

Externalisation and Adaptation of Multi-Agent System Behaviour

Liang Xiao and Des Greer

School of Computer Science,

Queen's University Belfast

Belfast, BT7 1NN, UK

email: { l.xiao, des.greer }@qub.ac.uk

phone: +44 (0)28 9097 4656

fax: +44 (0)28 9097 5666

Externalisation and Adaptation of Multi-Agent System Behaviour

Abstract. *This chapter proposes the Adaptive Agent Model (AAM) for agent-oriented system development. In AAM, requirements can be transformed into externalised business rules. These rules represent agent behaviours and collaboration between agents using the rules can be modelled using extended UML diagrams. Specifically, a UML structural model and a behavioural model are employed. XML is used to further specify the rules. The XML-based rules are subsequently translated by the agents. The UML diagrams and XML specification can both be edited at any time, the newly specified behaviours being available to the agent system immediately. An illustrative example is used to show how AAM is deployed, demonstrating adaptation of inter-agent collaboration, intra-agent behaviours and agent ontologies. With AAM there is no need to recode and regenerate the agent system when change occurs. Rather, the system model is easily configured by users and agents will always get up-to-date rules to execute at run-time.*

Keywords: *Adaptivity, Business Rules, Case Tools, E-Business, Modeling Languages, Requirements, Software Agent, Software Architecture, Software Engineering, Structural Modeling, Multi-Agent System, Object-Oriented Design, UML, XML.*

INTRODUCTION

Agent-oriented systems differ from object-oriented systems in that agents are active, while objects are passive. Thus, agents have the goal of having dynamic behaviours. Therefore, agent systems should be easily adaptable, being easily changed by engineers. Better still, would be that they were adaptive, where systems change their behaviours according to their context (Lieberherr, 1995).

Although many tools and techniques are available for agent-oriented systems development, there is no unified and mature way to do it. What is more, existing agent platforms, like JADE (Java Agent DEvelopment) (Bellifemine, Caire, Poggi, and Rimassa, 2003), require designers and developers to code agent behaviours in fixed methods and the way to write them varies from one platform to another. This lack of uniformity of approach means that maintaining agent systems is potentially expensive. Being able to automatically generate agent systems and adapt their behaviours with changing requirements would alleviate this maintenance burden.

The objective of the chapter is to find a way to externalise agent behaviours in a repository. The configuration of the agent behaviours can be made at run-time by changing the repository, supported by tools. Therefore new requirements can be continually reflected in the agent systems. We call this repository a requirements database and our approach Adaptive Agent Model. The requirements database is in XML format and the stored agent behaviours are represented as business rules.

BACKGROUND

In this section, we will first of all introduce agent systems in general briefly, and how such systems are currently developed. We then present some existing approaches towards system

adaptivity. After that business rules, being able to capture system behaviours, are presented as a means to achieve more flexible agent behaviour code. Following the demonstration of possible implementation of rules as agent behaviours, we come back to the design aspect and describe the addition of the rule in two existing extended UML notation systems, their usefulness and insufficiencies. Finally, our perspective and the main idea of our approach are given.

Agent systems and platforms

Software agents are defined as follows: “An agent is an encapsulated computer system that is situated in some environment, and that is capable of flexible, autonomous action in that environment in order to meet its design objectives” (Jennings, 2000, p. 280). Sending and receiving messages are the two main activities of agents. Various agent system development platforms are available, the JADE framework being one of them. JADE is aimed at developing multi-agent systems and applications conforming to FIPA (Foundation for Intelligent Physical Agents) (FIPA, 2005) standards. With JADE, an agent is able to carry out several concurrent tasks in response to different external events. To date, developers have, generally, been required to write repetitive and tedious code for the behaviour of every agent manually.

Existing approaches

Current approaches to agent-oriented system design and implementation are fundamentally based on the identification of agent interaction protocols, message routing, and the precise specification of the ontology. This need for complete upfront design makes it difficult to

manage agent conversations flexibly and to reuse agent behaviour (Griss, Fonseca, Cowan, and Kessler, 2002). Using Agent Patterns (Cossentino, Burrafato, Lombardo, and Sabatucci, 2002) is one way for better code encapsulation and reuse. In support of Agent Patterns, it is argued in (Cossentino *et al.*, 2002) that much research work such as Gaia (Wooldridge, Jennings, and Kinny, 2000), MaSE (DeLoach, Wood, and Sparkman, 2001), and Tropos (Castro, Kolp, and Mylopoulos, 2002) emphasise only the design of basic elements like goals, communications, roles, 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 context is not straightforward. In addition, this approach is not adaptive since 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 (Arai and Stolzenburg, 2002) and the Extensible Agent Behaviour Specification Language (XABSL) has been specified (Lotzsch, Bach, Burkhard, and Jungel, 2004) 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 behaviours, but cannot express behaviours that involve inter-agent collaboration. Moreover, although agent behaviours are modelled in XABSL, they must be compiled before being executed by the agent engine. Thus, changing the XABSL document always requires re-compilation.

Agent behaviours are modelled as workflow processes in (Laleci *et al.*, 2004) and a Behaviour Type Design Tool is described for constructing behaviours. This approach provides a convenient way to compose agent behaviours 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.

Business rules and agent behaviours

A business rule is a compact statement about some aspect of a business. It is a constraint in the sense that a business rule lays down what must or must not be the case (Morgan, 2002). Often, business rules are hard-coded into programs, but keeping business rules distinct from code has many advantages, including the possibility that they can remain highly understandable and accessible to non-programmers. XML-based rules have been used in the IBM San Francisco Framework (Bohrer, 1998) as templates to specify the contents and structures for code that is to be generated. With this approach, changing of XML rule templates allows mappings to new object structures. Figure 1 shows an example, where a generic XML rule has been converted to a specific Java method, `getDiscount ()` in this case.

```
<Rule>
  <Target> Attributes </Target>
  <Condition> scope = public </Condition>
  public &type; get&u.name;() {
    return iv&u.name;;
  }
</Rule>
```

If the name of one of the public attributes for an “Order” class was “discount”, and its type “Double”, then this template would generate:

```
public Double getDiscount() {
  return ivDiscount;
}
```

Figure 1. Example of code generation using rules

Because agent behaviours represent actual system requirements and are subject to change, the application of business rules to the agent world should offer similar advantages as in the object world.

Agent-oriented UML

Agent UML (AUML) by FIPA (FIPA, 2005) extends UML diagrams to cover needs for agent-oriented system design. In the context of agents and multi-agent systems, AUML class diagrams and interaction diagrams introduce new concepts, like agent, role, organization, message, protocol, etc. with their corresponding notations. Interaction Protocols (IPs) between agents are defined to describe various inter-agent activities in a pre-agreed message exchange style. Agents intending to participate in any IP must adhere to the AUML specification. Levelling is used for refinement of the interaction processes.

