protocols.

Class: A traditional passive component. Class objects respond to active agents when they are invoked, thus assisting in realising the behaviour of the running agents. Such agent-class collaborations are defined in rules.

Message: An objects container passing between agents. Messages with objects encoded in them are known by agents that create them and are expected by agents that receive them, if related rules are defined. It is also defined in rules what objects are encoded at the sending side, and how they are decoded at the receiving side. The passing of a message indicates the sender has made its contribution towards a business goal and now the receiver takes its responsibility to contribute to the same goal.

Environment: A three layer environment is required to run the system. The first layer consists of agents interacting with one another, passing messages in the environment, and retrieving behavioural knowledge from the next layer. The middle layer is a structured knowledgebase of rules, supplying these to agents from the previous layer and referring to and requiring the components from the next layer. The last layer consists of component services, ready to be invoked to facilitate the system to function. The knowledge in the middle layer is expected to be updated continuously during the running of the system corresponding to changing requirements at runtime.

3.1. Java agent development framework

Java Agent DEvelopment Framework (JADE) [7], fully coded in Java itself, is a FIPA (IEEE standards organisation) [4] compliant framework, popular for developing Java agents. JADE provides a distributed agent platform, which includes the Agent Management System (AMS), maintaining a directory of agent identifiers (AID), the Directory Facilitator (DF) offering a 'look-up' service, and the Agent Communication Channel (ACC) controlling message exchange within the platform. Each JADE agent is an instance of a user defined Java class. An agent can have multiple tasks executing concurrently, each implemented as one or more behaviours. JADE starts up an agent by giving it an AID, registering it with the AMS, and executing a *setup*() method, in which customised behaviours can be added using the *addBehaviour*() method. Multiple behaviours can be added, scheduled, and executed starting with

```
/* CustomerAgent.java */
public class CustomerAgent extends Agent {
   private AID sellerAgent;
   // this method is to be invoked by GUI when an enquiry is required
   public void enquiry(String selectedItemfromGUI) {
      ACLMessage msg = new ACLMessage(ACLMessage.REQUEST);
      msg.setSender(getAID());
      msg.addReceiver(sellerAgent);
      Item item = new Item(selectedItemfromGUI);
      Enquiry enquiry = new Enquiry(getAID(),item);
      // Enquiry as an action requests the seller to look up the item price
      Action action = new Action(sellerAgent,(AgentAction)enquiry);
      try {
         getContentManager().fillContent(msg, action);
         send(msg);
      } catch (Exception e) {}
   }……
}
```

Fig. 3. Code segment for customer agent.

the first one in the queue after setup. Behaviours can also be removed whenever necessary by using the *removeBehaviour*() method. A *takedown*() method can be used to implement any cleanup operation before the agent thread is destroyed, while *doDelete*() stops agent execution immediately. Agents communicate via Agent Communication Language (ACL) messages. Attributes can be set to ACL messages to refer to different performatives (purposeful actions performed during conversations between the agents) such as REQUEST, AGREE, INFORM, and so on. Java objects can be encoded into messages as 'payloads' and reconstructed later. In the messages, the AID of the receiver agents can be specified as a message destination, the ACL expression mapping to and from Java objects as message contents. An agreed-upon common language and ontology provides the syntax and semantics for the conversations. The methods *setContentObject*() and *getContentObject*() can be used to pass Java objects through agent communications. A *send*() method allows an agent to send a message. Messages received by an agent are put into the agent's private queue, typically accessible by using the *receive*() method.

3.2. Intra-component adaptivity

JADE agents implemented as Java objects are not adaptive since agent behaviour is decided at design time and written in fixed code. Therefore, a new behavioural pattern is required. We propose to externalise context dependent metadata and use agents to interpret their behaviour dynamically at runtime from it.

Case 1: Suppose a buyer agent is communicating with a seller agent. The buyer has obtained from the seller a list of currently available goods and now selects items and requests their corresponding prices. Upon any enquiry, the seller agent retrieves the goods details including default prices. Discount policies subject to amendment exist, specifying in various conditions the discounts that buyer can enjoy. Thus, appropriate discounts applicable to the current buyer according to the customer profile are applied before the actual prices are presented to the buyer. Figure3 illustrates buyer and seller agents as implemented using JADE.

Figure 3 shows how when the customer has chosen items from a user interface, the customer agent responsible for the shopping task of this customer contacts the seller by constructing a new message, setting its destination and action to perform, and finally sending it off. "Item" is a concept representing goods for sale with a type slot and a price slot. "Enquiry" is an agent action representing an action that

```
/* SellerAgent.java */
public class SellerAgent extends Agent {
   private rulesParser = new qub.aam.common.RulesParser();
   private ContentManager manager  = (ContentManager)getContentManager();
   protected void setup() {
      // check new customer request every second
      addBehaviour(new TickerBehaviour(this, 1000) {
         protected void onTick() {
            ACLMessage msg = receive(enquiryTemplate);
            try {
               if(msg != null) {
                  ContentElement ce = manager.extractContent(msg);
                  if(ce instanceof Action) {
                     Enquiry enquiry = (Enquiry) ((Action)ce).getAction();
                     AID buyerId = enquiry.getBuyer();
                     Item item = enquiry.getItem();
                     /* find and apply the rule that is relevant to customer discount and also applicable to this partic ular customer */
                     String discount = executeRule("customer.discount", buyerId);
                     double price = item.getPrice() * (1 - Double.parseDouble (discount));
                     ACLMessage inform = msg.createReply();
                     manager.fillContent(inform, price);
                     send(inform);
                  }
               }
            } catch (Exception e) {}
         }
      } );
……
   }
}
```

Fig. 4. Code segment for seller agent.

can be performed by some actor agents. Both the concept and the action must be implemented as Java classes and declared in a user-defined ontology that will be shared among communicating agents before being put into use. Here "Enquiry" as an action encapsulating the requesting buyer and the requested item can be sent as an action by the customer to the seller and understood by both agents to perform a price enquiry.

When a message that fits in the context of this enquiry conversation is received, the content of the message is extracted, and the action object restored, in this case an "Enquiry". The Java code for the seller agent is shown in Fig. 4. To simplify, suppose at one time only one item is selected by the customer for enquiry, then the seller needs to find out the discount applicable in the condition of this customer as a potential buyer of the specific item at this particular time. Because business policies must change continuously to achieve the best business opportunities, using fixed code such as complex logical structures greatly increases the maintenance burden. For this reason we propose to externalise this changeable information into easier to maintain XML-based rules repository, and let agents interpret it and decide the current applicable rules on the fly. A template rule made up of an "IF" condition, a "THEN" action and a priority components alongside its XML format is illustrated in Fig. 5, where op1 and op2 correspond to "less than", "equal to" or "greater than".

Figure 6 gives three concrete and self-explanatory examples of actual rules in our demonstrator system.

