

The Agent-Rule-Class framework for Multi-Agent Systems

Liang Xiao and Des Greer

School of Computer Science, Queen's University Belfast, Belfast, BT7 1NN, UK

Tel.: +44 (0)28 9097 4656; Fax: +44 (0)28 9097 5666; E-mail: {l.xiao, des.greer}@qub.ac.uk

Received 15 November 2005

Revised 13 April 2006

Accepted 5 July 2006

Abstract. Multi-Agent Systems (MAS) have become increasingly mature, but this maturity does not make the traditional Object Oriented (OO) approaches obsolete. On the contrary, building MAS in combination with OO constructs allows the reuse of existing components. Similarly, OO methodologies can benefit from extension towards an agent abstraction and so make use of the methods and tools for MAS. The Agent-Rule-Class (ARC) framework is proposed as an approach that builds agents upon traditional OO system components and makes use of business rules to dictate agent behaviours, aided by the OO components. By modelling agent knowledge in business rules, the proposed paradigm provides a straightforward means to develop agent-oriented systems based on the existing object-oriented systems and offers features that are otherwise difficult to achieve in OO systems. A Structural Model and a Behavioural Model are the central components in the ARC framework for agent-oriented system modelling. A supporting tool has been developed to ensure that agents implement up-to-date requirements from business people, reflecting desired current behaviours, without the need for frequent system rebuilds. The ARC framework provides a complete MAS development process supported by a new process model more suited to collaboration between OO developers, Agent Oriented (AO) developers and domain experts. The main outcome of using ARC is better adaptivity. The ARC framework is illustrated with a rail track example.

Keywords: Adaptivity, agent-oriented software engineering, business rule, design model, object-oriented software engineering, requirements modelling, UML

1. Introduction

Traditionally, human knowledge is transferred into software systems in the form of requirements documents, design models and eventually implemented code, the performance of which should precisely reflect the desired behaviours in the required system. The initially captured knowledge is typically documented in UML models. However, the UML artefacts rapidly lose their value as, in practice, changes are often done at the code level only.

The idea of emphasising system knowledge, and making them reusable models that can be converted to executable software, is promoted in Model Driven Architecture (MDA) [11,19]. In MDA, models are cen-

tral rather than an overhead in the development process. Adopting the objectives of MDA, we propose an agent-oriented paradigm, where knowledge on agent behaviours is structured as business rules. The amendment of these is carried out directly by business people, reflecting desired business requirements. The execution is performed by agents, so reflecting deployed requirements. Such an approach offers several advantages. Firstly, traditional OO methodology is the basis of the framework so that no radical development model change is involved. Secondly, knowledge of the system can be more easily maintained, as business rules are designed in text-format and can be controlled at run-time. Thirdly and lastly, such systems, even based on OO components, can enjoy the added benefits from the agent side.

One of these potential benefits is software adaptivity. Software evolution and maintenance is expensive and accounts for the majority of software lifecycle costs [23]. To make systems more adaptable, effort has been concentrated on making them easily changed by engineers. Better still, would be the ability to make systems adaptive, where systems change their behaviours according to their context [21].

The remainder of the paper is structured as follows. We begin, in the following section, by discussing the need for a new Agent Oriented Software Engineering (AOSE) paradigm of practical use for software adaptivity. Section 3 illustrates the ARC framework with a rail track management system case study, proposing the building of MAS upon existing OO components, and modelling agent knowledge in business rules. Implementation and deployment of the ARC is discussed in Section 4. The main contributions of the ARC framework are described in Section 5. It is demonstrated adaptivity, the main outcome of using the ARC, is achieved, as well as a new efficient and parallel development process that the ARC provides for MAS. We will relate the ARC framework to existing work in Section 6, while Section 7 provides some conclusions for the work.

2. Literature review

This section will review current approaches to software adaptivity including methods arising from OO methodologies and from agent-oriented methodologies, as well as considering existing agent modelling languages.

2.1. OO methodologies towards software adaptivity

Recent approaches aimed at software adaptivity include the strategy design pattern [12], the Coordination Contract [1], and the Adaptive Object Model (AOM) [40,41]. One fundamental property of all these approaches is that they are OO-based. Since objects are passive and traditionally have fixed methods, to make them adaptive, it is either inconvenient or impossible. For example, if using the strategy design pattern, future behaviours must be fully predictable. In the case of using the Coordination Contract approach, only inter-component collaboration can be adapted, so that it is insufficient. The AOM approach relies on meta-models and leads to an architecture which is hard to understand and maintain.

The object-oriented paradigm facilitates design by use of such principles as modularity and information hiding, but this implies ease of redesign rather than adaptation during operation. Considering the common inherent defects of all of these sample approaches and others, one of the primary motivations of this work is to bring about easy software maintenance through using agents. Unlike standard objects, agents are active. Instead of using static methods which are to be invoked and have the same effects all the time, agents are granted the flexibility to choose how to react. Nevertheless, what it is proposed is that investments in OO methodology need not be lost. Rather, existing components can be used to facilitate the operation of agents. Agents, consequently, become a higher level of abstraction over objects.