Agent-Object-Relationship (AOR) (Wagner, 2003) models show social interaction processes in organizational information systems in the form of interaction pattern diagrams. These model agents, ordinary objects, events, actions, claims, commitments, and reaction rules which dictate behaviours. AOR can be viewed as an extension of UML for agent systems and is capable of capturing the semantics of business domains.

Although AOR introduces an additional element of rule over the AUML notation system for modelling agent behaviours, the construction and editing of rules are not in its scope. Moreover, how agents, objects and rules work together are not described adequately. However, it provides an appropriate notation system for the agent world and we later adapt and use it for our conceptual modelling of agents, rules and their interactions.

Our approach

In answer to the weaknesses of the existing modelling patterns and coding approaches, we propose that the agent interaction models, represented in the form of UML, are related with the agent behaviour specification, represented as XML-based business rules. The combination of them is used to transform to the agent Behavioural Model for the agent systems at run-time.

The central component of the approach is rules. They capture customer requirements, participate as a behavioural element in the design models, are specified in XML, and interpreted by agents as behavioural guidelines while the system is running. The transformation of rules turns requirements database into executable systems. While such systems are running, rules have business classes available to act upon. They govern agent behaviours, make decisions for agents in various contexts, have control over the invocation of business classes and are adaptive. Each agent reacts to external events according to the XML-based rules at run-time. Rule definitions are easy to adapt, therefore different business classes can be invoked by agents to achieve dynamic effect.

SOLUTION APPROACH: ADAPTIVE AGENT MODEL

We propose an approach, the Adaptive Agent Model (AAM). In this we emphasise the integration of i) UML diagrams, which model inter-agent relationship, and ii) XML-based rule definitions, each of which describes an individual agent behaviour. UML model information will become part of the XML definitions and enable agents to understand their communication with the outside world. The transformation of a piece of requirement to rule descriptions, then a rule element in UML, after that XML specification, and finally an interpreted agent behaviour is systematically demonstrated using our case study.

Case study

To illustrate our approach and to use in our discussion later, we introduce an ecommerce case study. Suppose a retailer runs an online shop. The retailer has an association with customers and also with various supplier companies, who may or may not serve the retailer, depending on different policies that different companies would take in different sale seasons. If the requested order is profitable to the supplier company, it proposes a deal, including the price and delivery time, etc. for the order. The retailer accepts the proposal if it is satisfied with the deal. Overall, the relationship between customers, the retailer and supplier companies can change at any time. The business vocabulary is also changeable and the decision making process for each company, retailer and customer is unpredictable.

Requirements analysis

Functional requirements can be identified according to the described case study. They are organised according to the actors that use them. Obviously one actor may have multiple requirements related with other actors. These are uniformly documented in tables. Each table contains the information about a function owned by an actor, describing the function name, informative description of the function, the cause of the function, information used by the function, its outputs, required effects, and finally an identifier. Table 1 is such an example.

Table 1. Sample requirement table

Name	<i>saleProcessing</i>
Description	To handle order request from the <i>retailer</i> .
Cause	Receipt of a call for sale proposal from a <i>retailer</i> . The received business information is provided in the form of a combination of <i>retailer information, order identity, and ordered goods details</i> .
Information Used	Retailer information & order information.
Outputs	A sale proposal, to the <i>retailer</i> .

Required Effect	If the order is evaluated as <i>attractive</i> then a new sale proposal is created from the request and sent to the retailer; otherwise the request is rejected or renegotiated.
Identifier	Company. saleProcessing

We will use this specific requirement about the behaviour of the company on processing the request from the retailer for customer order throughout the remainder of the chapter.

Rule Model

In the OO world, for each functional requirement table, a method in a class could be written. However, because the relationship between the communicating parties may change, a fixed method would not be suitable to the described scenario. Moreover, each supplying company may change its sale policy, which means the way of evaluating order request and creating sale proposal varies with company policies and sale seasons. Thus, it is desirable to choose a configurable behavioural element for the executing component.

We use a business rule to represent a configurable behaviour for a runnable agent. One rule makes use of stable business classes and tells an agent how to collaborate with other agents by receiving/sending messages.

The following diagram is a generic Rule Model.

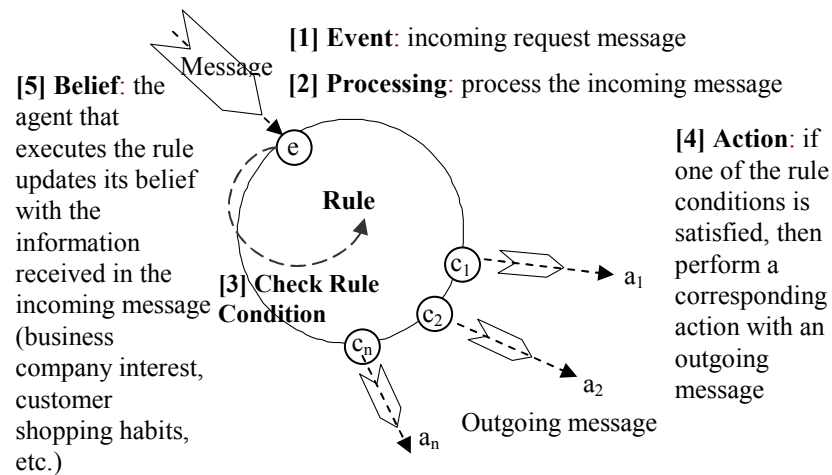


Figure 2. The Rule Model

In such a model, events cause agents to execute rules and if certain conditions are satisfied, some actions are triggered which in turn include generated events for other agents. An agent processes a rule using the following steps, in accordance with what is shown in Figure 2. A rule definition is made up of the steps that an agent takes to execute the rule.

- [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 business objects to be used in later phases.
- [3] Check condition. Find out if the $\{\text{condition } c_i\}$ is satisfied.
- [4] Take an action. If c_i is satisfied, then do the corresponding $\{\text{action } a_i\}$ that is related with $\{\text{condition } i\}$ as defined by the rule. Then, send a result message to another agent (possibly the triggering one). 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} .
- [5] Update beliefs. Using the information obtained from the message just received, the knowledge of the agent of the outside world is updated.

The actors that can be identified in the case study reveal the participating agents, which we call “CompanyAgent”, “RetailerAgent”, and “CustomerAgent”. Two types of classes can be identified, which we name “Order” and “Proposal”. The requirement table of Table 1 states a required behaviour of the “CompanyAgent”, which makes use of the two classes. The transformation of a requirement table to a rule is straightforward.

1. The “Cause” section is used to make the rule “event”;
2. The sections of “Information Used” and “Required Effect” are used to make the rule “processing”;
3. The sections of “Required Effect” and “Outputs” are used to make the rule “condition” and “action”.