These sample rules can be defined on the basis of customer profile. More rules can be defined based on the type of chosen item, the data and time of shopping, and so on to reflect promotion on certain types of goods, or reduction of prices during off-seasons, and so on.

```
IF objectName1.attributeName1 op1 value1
THEN objectName2.attributeName2 op2 value2
Priority value3
------------------------------------------
- <rule>
    <id>ruleId</id>
    <condition>
      objectName1.attributeName1 op1 value1
    </condition>
    <action>
      objectName2.attributeName2 op2 value2
    </action>
    <priority>value3</priority>
  </rule>
```

Fig. 5. Rule template.

i) *IF customer. name = Liang Xiao*
THEN customer. discount := 15% Priority: 5

ii) *IF customer. occupation = researcher*
THEN customer. discount := 10% Priority: 3

iii) *IF customer. type = premium*
THEN customer. discount := 5% Priority: 2

Fig. 6. Sample rules.

The rules are executed by agents by using a facilitating RulesParser module, shown in Fig. 7, which interprets the operational Java objects from the XML code. Rules relevant in the context of customer discount are retrieved, evaluated in the order of ranked priority and the rule with the highest priority evaluated as being satisfied would be eventually executed, returning a discount value. The method of testCustomerCon() retrieves the customer profile and uses this information to evaluate the rule condition. We omit the concrete code of the method here in the interest of conciseness. However, another interesting process is involved. The seller agent does not hold all customer information, since this is managed by the customer agent. Thus, the seller agent must request customer information from the customer agent in order to calculate the actual price with regard to the discount. The process is adaptive in two aspects. On the one hand, the rules set in the business environment are dynamically defined, so that the rules being enquired upon are changing over time. On the other hand, at any given time, enquiries on rules are performed dynamically because customer satisfaction against the same set of rules varies. Discounts to be applied are decided by these two factors. For example, three rules in Fig. 6 will be tested sequentially. Thus a researcher named "Liang Xiao" can enjoy a discount of 15% as a result of the first rule while another researcher named "Des Greer" does not satisfy the first rule but the second one. Details of the two aspects of adaptivity follow.

Suppose a Rules Manager Agent (RMA) for the general purpose of managing the rules repository is dedicated to evaluate rules in order of priority, instead of specific agents individually doing the same routine of rule evaluation and execution. Ordinary agents need only contact the RMA and get the computed result. The RMA retrieves and ranks rules each time a request is received, thus guaranteeing that the updated list of rules is available. Each time the top ranked rule is found unsatisfactory, the RMA switches to the next one, and so on, until the satisfactory rule with the highest priority is found. The agent responsible for the customer "Des Greer" and the RMA communicate as shown in Fig. 8. This extended UML diagram shows agents, rules, and their interactions via message passing. R1 is rejected when it is found that the name of the customer does not match with what is required by the rule R1. The

```
/* executeRule is used by SellerAgent to apply the applicable rule */
private String executeRule(String actionTarget, AID buyerId) {
   Vector globalRules = rulesParser.getRules();
   Vector con = (Vector)globalRules.elementAt(0);
   Vector act = (Vector)globalRules.elementAt(1);
   Vector pri = (Vector)globalRules.elementAt(2);
   // filter rules and obtain only those relevant to customer discount
   for(int i = 0; i < act.size(); i++) {
      if(((String)act.elementAt(i)).startsWith(actionTarget)) {
         filteredCon.addElement(con.elementAt(i));
         filteredAct.addElement(act.elementAt(i));
         filteredPri.addElement(pri.elementAt(i));
      }
   }
   /* recursively retrieve the rule with the highest priority and test if the rule is satisfied in accordance with the current condition until one
rule is found to be applicable */
   for(;;) {
      int highestPri = 0;
      int highestPriIndex = 0;
      // get the index of the rule with the highest priority
      for(int j = 0; j < filteredPri.size(); j++) {
         if(Integer.parseInt((String)filteredPri.elementAt(j)) > highestPri)
         {
            highestPri = Integer.parseInt((String)filteredPri.elementAt(j));
            highestPriIndex = j;
         }
      }
      /* test this rule against the condition of the customer, if satisfied then return the corresponding discount */
      if(testCustomerCon((String)(
         filteredCon.elementAt(highestPriIndex)), buyerId)) {
         String action = (String)filteredAct.elementAt(highestPriIndex);
         int index = action.indexOf("=");
         String discount = action.substring(index + 2);
         return discount;
      }
      // else remove this rule and return to loop for the next run
      else {
         filteredCon.removeElementAt(highestPriIndex);
         filteredAct.removeElementAt(highestPriIndex);
         filteredPri.removeElementAt(highestPriIndex);
      }
   }
}
```

Fig. 7. Code segment for rule execution.

RMA then switches to R2, which is satisfied because the customer is found to be a researcher when the information is obtained from the customer. R3 is never used. This process is dynamic. For example, the RMA will apply R1 to a customer named “Liang Xiao” and the process will finish, while R3 would be applied to a premium customer with no research background. Rules can be externally changed as required and will take effect automatically without amendment of the seller agent code (Fig. 4) or rule execution process code (Fig. 7).

All rules in the repository are formatted in XML style conforming to the template shown in Fig. 5. Therefore generically, the steps the RMA takes to find and execute rules on behalf of other agents are:

1. Get a list of relevant rules according to the <action> tag.

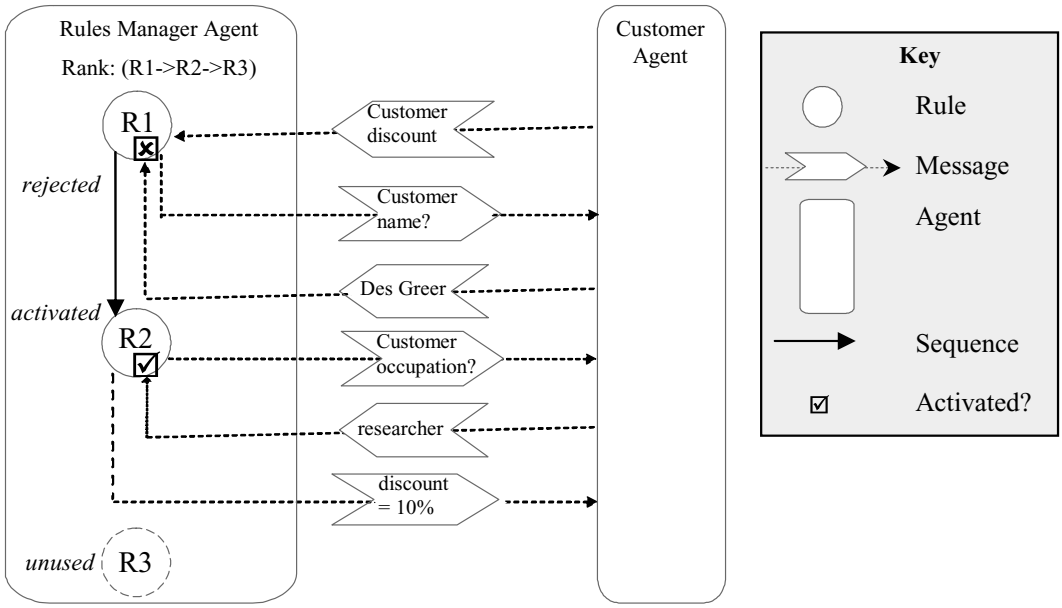

Fig. 8. Message passing sequence diagram.