Intelligent/autonomous agents have been proved to be useful for bringing dynamics, flexibility and adaptivity to travel planning [39], coordinated product development and manufacture [17] and manufacturing systems control [13]. There has been previous work [20, 31] on modelling business rules in agent systems. However, these are not easy to implement and their rules are not configurable at runtime by business people.

2.2. Agent-oriented software engineering (AOSE) methodologies

A major challenge in AOSE research is to turn agent-oriented software abstraction into practical tools [43]. Current AOSE methodologies usually focus on agent-oriented system analysis and design, from the identification of agent interaction protocols to message routing and communication. The need for complete upfront design and implementation makes it difficult to manage agent conversations flexibly and to reuse agent behaviours [14]. Although emerging research work such as Gaia [32,42], MaSE [9], Tropos [6] as well as many others [5] provide the means for the specification of agent-oriented software systems, a complete development process is rarely discussed. Similarly, tools supporting the maintenance of deployed MAS are given limited attention. Developers who are familiar with OO systems have not been systematically assisted by a consistent and complete approach which transforms the very beginning requirements specification to a final executable MAS, using their existing engineering knowledge and skills. Further, since new concepts (goal, role, organisation, etc.) are used in the construction of agent-oriented models, the original object component is often ignored. For example, the BDI (Belief, De-

sire, and Intention) agent paradigm [18] uses an agent model and an interaction model from an external viewpoint, and a belief model, a goal model, and a plan model from an internal viewpoint. Therefore, later in the implementation of MAS, existing components cannot contribute to the behaviour of agents, even though their encapsulated functionalities could be reused to support agents' behaviour. Thus one can not easily take full advantage of the agent abstraction over object or adapt the behaviours of MAS through re-configuration of supporting object components.

We have evaluated some AOSE approaches in the direction of using reusable modules such as OO components. Using Agent Patterns [7] is one way for better code encapsulation and reuse. In support of Agent Patterns, it is argued in [7] 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 [2] and the Extensible Agent Behaviour Specification Language (XABSL) has been specified [22] 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 [20] 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 behaviours 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 and others promote module reuse but do not build an architecture suitable for the reuse of appropriate modules. Usually code change is still required, other complexities introduced, and the abstraction of agent over object not fully exploited. In response to this common weakness, we put forward an MAS development process which includes analysis and design and also implementation and deployment. Our proposal requires that the development and maintenance of MAS is mainly focused on runtime re-configurable business rules, which capture the knowledge of agents in what, when, and how objects are to be used and reused. In this way code change is minimised.

2.3. *Agent-oriented modelling languages*

Agent UML (AUML) [3,4,10,27] extends standard UML to cover the needs of agent-oriented analysis and design. In the context of agents and Multi-Agent Systems, AUML class diagrams and interaction diagrams accommodate the concepts of agent, role, organisation, protocol, message, and so on with their corresponding notations. An Agent Interaction Protocol (AIP) can be defined to describe an agent communication pattern in a pre-agreed message exchange style. Agents intending to participate using any AIP must adhere to the AUML specification. Levelling is used for refinement of the interaction processes. AUML is seen as an extension of UML just like MAS has, rightly or wrongly, been seen as an extension of the OO paradigm. However, in AUML agents replace rather than co-exist with classes. Neither AUML nor UML support the collaboration of agents and objects towards system goals. This is somewhat at odds with the philosophy of AUML that states: "When it makes sense to reuse portions of UML, then do it; when it doesn't make sense to use UML, use something else or create something new" [3]. Using this same argument, it would be natural to reuse existing classes and add agents associated with them. We believe that the integration of agents and classes in an AIP would provide a layer of OO components supporting agents. This philosophy would boost technology reuse and facilitate smooth migration from OO development to MAS development, assisting the wider adoption of MAS. This is especially true in an open environment where agents have to collaborate with existing objects already running in that environment. The need for agents not only to communicate with other agents in their own society but also objects implemented in various languages is becoming more and more urgent in the next generation of systems, where agent technol-

ogy may be comparably dominant to object technology [24].

The Multi-Agent System Modelling Language (MAS-ML) [29], based on the TAO (Taming Agents and Objects) framework, also provides an incremental extension to UML for diagramming MAS comprehensively. OO concepts and modelling elements are preserved and Agent Oriented (AO) concepts and modelling elements added to complement existing ones. However, it remains at a conceptual and notational level with no contribution either in the form of a method to guide the abstraction of agents over objects or an approach to support the development of MAS using the modelling language.