A rule does not necessarily have multiple {condition, action} couplets. The requirement of “saleProcessing” in Table 1 turns to a rule with the following specification, concentrating only on the case that the deal will be done.

- | |
|---|
| <ol style="list-style-type: none">1. Event: receive a “Call for proposal” message from “RetailerAgent”;2. Processing: construct a new “Order” (object) from the message, and create a “Proposal” (object) according to the order for later use;3. Condition: check this “Order” (object) of its attractiveness: (Order.isOrderAttractive () == TRUE);4. Action: If the order is attractive, encode the ready to use “Proposal” (object) into a message and send the message to “RetailerAgent”;5. Belief: “RetailerAgent” has placed an order at this moment. |
|---|

Figure 3. Transformed requirements as a rule called “saleProcessing” for the case study

In the requirements transformation process, concepts like “Order”, “Proposal”, and “attractiveness” are expressed explicitly. However, it is the designers that designate classes and methods for these, later on.

By means of the Rule Model the requirement in Table 1 is modelled as a dedicated rule that will be used by an agent for a specific task and, uses business classes to realise that purpose. This is in contrast with the traditional model that a class method or function call has a fixed body and input/output, designed for a particular type of object. In our model, what and how classes are to be invoked can be specified in rules and these are configurable. The mutable requirements on component collaboration can be externalised in rules and reflected in agent knowledge in terms of their collaboration partners, events processing, and the response messages. Different actions can be set in rules as reactions to different conditions, in an order of user preferences/priorities. Business rules, as we specify here, make agents another abstraction over classes, and therefore superior to classes.

Rationality of using Rule Model for adaptive agent system

Agent systems always require interactions among many agents, modelled as message passing, such that the message sender requests a service from the message receiver. The message receiver uses its internal business objects for the computation required to fulfil the request and then, possibly, takes a further action. Different situations will arise and these are modelled as rules that agents should obey. Thus, a rule is responsible for the behaviour of an agent in dealing with a particular situation. Multiple rules can be defined to let the agent collaborate with other agents to achieve different goals.

Such a model, by using the communication natural of agent system and defining rules for the communication, can help to achieve adaptivity. A rule specifies an agent interface and describes the functionality the agent provides. An agent interface is a contract that is made between an incoming event and an outgoing action, both involving an external agent.

An interface specifically dedicated for the description of system interactions can bring adaptivity. Message-Oriented Middleware (MOM) (Mahmoud, 2004) is a middleware infrastructure that offers distributed messaging communication similar to a postal service. MOM has an architectural style well suited to support applications that must react to changes in the environment. It provides an independent layer as an intermediary for the exchange of messages between senders and receivers. This allows source and target systems to link without having to adapt them to each other (Mahmoud, 2004).

Having a more loosely coupled architecture, the AAM not only provides the independence of the interface layer between all participants, but also the functionalities of each of them is collectively centralised in a rule base. Thus, each agent in our system is adaptive externally and internally.

Design Models

Once rules are collectively transformed from the requirements tables, they must be related with the agents that will use them and the classes that will be used by them in design diagrams. For example, Figure 3 indicates that the rule “saleProcessing” will be executed by “CompanyAgent” when the agent receives a call for sale proposal message from the “RetailerAgent”. An “Order” class and a “Proposal” class may be invoked by the rule to assist this operation. Traditional UML models need to be extended to accommodate not only the concept of class, but also agent and rule, more importantly, their relationship. Two main models have been designed, for the design of systems using our AAM approach.

Structural Model: Agent Diagram

Structural Models are built through *Agent Diagrams*, and show agents, business rules, business classes and their relationship. Agents manage rules and rules manage the invocation of business classes. Such models are used for agent identification, agent relationship identification, and eventually building an Agent/Rule/Class hierarchy. They are later the basis for the Behavioural Models.

Agents are identified to represent distinct conceptual domains. Agent Diagram has the Class Diagram, the backbone of UML (Fowler, 2004) as its counterpart in the object-oriented models. In our AAM approach, agents are regarded as superior to classes. Each rounded cornered box represents an agent and is divided into three compartments. The top compartment holds the name of the agent, the middle compartment holds the classes managed by the agent along with their instantiation and the bottom compartment holds the rules that govern the functions of the agent. This construct resembles a class name, an attribute list, and an operation list constituting a class diagram in the OO world.

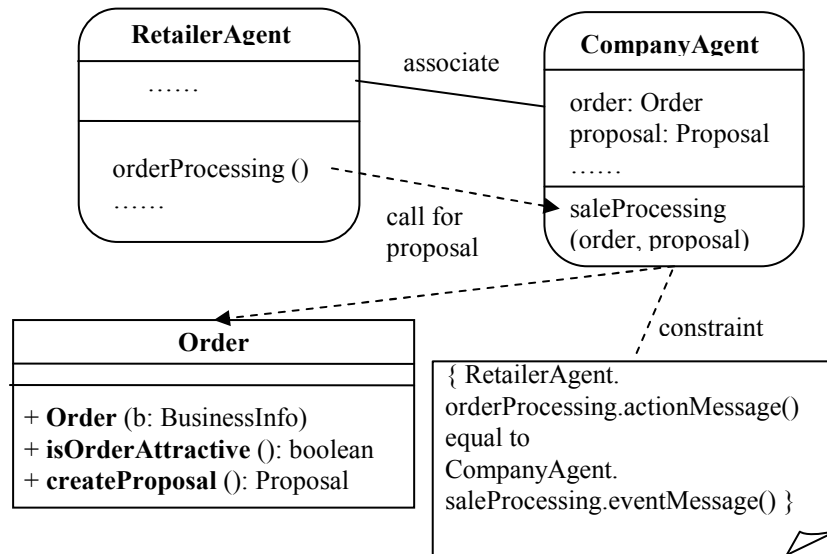


Figure 4. The Agent Diagram for the case study

In Figure 4, two identified agents “RetailerAgent” and “CompanyAgent” for our case study are shown. “RetailerAgent” has a rule “orderProcessing” that will construct an object with type “BusinessInfo”, package it into a “Call for proposal” message and send the resulting message to “CompanyAgent”. To respond to such requests, “CompanyAgent” will offer a deal, if the order is attractive, using the rule “saleProcessing”. Thus, we have an association relationship between the two agents involved and a constraint for them. They resemble an association between two classes and a constraint for classes in the OO world. During the processing of rule “saleProcessing”, an “Order” object will be constructed from the received “BusinessInfo” structure and the constructed object should pass an “isOrderAttractive” check before “CompanyAgent” proceeds to offer a deal, “Proposal” for the order. Thus, such a business class of “Order” is related with “CompanyAgent” via “saleProcessing” and it has at least three methods that will be invoked by the agent rule.