2. Get the rule that currently has the highest priority according to the <priority> tag.
3. Send messages to the initialising agent asking about context information it holds according to the <condition> tag.
4. According to the returned result, if the condition is satisfied, then reply to the initialising agent with the appropriate value set in the rule. Otherwise, remove this rule from the rules set and go to Step 2. A default value is returned, if this is the last rule in the rules set.

The provision of tools for business analysts and strategy makers adds to the dynamic capability of the approach by allowing on-the-fly configuration of policies. Figure 9 shows a web-based editor that has been developed. The business entities which are used to compose these rules are abstracted and listed on the interface for simple selection, so that people who have no programming experience can specify relationships as rules in a straightforward way. The current available rules list is retrieved from the XML repository and shown on the interface for viewing and editing. Whenever rules are updated from the interface, the repository is updated accordingly. This easily allows the adjustment of existing business strategies and the addition of new ones. For example, a new policy with the highest priority among all policies may be defined that grants all staff from a particular university a discount of 30%. Then customer "Des Greer", being a staff from that university can benefit from this new policy since it overrides the others.

With the rule prioritisation mechanism, the imposition of business strategies can be shifted from one set of rules to another freely and easily. Rule definitions can be preserved and instead priorities of them are changed to reflect the current needs. The rules are thus structured like a class hierarchy. Rules which are applicable with higher priorities, like specific subclasses in the class hierarchy override those with lower priorities, like generic classes in the class hierarchy. The lower the priority of rules, the more common are the conditions where they are applicable and the more likely they are to be overridden by more specific ones. This architecture enables a flexible and maintainable business configuration system.

Figure 10 shows the interactions between Ordinary Agents and the RMA which involve the dynamic message-passing process from Fig. 8, and the role of the rules editor shown in Fig. 9 which updates

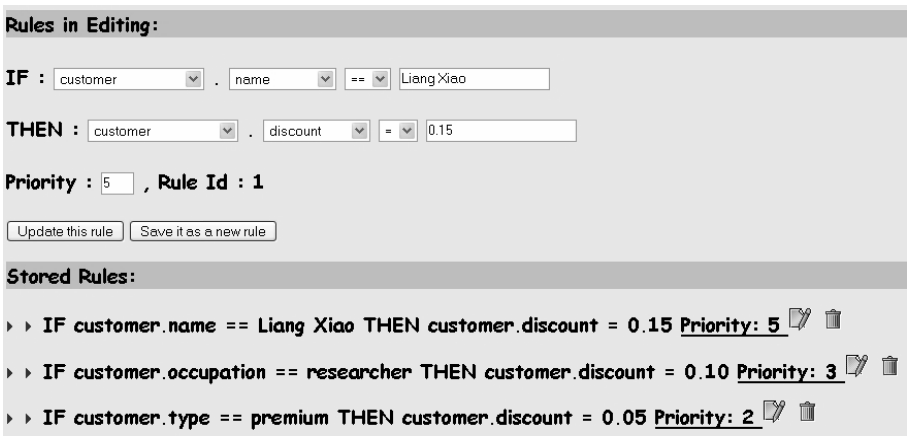

Fig. 9. Rules editor interface.

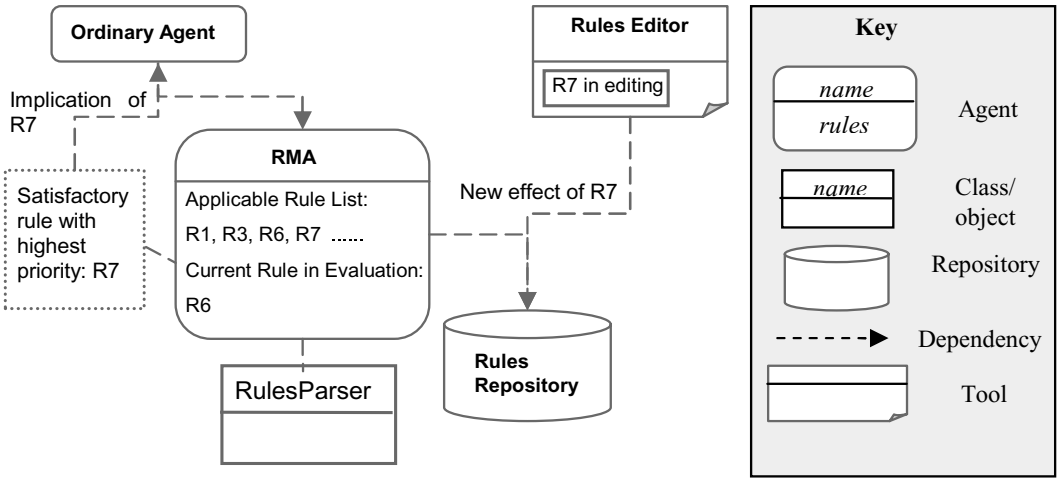

Fig. 10. AAM adaptive function pattern.

the rules repository. By replacing previously fixed agent behaviour by dynamic rule execution which includes an iterative interaction process evaluating externalised rules, agents can now behave adaptively not only to the current conditions of that particular agent but also the policies set at that moment. Recall that in using the Strategy Pattern all strategies must be predicted, in a hard-coded, non-configurable way. This limitation has been overcome here by the separation of changeable policies from the main agent programme. This subsection demonstrates the adaptivity achieved through externalised policies. Additional support for adaptive use of an ontology and adaptive external class method invocation in relation with this technique is found in [14,15].

The environment presented in Fig. 10 provides a mechanism for achieving dynamic MAS infrastructure and reconfiguration. This fits in one of the configuration schemes for agent applications and their environment as categorised in [12], environment being used for adaptive structure of software systems. In this literature, it is proposed that changes in the software systems are important events that must be addressed by agents via the environment as an intermediary. The environment provides a reflection of the structured software system and lets agents be aware of the extra/changing functionalities required.

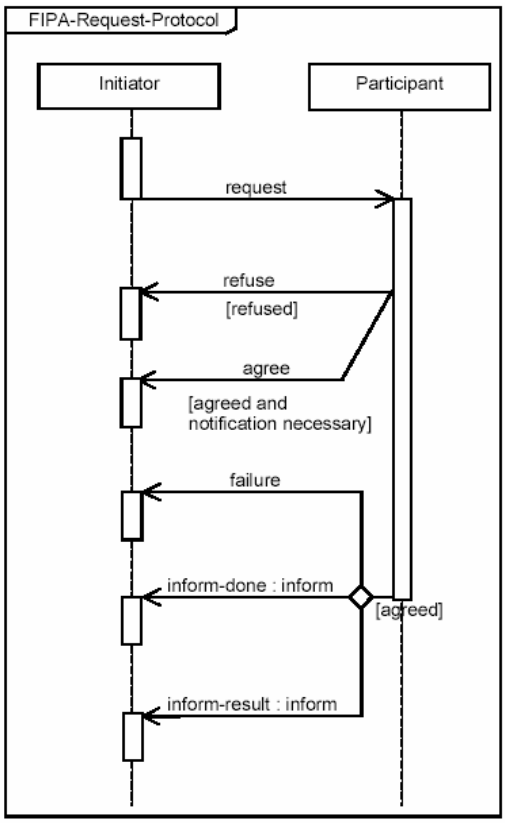

Fig. 11. FIPA-Request protocol [4].

The MAS becomes more extendable and reusable through its deploying environment in this view.

Here in the AAM, the rules repository, the rules editor, the rules parser, and the RMA together form an environment within which agents are dynamically informed of the current policies and strategies for application from alternatives. The rules repository provides storage of such knowledge in the environment. The rules editor provides an interface between the system and human experts who supply their needs to the system and make them available in the environment. RMA facilitates ordinary agents to apply the appropriate portion of the knowledge as provided by the environment. Finally, the rules parser supports the RMA, the facilitator, to better understand the knowledge offered in the environment.

Such an environment is not fixed but rather highly dynamic, reflecting the changing requirements, these being immediately available for agents. The environment provides opportunities for agents to adapt their behaviour and the overall system reconfigures its infrastructure. It is through the environment composed of the multiple entities shown in Fig. 10 that intra-agent function adaptivity is achieved. A complementary environment will be introduced in the next section for inter-agent collaboration adaptivity.

3.3. Inter-component adaptivity

Considering the interactions shown in Fig. 8 it is obvious that dynamic processes like this are normal but can not be foreseen in advance. Support for adaptivity of agents in choosing at runtime other agents with which to form dynamic interaction patterns is difficult using traditional technologies or platforms.

```
/* the customer requests the catalogue from the seller */
/* CustomerAgent.java */
ACLMessage requestCatalogue =
   new ACLMessage(ACLMessage.REQUEST);
requestCatalogue.setProtocol
   (FIPANames.InteractionProtocol.FIPA_REQUEST);
requestCatalogue.addReceiver(sellerAgent);
   /* requestCatalogue message has been sent that initialises the conversation, an INFORM message is received and being handled */
addBehaviour(new AchieveREInitiator(myAgent, requestCatalogue) {
   protected void handleInform (ACLMessage inform) {
      productCatalogue = (Vector)inform.getContentObject();
      myGUI.update(); // update user GUI with the up-to-data catalogue
   }
} );
/* SellerAgent.java */
MessageTemplate requestTemplate = MessageTemplate.and
(MessageTemplate.MatchProtocol(FIPANames.InteractionProtocol.FIPA_REQUEST), MessageTemplate.MatchPerformative(ACLMessage.
REQUEST));
addBehaviour(new AchieveREResponder(myAgent, requestTemplate) {
   /* this method is called when the initiator's message is received that matches the message template passed in the constructor: if a
REQUEST message as specified in a FIPA REQUEST interaction protocol is received, an AGREE message will be sent back as a result */
   protected ACLMessage prepareResponse(ACLMessage request) {
      if (checkCustomer(request.getSender().getName())) {
         // the seller agrees to inform the customer the catalogue
         ACLMessage agree = request.createReply();
         agree.setPerformative(ACLMessage.AGREE);
         return agree;
      } // else send a REFUSE message
   }
   /* this method is called after the AGREE message has been sent, an INFORM-RESULT message will be sent back as a result */
   protected ACLMessage prepareResultNotification(ACLMessage request, ACLMessage response) {
      // if the catalogue is prepared, send it to the customer
      if (checkCatalogue()) {
         ACLMessage informCatalogue = request.createReply();
         informCatalogue.setPerformative(ACLMessage.INFORM);
         informCatalogue.setContentObject(productCatalogue);
         return informCatalogue;
      } // else send a FAILURE message
   }
} );
```

Fig. 12. Code segment of catalogue request conforming to the FIPA-Request interaction protocol.

Objects normally have to interact in a fixed mode. This restriction on component dependencies must be removed away from agents to allow free communication even in unpredicted patterns arising from new needs or goals. This idea of agent society has led to the term multi-agent systems (MAS) where agents can coordinate through cooperation or competition in different conditions for different purpose, resembling human society. Such systems are highly dynamic and must be adaptive in terms of internal communication.

FIPA specifies a set of standard Interaction Protocols: FIPA-Request, FIPA-Request-When, FIPA-Query, FIPA-Subscribe, and so on. These can be used as agent conversation templates. JADE distinguishes the Initiator role and Responder role, being agents starting conversation and joining conversation after contact. AchieveREInitiator and AchieveREResponder implement the roles required by most protocols. For example, the FIPA-Request protocol as shown in Fig. 11 specifies that when one agent requests another to perform some action, the participant can decide to refuse or agree. If it is agreed,

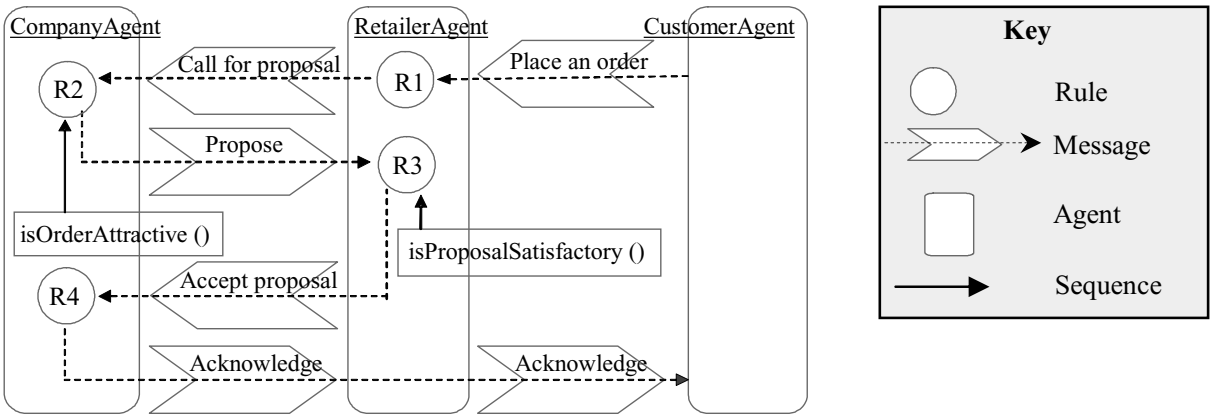

Fig. 13. Message passing sequence diagram for changed inter-agent relationships.

another message is followed, either indicating its failure in its attempt to fulfil the request, or informing its success in the completion of the request, or informing the result of the successfully completed quest. In such an interaction process, all involved agents must tag all their ACL messages with a globally unique, non-null conversation identifier, and also the current states of the conversation in order to manage their communication process. For instance AchieveREResponder which implements the responder role requires pattern matching of messages by using message templates to judge if the received message is expected by a protocol and to be processed at a particular moment. If this is the case, suppose all other business dependent terms are satisfied, then it calls a *prepareResponse*() method to send the first response (for example the "agree" message), and *prepareResultNotification*() method to send the last response (for example the "inform-result" message). Such processes have the same pattern every time the agents interact and implement pre-defined protocols.

The code in Fig. 12 is extracted from our CustomerAgent and SellerAgent programmes and is as required by JADE. It is used when the customer requests the available goods list from the seller before he/she chooses items and the seller evaluates the discount. The customer agent must initially construct and send a REQUEST message, the seller agent must respond to the request an AGREE/REFUSE message, followed by an INFORM-RESULT/FAILURE message, and finally the customer handles the result. The JADE programming model must be followed exactly to implement the protocol defined in Fig. 11.