The diagram of Figure 4 structurally documents the system model, highlighting “saleProcessing” rule, which makes use of “Order” and “Proposal” classes and will be used by

“CompanyAgent”. This rule-centred diagram is constructed from the requirements rule in Figure 3 using the following steps.

1. Find out in the descriptions of “Event” and “Action”, the message passing pattern between participating agents, and identify the agent which the rule belongs to. Draw the agent boxes and passing messages in the diagram;
2. Analyse “Processing” and extract all the business classes that are used by the rule. Relate the classes with the agent/rule and update the middle and bottom compartment of the agent box. Draw the class boxes and connect them with the rule in the bottom compartment of the agent box;
3. Consider the possible methods that the recognised classes may have by examining “Processing” and “Condition”. From these respectively, at least a constructor method and a method with return type Boolean should be identified. Add class methods to the class boxes in the diagram.

Behavioural Model: Agent Communication Diagram

Agent Diagrams capture the static relationship between different entities and depict the whole system. *Agent Communication Diagrams* are used to model the interaction of agents. Such Behavioural Models organise agents, rules and messages around business processes. For every business process, all participating agents will appear in the diagram, with message passing between them to accomplish certain business goals.

Software Architecture refers to the communication structures for system entities. In traditional object-oriented systems, objects are aware of which other objects they will pass messages to, but are unaware of which objects will pass messages to them. Full architecture independence requires that the detail of where objects will send messages should also be

hidden (Hogg, 2003). In agent-oriented systems, business processes are implemented by the collaboration of agents. The management of this collaboration requires the agent architecture to be well modelled. In order to generate agent systems and be able to adapt them afterwards without re-generation and re-compilation, full architecture independence (two-way encapsulation) is required, and the interaction information should not be hard-coded so that agents can adapt their collaboration in communication according to changing requirements.

In our approach, an extended UML diagram, as shown in Figure 5 is used to model agent collaboration, describing how message passing among coordinated agents can accomplish business tasks. These diagrams provide a blueprint for involved business rules, the composition elements of our diagrams. Each rule governs an individual agent behaviour in the participating collaboration. Rules are connected to form a flow of decision making, process by process, one decision being made at each connection point. As such, the model visualises the actual system function in a sequence of agent actions dictated by rules. User specified agent collaboration in the diagrams is used to generate the inter-agent part of the rules definition, in XML format. It is through these rules that agent systems are adapted both in collaboration and internally without re-code or re-generation, since we let agents get appropriate rules to execute only at run-time, and rules get configured continuously through supporting tools we provide.

Accurately, the diagram used for the design of multi-agent behaviours is the Agent Communication Diagram. It has been developed based on the Agent Diagram and used for the generation of agent systems. Figure 5 describes the process for the case study, where a customer orders products from a business company through a retailer. Business classes are not shown on the diagram but the invocations of their methods are, such as, the one for condition check. R2 has been shown previously as “saleProcessing” in the bottom compartment of “CompanyAgent” in Figure 4.

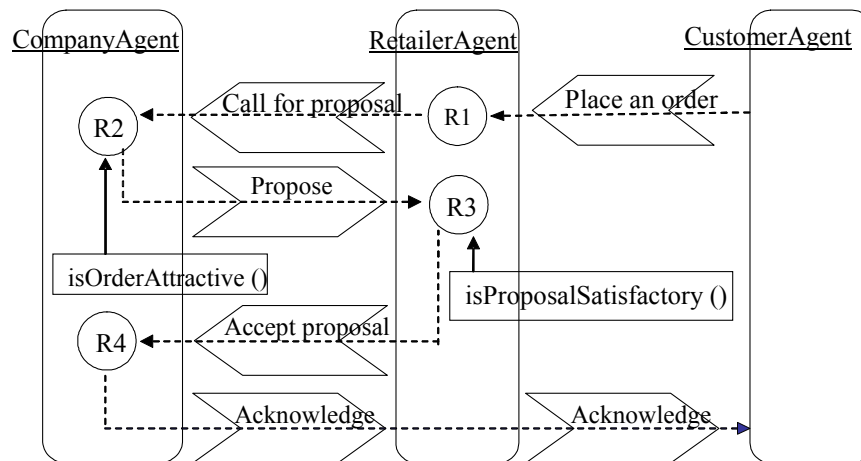


Figure 5. An Agent Communication Diagram describing a business process

A similar transformation can construct the Behavioural Model from requirements rules, just as has been done to make the Structural Model. Alternatively, the model can be built based on the previous one.

1. Draw all the agents and, within them, the appropriate rules;
2. Draw all the passing messages between associated agents in their rules. Wherever a message goes from one rule to another, the “Action” of the former rule and the “Event” of the later one describes the same message and they match in the recipient and sender;
3. Draw only the {condition, action} couplets that eventually contribute to the successful result.

A structural diagram concentrates on one rule, while a behavioural diagram ignores structural and low level details and puts agent actions logically in a sequence for a complete business process. Only the main route, which directs to the successful result, is shown in the diagram. The route has many divisions, each of which has a message passing between two rules in two agents. Figure 5 describes our case study with all participating agents and their rules collectively connected. This conforms and extends the Rule Model for multiple

agent/rule collaboration. The model has a different view from the previous model of the same system.

Rule implementation

The traditional software system development process can be viewed as a series of transformations through the form of requirements documents, design models and implemented code. The performance of the final product precisely reflects the desired behaviour in the required system. The initially captured knowledge, usually documented in UML diagrams, is essential to the system implementation. However, these models rapidly lose their value as, in practice changes are often done at the code level only.

In our approach, rules serve as a requirements database. Then after being transformed into a UML element, they represent agent behaviours. After that they are specified more accurately in XML. Finally they are interpreted by the running agent software.

UML-style diagrams are good at showing collaboration among agents, while XML specification is good at precise definition of agent behaviours, an aspect that UML diagrams lack (Fowler, 2004). The use of rules allows designers to use the combination of UML and XML, one complementing another, where the former models the system blueprint and the latter models the behavioural details. Because the UML and XML models are combined for interpretation as agent behaviours at run-time, the design and implementation of the system are seamlessly integrated. Changes are done and can only be done at the model level. This is less error prone and safer than direct change of code.

According to the Rule Model, we encode the diagrammatic rule in the models in XML, with the structure {event, processing, condition, action, priority}. The computer-readable Java style code specifies on receipt of an event, how an agent should act if the condition of the rule is

satisfied. Rules are considered for execution by agents according to priorities set by users. The XML representation for rule R2 is given in Figure 6.

The construction of rule in XML from the models takes the following steps.