The fixed dependencies as required by the FIPA/JADE interaction protocols impose a restriction on adaptive collaborative behaviour in agents. We propose to overcome this restriction and achieve adaptive inter-component dependencies by using XML-based rules to externalise dependencies as metadata. Together with the intra-component adaptivity for individual component functions presented in Section 3.2 they comprise a uniformed Adaptive Agent Model.

In the real world, it is completely possible that, due to changed business strategies, a third party is introduced into our case study as a mediator between the seller end and the customer end at any moment while the system is running. Case 2 provides an example.

Case 2: It could be required now that a customer contacts a local retailer, which supplies goods to customers from various supplier companies, who may or may not serve the retailer. Overall, the relationships between the customers, retailers, and supplier companies can change at any time. A customer may buy goods from another retailer if he/she is not happy with the current one. A retailer may withdraw an existing partnership with a supplier company due to an unsatisfied price offer.

```
<?xml version="1.0"?>
<xs:schema targetNamespace="http://www.cs.qub.ac.uk/~L.Xiao/"
xmlns:xs="http://www.w3.org/2001/XMLSchema"
xmlns="http://www.cs.qub.ac.uk/~L.Xiao/">
<xs:complexType name="msgcontent">
   <xs:sequence>
      <xs:any minOccurs="0"/>
   </xs:sequence>
</xs:complexType>
<xs:complexType name="msg">
   <xs:sequence>
      <xs:element name="from" type="xs:string"/>
      <xs:element name="to" type="xs:string"/>
      <xs:element name="title" type="xs:string"/>
      <xs:element name="content" type="msgcontent"/>
   </xs:sequence>
</xs:complexType>
<xs:complexType name="eve">
   <xs:sequence>
      <xs:element name="type" type="xs:string"/>
      <xs:element name="message" type="msg"/>
   </xs:sequence>
</xs:complexType>
<xs:complexType name="act">
   <xs:sequence>
      <xs:element name="type" type="xs:string"/>
      <xs:element name="message" type="msg"/>
   </xs:sequence>
</xs:complexType>
<xs:complexType name="con-acts">
   <xs:sequence>
      <xs:element name="con-act" maxOccurs="unbounded" minOccurs="0">
         <xs:complexType>
            <xs:sequence>
               <xs:element name="condition" type="xs:string"/>
               <xs:element name="action" type="act"/>
            </xs:sequence>
         </xs:complexType>
      </xs:element>
   </xs:sequence>
</xs:complexType>
<xs:complexType name="vars">
   <!-- definition similar with the previous type -->
</xs:complexType>
<xs:element name="rule">
   <xs:complexType>
      <xs:sequence>
         <xs:element name="name" type="xs:string"/>
         <xs:element name="business-process" type="xs:string"/>
         <xs:element name="owner-agent" type="xs:string"/>
         <xs:element name="global-variable" type="vars"/>
         <xs:element name="event" type="eve"/>
         <xs:element name="processing" type="xs:string"/>
         <xs:element name="condition-action" type="con-acts"/>
         <xs:element name="priority" type="xs:string"/>
      </xs:sequence>
   </xs:complexType>
</xs:element>
</xs:schema>
```

Fig. 14. Agent behavioural rule schema.

Event: receive a "Call for proposal" message from "RetailerAgent";

Processing: decode the proposal message, construct new objects relating to the order that the customer has been placed through the retailer, and possibly create a proposal object according to the order;

Condition: check this order of its attractiveness;

Action: if the order is attractive, encode a proposal object into a message and send the message to "RetailerAgent".

Fig. 15. Rule R2 as shown in the diagram of Fig. 13 performs according to the schema defined in Fig. 14.

The agent interactions for this scenario are shown in Fig. 13. In this example, an order is placed by a customer, processed by a retailer who, in turn makes a call for proposal to a company. The company replies with a proposal, if the order is attractive and the retailer accepts it, if the proposal is satisfactory. The process completes with an acknowledgement from the company to the retailer who then acknowledges the customer.

Using most existing approaches it would be expected that the system would need a complete reconstruction after such a dramatic change, even if using agents and so requiring the introduction of a new agent and new dependencies of the existing agents. Our proposed adaptive agents, however, can avoid such code surgery since previously required behaviour is not coded directly, but instead interpreted at runtime from rules as shown in Fig. 13, the definitions of which are externalised in an XML repository.

Here rules as previously specified in Figs 5 and 6 have been extended. They are enriched in format and also include complete content, not only rules processing but also their ownerships as well as incoming and outgoing parties under various conditions. Now rules can be used to adapt both individual agent behaviour and inter-agent collaboration. Specifically, the shared schema specification as shown in Fig. 14 declares the behavioural pattern of agents: upon receiving a triggering event, after an event processing, a series of actions to perform if corresponding conditions are satisfied, and the priority sorting rule orders. Concrete rules in this format, found in [14,15], are demonstrated in actual business applications. According to the schema, multiple {condition, action} pairs are allowed. Event and action sections detail message passing. Message contents can include any encoded object required by business needs.

A rule definition is made up of the steps that an agent takes to execute the rule. Those steps are as follows.

1. Check event. Find out if the rule is applicable to deal with the perceived event.
2. Do processing. Decode the incoming message, including the construction of objects to be used in later phases.
3. Check condition. Find out if {condition c_i} is satisfied.
4. Take an action. If c_i is satisfied, then do the corresponding {action a_i} that is related with {condition i} as defined by the rule. Then, send a result message to another agent. If c_i is not satisfied, and this is not the last condition, then go back to Step 3 and check the condition c_{i+1}.

Figure 15 shows the processing components of the rule R2.

The addition of a retailer to an existing customer-seller structure can be accommodated simply by altering the current rules to whatever specific requirements have arisen. Also, the original difficulty of agent behaviour reuse, such as subclasses implemented and embedded in agent classes in JADE is now solved by simple configuration of rules using a template defined by the schema.