1. Each rule definition has a root element of `<business-rule>`;
2. An element of `<name>` and `<owner-agent>` defines the name of the rule and the agent that owns it respectively. These have been given in the Structural Model already;
3. An element of `<global-variable>` defines all the business classes with their instantiation that will be used by the rule. These classes can be found in the Structural Model, where they are related with the rule;
4. An element of `<event>` and another element of `<action>` define the message flow, and the structure of the messages. An element of `<from>` and another element of `<to>` in the `<message>` structure defines where the incoming message comes from and where the outgoing message goes to. These can be found in the Behavioural Model, where each rule is connected with two messages, one in and one out. An element of `<content>` in the `<message>` structure defines the encoded objects in each message. They are not given in the models but can be found in the requirements rules. The objects defined in the `<global-variable>` are usually involved;
5. An element of `<processing>` and another element of `<condition>` define the construction of business objects from the event message, and invocation of methods on them. All the methods can be found in the Structural Model, where classes are related with the rule.

The evaluation method for the `<condition>` can be found in the Behavioural Model.

```
- <business-rule>
  <name>saleProcessing</name>
  <business-process>retailer business</business-process>
  <owner-agent>CompanyAgent</owner-agent>
  - <global-variable>
    - <var>
      <name>order</name>
```

```

        <type>Order</type>
    </var>
    - <var>
        <name>proposal</name>
        <type>Proposal</type>
    </var>
</global-variable>
- <event>
    <type>receipt of message</type>
    - <message>
        <from>RetailerAgent.orderProcessing</from>
        <to>CompanyAgent.saleProcessing</to>
        <title>Call for proposal</title>
        - <content>
            - <businessInfo>
                - <retailer> ... </retailer>
                - <order>
                    <id>10010001</id>
                    - <product>
                        <classification>book</classification>
                    ...
                </product>
            </order>
        </businessInfo>
    </content>
</message>
</event>
<processing>
    order = new Order (businessInfo)
    proposal = order.createProposal ()
</processing>
<condition>
    order.isOrderAttractive() == true
</condition>
- <action>
    <type>send a message</type>
    - <message>
        <from>CompanyAgent.saleProcessing</from>
        <to>RetailerAgent.proposalProcessing</to>
        <title>Propose</title>
        - <content>
            - <proposal>
                <id>10011101</id>
                <businessInfo> ... </businessInfo>
            </proposal>
        </content>
    </message>
</action>
<priority>5</priority>
</business-rule>

```

Figure 6. The XML definition for rule “saleProcessing” owned by “CompanyAgent”

In the diagram of Figure 5, “CompanyAgent” reacts to the “Call for proposal” message from “RetailerAgent” by executing the above specifically defined rule “saleProcessing” in XML, shown in Figure 6. In general, each agent executes a rule in the following way.

1. Get a list of its managed rules from a rules document according to the <owner-agent> section.

2. Filter these rules and retain those which are applicable to the current business process according to the <business-process> section.
3. Get the rule currently has the highest priority according to the <priority> section.
4. Check the applicability of this selected rule. That is, check if the <event> section matches the event that has occurred. In other words, check if the agent that triggers the received message is the same as that given in the <from> section of the <message> in <event>, and the received message format is also as specified in the <message>. If that is not the case, go to Step 9.
5. Decode the message received and build business objects from it following the <processing> instructions. Constructor methods of existing classes will be involved. Global variables declared in the <global-variable> section will be used to save the results.
6. Check if the current condition specified in the rule is satisfied according to the <condition> section. Constructed business objects will be involved, and their methods will be invoked upon to assist the rule to function. If the condition is not satisfied and it is not the last condition, move to the next condition and repeat Step 6, otherwise go to Step 9.
7. Execute the corresponding <action> section. This involves encoding constructed business objects that refer to <global-variable> into a message. Send the message to the agent which is specified in the <to> section of the <message> in <action>.
8. Analyse the business objects which have been decoded from the message received and update the agent's beliefs with the new information available.
9. Remove this selected rule from the rules set obtained in Step 2 and if not the last rule, go to Step 3.

10. Wait for the next event.

Supporting tool and agent system implementation

A CASE tool has been developed to enable the specification of the agent collaboration, rule definitions and message flows. Figure 7 captures a window from this tool showing the construction of an Agent Communication Diagram in its main panel. Rules can be defined either in XML text or using a more user-friendly tree structure as shown in the left panel. The tree is structured using the same schema that constructs the document in Figure 6. Business classes can be registered using the tool and after that selected for the specification of messages passing between agents. For example, Figure 7 shows that “businessInfo” has been chosen as the content of the event message and “proposal” as the content of the action message for the “saleProcessing” rule, conforming to the specification in Figure 6. The section of <from> and <to> for event and action messages can be generated when the direction of the messages are set up visually in the main panel of the tool. Existing class methods can be selected for <processing> and <condition>, and a number for <priority>. XML code is eventually generated from the completed tree structure and saved in a rules document.

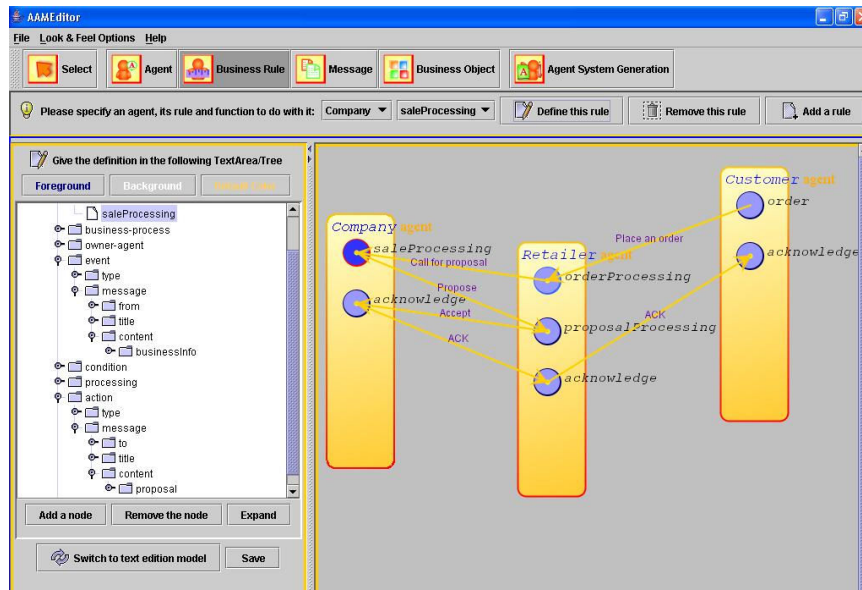


Figure 7. AAM supporting tool

Our supporting tool uses a business rules document as the database. Once business processes are specified graphically in the tool, agent interaction models, rule reaction patterns and message flows are established accordingly. The agent system framework is automatically generated such that each rule maps to an agent behaviour. Program code is not generated at this moment. Instead, XML-based rules are plugged in and are subsequently translated by agents at run-time. Figure 8 shows the pseudo code that “CompanyAgent” will interpret from the “saleProcessing” rule to execute as one of its behaviours.

The system runs on the JADE platform and can be in a distributed network. All agents access the central XML-based rules document via a parsing package. By using this package, agents can do the comparison to check the applicability of rules (① in Figure 8) and run pre-defined statements embedded in the XML tags of the rules (②,③,④ in Figure 8). These are interpreted from the XML specification in Figure 6. While the system is running, the rule specification can be continually changed through the tool. This allows dynamic adjustment of agent communication structure and therefore the software architecture of the system.

A shared module in the XML parsing package called “Rule”, being able to access the XML definition of rules and assemble corresponding objects, is used by all agents. The methods `getPriority()`, `getEvent()`, and `getAction()` are provided by “Rule”.

```

thisAgent. addBehaviour (Rule thisRule) {
    thisBehaviour. setPriority (thisRule. getPriority ());
    Order order;
    Proposal proposal;
    Message m = thisAgent. receiveMessage ();
    while (m != null)
    {
        Agent fromAgent = m. getSenderAgent ();
        if (fromAgent. equals
            (thisRule. getEvent (). getMessage (). getFromAgent ())) // ❶
        {
            /* the rule is applicable to the received message */
            BusinessInfo businessInfo = (BusinessInfo) m. getContentObject ();
            order = new Order (businessInfo); // ❷
            if (order. isOrderAttractive ()) // ❸
            {
                /* the condition of the rule is satisfied */
                proposal = order. createProposal (); // ❹
                Message m2 = new Message ();
                m2. setContentObject (proposal);
                Agent toAgent =
                    thisRule. getAction (). getMessage (). getToAgent ();
                m2. addReceiverAgent (toAgent);
                thisAgent. send (m2);
                /* update this agent's beliefs */
                thisAgent. addBelief (System. getcurrentTime (), fromAgent, m);
            }
        }
        m = thisAgent. receiveMessage ();
    }
}

```

Figure 8. Pseudo code for behaviour of “CompanyAgent”, mapping to its “saleProcessing” rule

Deployment

The deployment of an implemented system using the proposed approach is shown in Figure 9. The actual agent system (❸ in Figure 9), running on the JADE platform in a distributed network, is initially generated from the supporting tool (❶ in Figure 9). A central XML-based

business rule repository (❷ in Figure 9) is deployed in the network, containing the rule definitions and the registered business classes that are used by the rules. The XML parsing package is implemented as a JavaBeans component, responsible for parsing the XML format of business rules and presenting the parsed business knowledge in the tool. The tool is continuously used by business people to maintain requirements (❶ → ❷). The edition through the tool for the requirements change is saved in the XML repository using the same JavaBeans.

All agents access the repository via the JavaBeans as well, in order to obtain the most up-to-date knowledge in an easy to operate format. In the beginning, each agent has the knowledge of whom and how they will collaborate with, dictated by the initial rules. While the system is running, the business requirements model can be continuously under maintenance through the tool. With the assistance of the JavaBeans, each agent in the generated agent system interprets the updated requirements knowledge for action/reaction (❷ → ❸). Eventually agents can always get the desired behaviours as soon as they have been specified through the tool, and can be continuously updated.

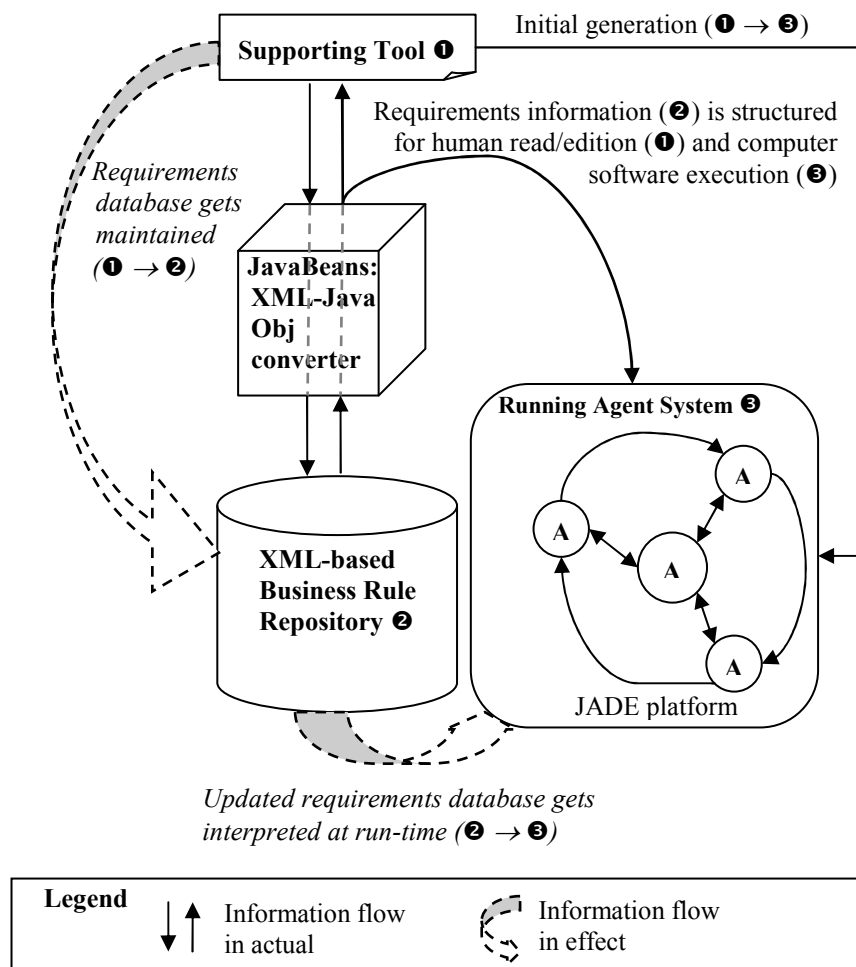


Figure 9. Deployment of the system

Adaptation

Modelled as business rules, the requirements database in our system can be adapted in three aspects for three purposes, from the perspective of the running agents. They are respectively: the collaboration between agents; the internal behaviours of each individual agent; and the classes that agents can make use of.

Adapting inter-agent collaboration

Being able to adapt the collaboration between agents at run-time, AAM achieves two-way encapsulation. Agent behaviours are guided by rules so that they do not need to know who they will contact in advance. To reflect business process change, the Behavioural Models can easily be changed visually with the tool. These changes are automatically reflected in the XML definitions of corresponding agent rules, for example, in their `<event>/<message>/<from>` and `<action>/<message>/<to>` sections. This enables agents in the running system to have their partners changed in order to accomplish the updated business processes. On receipt of any message, an agent reads the most recent rules, analyses them and finds out the appropriate agents to send messages to. In the case study, we may wish to re-configure the rule “saleProcessing”, and let the “CompanyAgent” take a new action in a condition originally not predicted.