The same RulesParser that has been used as a JavaBeans component serving the Java ServerPage based rules editor shown in Fig. 9, a facilitating module by the customer agent to retrieve and execute applicable rules, is used a third time here by the Java Swing based diagrammatic editor. Figure 16 shows

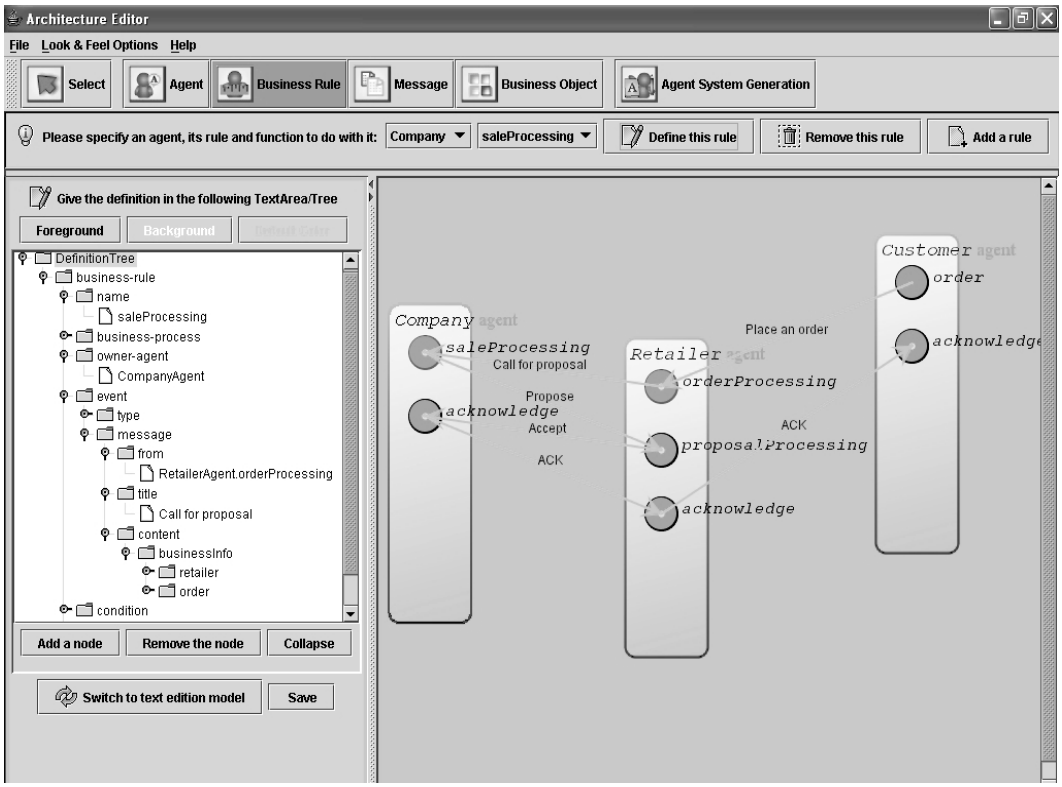

Fig. 16. Tool support for inter-agent collaboration specification.

the construction of agent communication diagrams in its main panel and the definition of rules via a tree structure. The tree structure realises the XML schema defined in Fig. 14 and XML based rules are generated from it and stored in the repository. These rules decide the agent communication pattern shown diagrammatically. The edition of diagrams in the main panel passes the specified agent relationships to affected rules in the left panel and vice versa so that diagrams and trees are consistent. The RulesParser interprets XML rules and shows them on the tool for business customisation and it also enables agents to execute rules in a similar pattern like the customer agent does as shown in Fig. 7. When the agent system is running, both this tool and the web based one can be used by business people continuously to maintain business knowledge. Agents always use the up-to-date rules and bring business requirements into reality as soon as they become available through the configuration using the tools. Figure 17 summarise the process and demonstrates the architecture of the system extended from Fig. 10, agents collaborating freely and forming any required interaction pattern by specifying condition/action pairs.

Additional support of adaptive use, invocation, and replacement of external classes in support of rule executing, for example, the evaluation of order attractiveness method is found in [15]. It would be useful when an alternative version of a class can be found and used to replace an old version by agents on the fly to achieve a changed effect.

Similar to what has been discussed at the end of the previous section, the rules repository, the rules editor, and the rules parser together form an environment as shown in Fig. 17. This environment supports agents to dynamically deploy requirement changes with regard to inter-agent collaboration. Agents thus can opportunistically collaborate with the appropriate partners to achieve their (changing) goals. Again,

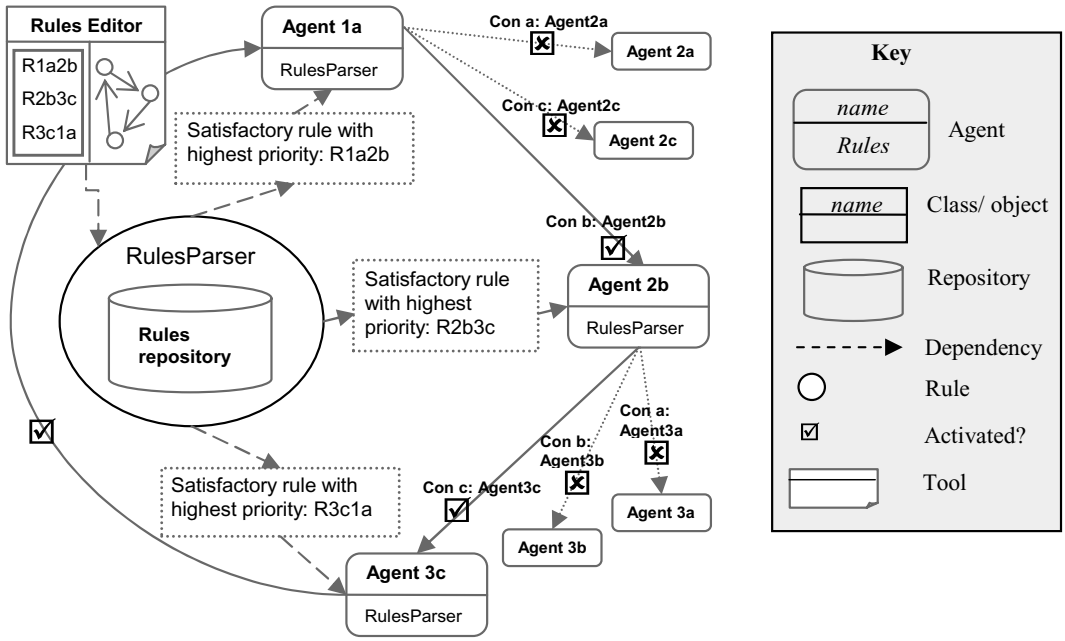

Fig. 17. AAM adaptive dependency pattern.

it is the use of such an environment that opportunities are provided for agents to adapt their behaviour and the overall system reconfigures its infrastructure.

By providing observable entities and sharable communications, the environment is believed to be able to raise the level of awareness of interactions among agents [11]. In surveys of the notion and use of the environment in MAS, the environment is found potentially useful as mediation or channel for flexible agent interaction [10]. In the AAM, this is realised and reflected in the explicit modelling of inter-agent collaboration model. With the interactive behaviour rules, agents strategically make decisions about their actions towards the environment when they perceive events received from the environment. Pulling together such behaviour patterns of individual agents, the overall system can adapt and reconfigure its infrastructure via the environment.

Concerning both patterns of the AAM along with the tools they provide as presented in this section and the previous one, we have offered an integral environment that supports system adaptation under changing requirements in two dimensions: intra-agent function and inter-agent collaboration.

To summarise, Fig. 18 illustrates the deployment diagram of the AAM environment. Here, two types of rules are integrated in the model repository, along with the concepts that are used to make up them, being configured by a set of editors. The overall model that resides in the repository is applied at runtime by the AAM MAS system upon underpinning agent platforms (e.g. JADE). The execution of the model by agents is facilitated by a Rules Manager Agent (RMA), a Facts Manager Agent (FMA) and a Class Manager Agent (CMA) for populating runtime fact knowledge to agents [14]. Business objects are invoked by agents on demand as acknowledged by the model.

4. Conclusions

We have proposed the Adaptive Agent Model as a way to provide an environment where agents can behave and interact dynamically. A key component in such an environment is the externalised metadata

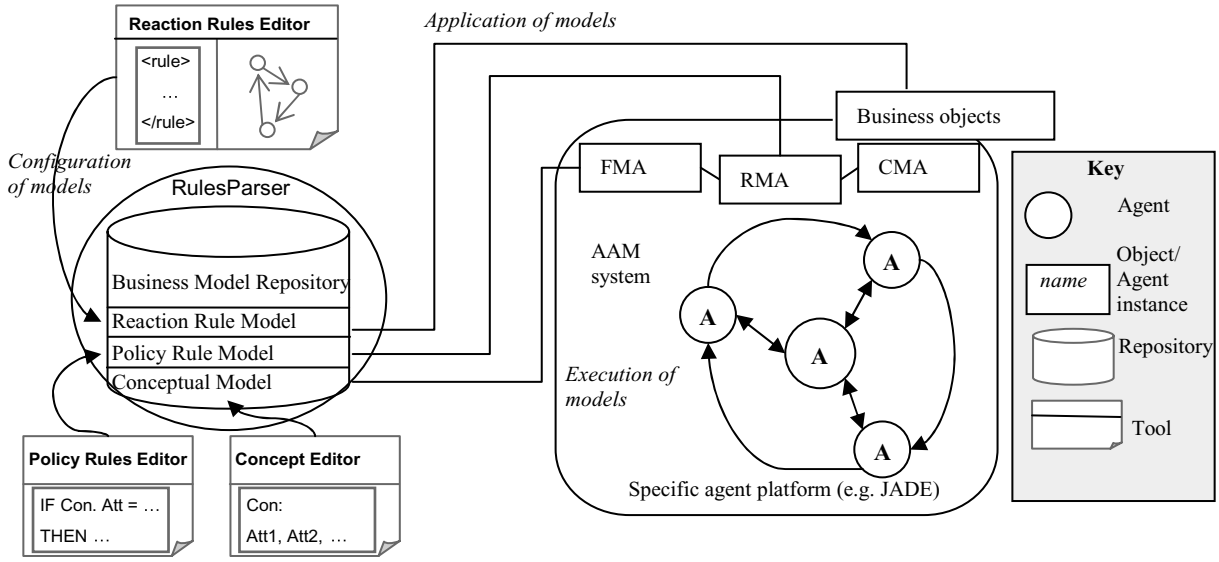

Fig. 18. The deployment of AAM environment.

that software agents use to flexibly interpret their required behaviour. Use of metadata has been proposed elsewhere to reduce duplicated code and hence alleviate programming tasks [5]. A typical technique used is code generation which requires that generation is done with every significant change. Manual changes to generated files are then lost at the next code generation. The AAM does not require generation for each change but instead the change takes effect automatically for use by running agents. The AAM uses rules in XML format to serve as metadata and capture the business needs which can be easily configured using tools. Combining event, condition and action in rules simplifies existing programming models for context dependent behaviour. Agents can, at runtime, decide which rules to apply in changing conditions, process and produce messages flexibly, and send them to dynamically chosen collaborators, being instructed by the metadata stored in XML. The two major adaptivity targets of individual component function and their dependency relationship have been addressed by establishing the Intra-component and Inter-component adaptivity. Using the AAM, existing limitations where behaviour must be pre-defined for each individual agent and pre-formatted interaction patterns imposed upon agents are both overcome. In other papers [14, 15], we have discussed the dynamic evolution of ontology used by agents.

The AOM externalises business definitions as metadata and uses a dedicated interpretation system to carry out the required operations through the metadata on the fly. The idea of externalisation is further advanced in the AAM since the behaviour rules are now modelled in a knowledgebase as part of the environment. The externalisation of agent behaviour in rules allows dynamic component invocation and so achieves runtime adaptivity. The AAM environment manages and maintains the knowledge of the system that is external to running agents. This separation helps the AAM to achieve its configurability. Configuration of the overall architecture in UML-like diagrams adapts global interaction and the configuration of individual rules in XML adapts individual function. There is no need to stop the system while changing rules and making them effective. In addition, the adaptation is in the hands of business people themselves and so they can change the system according to their ever changing business needs, with immediate effect. Businesses are reluctant to shift their computing systems, in which they have already invested and are continuously producing value to the business only because of a perceived aesthetic attraction of new technology. Use of the AAM involves business experts as part of

the environment and also allows their direct impact on their system continuously from the environment, overall system architecture being re-configurable and individual strategies and policies being extensible and amendable. In this way, the acceptance of the AAM can be accelerated.

The AAM environment structures requirements knowledge in a form that is observable and executable to agents and externally maintainable to humans. It is argued in [13] that it is the environment's responsibility to define the rules that inter-relate agents as well as agents and resources (e.g. components and services) that are managed by the environment. The dynamics of the system, independent of agents, can thus be managed by the externally maintained rules residing in the environment. These appear in the AAM as inter-agent collaboration rules and we conceive intra-agent policy rules as the environment's other responsibility, defining the global constraints that agents must conform to. The use of rules in a MAS environment ensures the deployment of the up-to-date requirements. Our rule-based model conforms to the reference model presented in [13], but it is at the same time more practically useful. In the reference model, the behaviour of agent interaction within environment can be decomposed into a set of modules. Among these, "sense", "percept", "action", and "message" are the ones directly associate an agent with its application environment, through the "Perception", "Interaction", and "Communication" functional modules. These resemble the compositional parts of AAM rules (see Figs 15 and 17): initially an agent senses/perceives an "event" from the environment, it then carries out a "processing" procedure internal to the agent, after which a decision is made and an "action" is sent out to the environment. The overall process involves an interaction process and both "event" and "action" parts involve agent communication via message passing. Conforming to the reference model for environment engineering, our rule-based model is implementable and maintainable.

In recent work we have demonstrated the suitability and usefulness of the AAM for constructing adaptive requirements models [16], design models [14], and in particular the building of agent-oriented systems upon object-oriented systems [15]. Overall, the AAM aims at being a complete methodology that: guides the full agent-oriented development process; provides artefacts such as meta-model, structural and behavioural diagrams and XML schema, notations; and offers supporting tools that automate the process and facilitate the design and development. A top-down and a bottom-up approach are used together so that (additional) requirements can be easily assigned to responsible agents and existing systems or system components can be fully reused. Therefore not only agent-focused people can benefit from the AAM by reusing objects but also object-focused people can quickly understand, master, and adopt the agent approach by an extension of the object construct.

The knowledge model repository provided by the AAM enables convenient management of environmental knowledge as well as easy and continuous configuration. To date the model has been demonstrated mainly using business information systems, the need for a centralised model and the cost of accessing this continuously being a known limitation. Thus, at present, the AAM may not be suitable for computation intensive applications or indeed for very stable applications that do not need adaptivity. Additionally, making use of and adapting centralised models could cause problems due to the existence of a single point of failure. A solution to alleviate this problem could be the distribution of models to responsible agents that each will carry out the interpretation and execution of a respective portion. When the models are maintained, the changed parts are pushed to those agents with the affected portions. Alternatively, agents can query and pull the updated version of models before they execute their portion of the models. In this way, a centralised model repository is used for management and configuration and at the same time, agents each keeps a copy of the portion of the models useful to themselves which will be synchronised with the model repository under maintenance and so they are always up to date. We will investigate this and improve the approach in future work.