Suppose we wish to introduce a new occasion where if the current “CompanyAgent” does not evaluate the received order request to be “attractive” or can not fulfil the order request, it forwards the order to another “CompanyAgent”. This new requirement can be specified, implemented and deployed by agents automatically by configuring the Agent Communication Diagrams using the tool. The achievement of this dynamic collaboration is through painless model adjustment rather than expensive code change. Further, we achieve the model-driven communication architecture.

Adapting intra-agent behaviours

The behaviours of agents in processing the event, checking the condition, and taking the action are externalised in business rules. This means that they can be configured dynamically.

In fact, by changing the <event>, <processing>, <condition>, and <action> fields in appropriate rules, alternative methods of the managed business objects can be selected for invocation. In the case study, we can re-configure the rule “saleProcessing” to invoke a new evaluation method of the “Order” class or even a method of a new “Order” class to check the attractiveness of the order. In addition, we can configure two couplets of <condition> and <action>, so that for ordinary customers and company customers, different means to generate sale proposals can be used. All this can be carried out at run-time.

Adapting ontologies

Only business concepts registered through the tool and saved in the rules document may appear in agent messages. When a new business concept is required, it can be registered with its properties, and a new business class with attributes will be generated by the tool. New vocabularies thus can become available for the specification of agent rules through the tree structure on the left panel of the tool (Figure 7). Also at run-time new classes with new methods thus can become available for invocation by the running agent system. Eventually, all agents will be able to understand the new vocabularies the other agents in the system are using even those registered after the system has been running for a while. Hence, ontologies are always updatable. For the case study, suppose that an additional attribute of the “BusinessInfo” business class is required and added while the system is running, the updated class becomes available to all agents and they start to use the new concept immediately.

EVALUATION: THE MODIFIABLE ARCHITECTURE OF AAM

AAM achieves the quality of modifiability (Bass, Clements, and Kazman, 2003) in its architecture in terms of its prevention of ripple effects and deferment of binding time.

Prevention of ripple effects

A ripple effect from a modification is the necessity of making changes to modules not directly affected by it (Bass *et al.*, 2003). The introduced Agent/Rule/Class hierarchy, having a higher level abstraction of agents over classes and a rule interface between agents helps prevent ripple effects, so reducing the time and cost to implement changes.

Semantically, agents are considered more meaningful communication entities than objects. They are actors that have the rule interface publicly and use a combination of multiple concrete objects privately. This conforms to the idea of information hiding, where changes are isolated within one module, usually a private one, and changes propagating to others, usually public ones are prevented. Rules specified for agents serve as the descriptions of agent responsibilities. They separate the interactions between agents and the use of objects by agents. The change of one agent in its use of objects is kept private and has no influence on the agent that uses the result, as long as the interaction pattern of the two agents is unchanged. For example, no matter what has been changed in the <processing> or <condition> section of the “saleProcessing” rule in Figure 6, the “RetailerAgent” would not be affected in its action, although the “CompanyAgent” starts to use a different means to generate proposal or evaluate order attractiveness. Nevertheless, the “RetailerAgent” expects a proposal as a result from the “CompanyAgent”, as it usually does. In rules, the <processing> and <condition> sections are private to agents and <event> and <action> sections are the public interface.

The supporting tool for AAM always validates and ensures that the action message of A is syntactically equal to the event message of B, if A sends a message to B. This prevents the changes required by syntactic dependencies from propagating: for B to compile/execute correctly, the type of the data that is produced by A and consumed by B must be consistent with the type of the data assumed by B (Bass *et al.*, 2003). For example, if the structure of the “proposal” in the action message of the “CompanyAgent” is required to change, the “RetailerAgent” would expect the new structure automatically when the change has been made (the “Adapting ontologies” section described the details of this).

Deferment of binding time

The mechanism of AAM lets agents interpret changeable rules at run-time and so the binding of the actual software agents and their function specification is deferred until then. This helps to control the time and cost to test and deploy changes.

When a modification is made by the developer, there is usually a testing and distribution process that determines the time lag between the making of the change and the availability of that change to the end user. Binding at run-time means that the system has been prepared for that binding and all of the testing and distribution steps have been completed. Deferring binding time also supports allowing the end user or system administrator to make settings or provide input that affects behaviour (Bass *et al.*, 2003).

In AAM, rules are constructed with the supporting tool and ensured of their validity. No matter how they are changed, they are free of testing for their syntax. Changing rules does not cause any necessary change to the deployment of agents. In addition, the tool is simple enough for use by non-developers to make changes that will be reflected at run-time. The achieved

benefit from the deferred binding time is at the cost of additional interpretation time while the system is running.

FUTURE WORK AND CONCLUSION

Agent behaviours reflect functional requirements. These behaviours are modelled and externalised as rules in the Adaptive Agent Model. The rules are, in effect, executable requirements. In the design models they are present in extended UML diagrams. In the implementation models they are centrally managed and easily changed through their XML-based definitions. Because rules are easy to edit, and agents always get the most recent rules for interpretation, deploying new requirements requires minimal effort.

The XML specification of the rules, related with the corresponding UML elements, makes our models which combine UML with XML reusable. The models are continuously reused, not only for the regular revision by users, but also for constant interpretation by software agents. The maintenance of the AAM models is, in fact, equivalent to the maintenance of the final software system.

One weakness of AAM is that the framework's externalisation of agent behaviours in XML-based rules will degrade the performance of such systems. Every time an agent acts and reacts to events, it will read the rules document, test rules' applicability, find the one with the highest priority, and execute it. Therefore, there is a trade-off between ease of adaptation and performance. Resolution of this issue remains an aspect of future work.

Ultimately, we expect to achieve self-adaptivity in the AAM where, as agents interact with end users they perceive their behaviours and preferences. As shown in Figure 10, this allows agents to update their beliefs, and so deduce rules that can be added to the central rules

document. These inferred rules can be shared and executed by all agents and are subject to amendment. After some time, a mature and reliable rule set, independent of those acquired through the tool can be established.