An opportunity has also been noted in the process of prioritising rules. At present we have ranked rules using an attribute priority, assigned by a business expert. This is a simplified starting point as we seek to establish the principles and utility of the AAM. In the long term this is somewhat naïve since it implies the possibility of a rule being selected despite not being the best rule for a given situation. As an example, in an e-commerce scenario, a user may achieve a more advantageous price if a lower priority rule is chosen. However, if delivery speed is more important the rule should be chosen to provide the best delivery speed. Thus, in future work we will consider the prioritisation of rules as being flexible rather than fixed. In such a scenario, the choice of rule would depend on the goal being sought by the initiating agent. Therefore, with this new approach each rule would have an objective value depending on the goal being sought. The rule with the highest objective value would be the one executed.

Further work is planned to explore environmental factors indirectly decided by the runtime context, along with business models directly decided by business experts, which both can influence the MAS and its running behaviour. An agent may later be able to shift its collaboration with one of several alternative service providers due to its changing beliefs (based on accumulated knowledge supplied via the environment) to the quality of services they provide. For example, an auction site may provide a shared environment with a ranking system for sellers that automatically updates their reputation according to buyer feedback. This process keeps running continuously which always guides buyers to select the most reputed ones from alternative sellers. The AAM will be made more powerful if it incorporates an additional knowledgebase capturing runtime environmental context along with the current business knowledge model together driving agent behaviour. Related with this, we also plan to add more intelligence and autonomy to agents in the AAM so they can explore in the environment freely and build up their own set of rules in addition to the set provided to them. Eventually the aspiration is that the AAM will provide a self-adaptive MAS environment.

References

[1] T. Arai and F. Stolzenburg, *Multiagent Systems Specification by Uml Statecharts Aiming at Intelligent Manufacturing*, Proceedings of the First International Joint Conference on Autonomous Agents & Multi-Agent Systems, 2002, pp. 11–18.
[2] P. Boinot, R. Marlet, J. Noye, G. Muller and C. Consel, *A Declarative Approach for Designing and Developing Adaptive Components*, Proceedings of the Fifteenth IEEE International Conference on Automated Software Engineering, 2000, pp. 111–119.
[3] M. Cossentino, P. Burrafato, S. Lombardo and L. Sabatucci, *Introducing Pattern Reuse in the Design of Multi-Agent Systems*, AITA'02 workshop at NODe02, 2002.
[4] Foundation for Intelligent Physical Agents, http://www.fipa.org/.
[5] M. Fowler, Using Metadata, *IEEE Software* **19**(6) (2002), 13–17.
[6] E. Gamma, H. Richard, R. Johnson and J. Vlissides, *Design Patterns*, Addison-Wesley, 1994.
[7] JADE platform, http://jade.tilab.com/.
[8] G.B. Laleci, Y. Kabak, A. Dogac, I. Cingil, S. Kirbas, A. Yildiz, S. Sinir, O. Ozdikis and O. Ozturk, *A Platform for Agent Behavior Design and Multi Agent Orchestration*, Agent-Oriented Software Engineering V, LNCS 3382, Springer, 2005, pp. 205–220.
[9] M. Lotzsch, J. Bach, H. D. Burkhard and M. Jungel, *Designing Agent Behavior with the Extensible Agent Behavior Specification Language XABSL*, RoboCup 2003: Robot Soccer World Cup VII, LNAI 3020, Springer, 2004, pp. 114–124.
[10] E. Platon, M. Mamei, N. Sabouret, S. Honiden and H. V. Parunak, Mechanisms for environments in multi-agent systems: Survey and opportunities, *Autonomous Agents and Multi-Agent Systems* **14**(1) (2007), 31–47.
[11] J. Saunier, F. Balbo and F. Badeig, *Environment as Active Support of Interaction*, Proceedings of the Third International Workshop on Environments for Multiagent Systems, 2006, pp. 61–78.
[12] P. Valckenaers, J. Sauter, C. Sierra and J.A. Rodriguez-Aguilar, Applications and environments for multi-agent systems, *Autonomous Agents and Multi-Agent Systems* **14**(1) (2007), 61–85.
[13] D. Weyns, A. Omicini and J. Odell, Environment as a first class abstraction in multiagent systems, *Autonomous Agents and Multi-Agent Systems* **14**(1) (2007), 5–30.

[14] L. Xiao and D. Greer, The Agent-Rule-Class Framework for Multi-Agent Systems, Special Issue on Agent-Oriented Software Development Methodology, *International Journal of Multiagent and Grid Systems* **2**(4) (2006), 325–351, IOS Press.
[15] L. Xiao and D. Greer, Externalisation and Adaptation of Multi-Agent System Behaviour, in: *Advanced Topics in Database Research*, (Volume 5) Chapter IX in K. Siau, ed., Idea Group, Inc., 2006, pp. 148–169.
[16] L. Xiao and D. Greer, *Agent-oriented Requirements Modelling*, Proceedings of the First International Workshop on Requirements Engineering for Business Need and IT Alignment (REBNITA'05), pp. 28–37, Paris, France, 29–30 August, 2005. In conjunction with the Thirteenth IEEE Requirements Engineering Conference (RE'05).
[17] J. Yoder, F. Balaguer and R. Johnson, Architecture and Design of Adaptive Object-Models, *ACM SIGPLAN Notices* **36**(12) (2001), 50–60.
[18] J. Yoder and R. Johnson, *The Adaptive Object-Model Architectural Style*, Proceedings of the IFIP 17th World Computer Congress – TC2 Stream / 3rd IEEE/IFIP Conference on Software Architecture: System Design, Development and Maintenance, 2002, pp. 3–27.

Authors' Bios

Liang Xiao is a Postdoc Research Fellow at the RCSI HRB Centre for Primary Care Research in Dublin. Before coming to Ireland, Dr. Xiao was a Research Fellow at the University of Southampton. His research activities in Southampton included working on the EU Framework 6 projects HealthAgents and OpenKnowledge. He obtained his BSc at the Huazhong University of Science & Technology in China, his MSc at the University of Edinburgh, and PhD at Queen's University Belfast. He has worked in the telecommunications industry as a Software Engineer following his graduation in China. His experience on solving real domain problems has stimulated his research interests in the area of Software Engineering in Edinburgh and Queen's. Specifically, his research work focuses on Software Adaptivity, Model Driven Architecture, and Agent-oriented Software Engineering.

Des Greer is a lecturer in Computer Science at Queen's University, Belfast. He is a graduate of QUB and earned a masters and doctorate at the University of Ulster. Before his career in academia at Queens and previously at Ulster, he worked in industry as an analyst-programmer and his research is inspired by real problems in software engineering. His particular research interests are in software evolution planning, agent oriented software engineering, and software adaptivity. Dr Greer teaches Software Engineering and Rigorous Software Design at undergraduate and graduate levels.