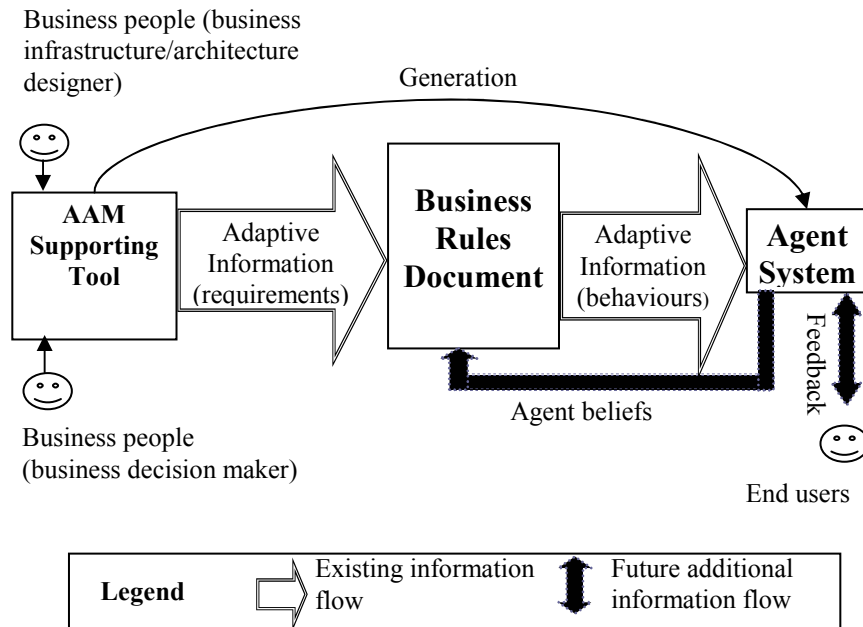


Figure 10. Future Adaptive Agent Model

Further, we plan to develop the Reflective and Adaptive Agent Model (RAAM), the logical follower of AAM. Usually, a reflective system will have a number of interceptors and system monitors that can be used to examine the state of a system, reporting system information such as its performance, workload, or current resource usage (Mahmoud, 2004). RAAM will build on AAM and provide an improved service for its user needs by supporting advanced adaptation features. It allows for the automated self-examination of system capabilities and adjusts and optimises those capabilities automatically. The proposed new feature of auto-adaptivity in RAAM is a natural add-on to the current approach. This characteristic is in contrast with the adaptivity already achieved in AAM, where the system adjusts itself automatically whenever there is a non-functional need, rather than changes according to

functional requirements only. Using dedicated agents to examine and react to the undesirable results from the execution of ordinary rules by ordinary agents is a straightforward means to realise auto-adaptivity and hence build quality into the system. These agents would specifically explore inappropriate decisions made by human beings, negative impact to the overall performance caused by carrying out certain rules, insecure operations caused by certain agents, and respond by suggesting amendment and enhancement. For example, new agents may be created and assigned tasks when degrading system performance is detected or original agents fail. Higher level rules might be specified for these dedicated agents for their examination and reaction.

AAM would be useful for those domains that have frequently changing requirements where re-development would otherwise be costly. Particularly, AAM should work well when there is collaboration between many different entities and where this collaboration may be subject to adjustment, as a result of changing business processes. AAM is also suitable where the business environment is frequently changing with emerging concepts and behaviours.

Other future work will include the development of richer business rules. The Adaptive Agent Model will be made more powerful and more flexible, but work so far indicates that it is highly relevant and useful to the development and evolution needs of multi-agent systems.

REFERENCES

1. Lieberherr, K. (1995). Workshop on Adaptable and Adaptive Software. *Proceedings of the Tenth Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA)*, Austin, Texas, USA, October 15-19, 1995, 149-154. New York, NY, USA: ACM Press.

2. Bellifemine, F., Caire, G., Poggi, A. & Rimassa, G. (2003, September). JADE - A White Paper [Electronic version]. Retrieved July 26, 2005, from <http://jade.tilab.com/papers/WhitePaperJADEXP.pdf>
3. Jennings, N. R. (2000). On Agent-Based Software Engineering. *Artificial Intelligence*, 117(2), 277–296.
4. Foundation for Intelligent Physical Agents (FIPA) (2005). FIPA specifications. Retrieved July 26, 2005, from <http://www.fipa.org/specifications/>
5. Griss, M., Fonseca, S., Cowan, D. & Kessler, R. (2002). SmartAgent: Extending the JADE Agent Behavior Model. Technical Report HPL-2002-18, School of University of Utah.
6. Cossentino, M., Burrafato, P., Lombardo, S. & Sabatucci, L. (2003). Introducing Pattern Reuse in the Design of Multi-Agent Systems. *Agent Technologies, Infrastructures, Tools, and Applications for E-Services*, Kowalczyk, R., Muller, J., Tianfield, H., Unland, R. (eds.), LNAI 2592, 107-120. Berlin: Springer-Verlag.
7. Wooldridge, M., Jennings, N. R. & Kinny, D. (2000). The Gaia Methodology for Agent-Oriented Analysis and Design. *Journal of Autonomous Agents and Multi-Agent Systems*, 3(3), 285-312.
8. DeLoach, S. A., Wood, M. F. & Sparkman, C. H. (2001). Multiagent Systems Engineering. *International Journal on Software Engineering and Knowledge Engineering*, 11(3), 231-258.
9. Castro, J., Kolp, M. & Mylopoulos, J. (2002). Towards Requirements-Driven Information Systems Engineering: The Tropos Project. *Information Systems*, 27(6), 365-389, Elsevier, Amsterdam.

10. Arai, T. & Stolzenburg, F. (2002). Multiagent systems specification by UML statecharts aiming at intelligent manufacturing. *Proceedings of the first international joint conference on Autonomous agents and multiagent systems: part 1*, Bologna, Italy, July 15-19, 2002, 11-18. New York, NY, USA: ACM Press.
11. Lotzsch, M., Bach, J., Burkhard, H.-D. & Jungel, M. (2004). Designing Agent Behavior with the Extensible Agent Behavior Specification Language XABSL. *RoboCup 2003: Robot Soccer World Cup VII, volume 3020 of Lecture Notes in Artificial Intelligence*, 114-124. Springer.
12. Laleci, G. B., Kabak, Y., Dogac, A., Cingil, I., Kirbas, S., Yildiz, A., Sinir, S., Ozdakis, O. & Ozturk, O. (2004). A Platform for Agent Behavior Design and Multi Agent Orchestration. *Agent-Oriented Software Engineering V: 5th International Workshop, AOSE 2004, volume 3382 of Lecture Notes in Computer Science*, 205-220. Springer.
13. Morgan, T. (2002). *Business Rules and Information Systems*. Addison-Wesley.
14. Bohrer, K. A. (1998). Architecture of the San Francisco Frameworks. *IBM Systems Journal*, 37(2), 156-169.
15. Wagner, G. (2003). The Agent-Object-Relationship Metamodel: Towards a Unified View of State and Behavior. *Information Systems*, 28(5), 475-504.
16. Mahmoud, Q. H. (Eds.). (2004). *Middleware for communications*. Chichester, England: John Wiley & Sons.
17. Fowler, M. (2004). *UML Distilled* (3rd ed.). Boston, Massachusetts, Addison-Wesley.
18. Hogg, J. (2003, October). Applying UML 2 to Model-Driven Architecture [Electronic version]. Retrieved July 26, 2005, from http://www.omg.org/news/meetings/workshops/MDA_2003-2_Manual/5-1_Hogg.pdf

19. Bass, L., Clements, P. & Kazman, R. (2003). *Software Architecture in Practice* (2nd ed.). Boston, Massachusetts, Addison-Wesley.