Agent-Object-Relationship (AOR) [31] models show social interaction processes in organisational information systems in the form of interaction pattern diagrams. These model not only agents, but also ordinary objects, events, actions, claims, commitments, and reaction rules which dictate behaviours. AOR can also 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 will adapt and use it for our conceptual modelling of agents, rules, classes, and their interactions.

3. Approach

Below, we will introduce the primary elements of the ARC framework. These reuse existing technologies, upon which a fine-grained architecture is imposed, intending to create an immediately implementable MAS. More sophisticated mechanisms such as learning and reasoning can be added on top, where needed. First we will define the roles of the main elements.

Agent: A conceptual unit for organising 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 database of structured knowledge 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 objects, ready to be invoked to facilitate the execution of the system 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.

The ARC framework structures Agent, Rule and Class in a hierarchy: (i) agents are used to model conceptual domain units and guided by collections of rules in domains; (ii) rules are used to capture requirements and guide agent behaviours; and (iii) classes are employed by rules to support the function of agents.

The framework makes use of two principle design models built using its modelling elements. Firstly, a *Structural Model*, with UML Class Diagram as its counterpart in the OO world, is developed for structural relationship modelling. Secondly, a *Behavioural Model*, with UML Sequence Diagram as its counterpart, is proposed for behavioural interaction modelling. Both models are developed based on a Rule Model, which captures the requirements and re-constructs them, distinguishing agents from classes and instructing agents to use available classes at various opportu-

nities. Formal definitions are given to describe how requirements are transformed using the Rule Model and how the two design models are developed. Because in the models, rules have been individually defined and collectively compose agent communication patterns via tool support, MAS implementation from the design models is well assisted.

In the remainder of this section we will provide our case study before describe the model elements and the design models. Techniques of transforming the requirements models illustrated by the case study are given. The rationale and justification of our modelling approach are also provided.

3.1. Case study

To illustrate our agent-oriented software development methodology, a real life rail track management system is used as a vehicle through which a complete AOSE development process starting from the requirements specification to implementation is discussed. The functional requirements specification underpinning the approach, like many other traditional ones, is documented textually, in a form-based fashion using constrained natural language. This actual requirements document of the rail track system specification has been investigated and the appropriateness of our agent-oriented requirements modelling approach assessed using it. The system under study is mainly responsible for the running of a railway on a daily basis, monitoring train running with regard to incidents and ensuring the safety of the train service by conveying issues to relevant parties for resolution. Being a very complex and safety critical system, the document has more than 250 pages and contains a large number of function descriptions in a unified form as typified in Table 1. Relationships between function tables are not immediately obvious and this makes it difficult to maintain consistency.

The rail track specification concerns the Production Function for the network which in turn comprised three main areas: Train Running and Performance, Infrastructure Management and Performance, and Common Communications, each of which is sub-divided into Business, Incident, and Execution domains. These areas are not separated from each other but closely linked, representing different aspects of the specification. Train Running Business domain in the specified Production Function supports the principal service to customers, including delivery of planned train paths and response to requests for further train paths. Relat-

ing to the domain, *Train Operators* run *train journeys* on the network. Train Operators are normally freight or passenger train operating companies. Each train journey is first supplied in the form of a plan, either as part of the working timetable (planning), or the result of a *request* from a customer (re-planning). Details about the planning, re-planning, and implementation of train journeys are ignored here in the interest of concentrating on the chosen functions. The two sample functions in Table 1 are requirements for delivery of late additional train journeys, assuring their validity. "Accept Late Addition" is a major requirement of handling a late request for a train journey and "Validate Train Plan" is a supplementary requirement validating the new train journey created from the request received previously. Functions in the case study use a standardised form stating the cause of a function, the information it uses, its output, its required effect and so on. In some cases irrelevant sections may be omitted, and a "Sub-Req of" section added indicating one requirement is a sub-requirement of another, "Validate Train Plan" being a sub-requirement of "Accept Late Addition" in this case.

The IEEE standard for Software Requirements Specification (SRS) [15] provides a template for documenting a software requirements specification. The rail track specification largely follows this but has some extra fields and is organised by domains, the IEEE standard leaving the organisation style to the analysts. In the IEEE standard every specific functional requirement is required to include input (stimulus), output (response), and a function as performed by the system in response to the input or in support of the output. Given that most of these components should be present, whatever the format, only a modest effort is required to transform any alternative format specification to this format.

An analysis method has been proposed for building a similarly structured semi-formal representation [8]. The method identifies the current requirements via a reformulation in which the input and output references are made explicitly visible. For requirements formulated in such a structured manner four classifications are made: i) requirements on input only (independent of output); ii) requirements on output only (independent on input); iii) function/behaviour requirements (output is dependent on input); iv) environmental requirements or assumptions (input is dependent on output). These perfectly map to our individual function components of "Cause & Information Used", "Outputs", "Required Effect", and "Assumption", respectively, the last two containing the pre-condition and post-condition

Table 1
Sample functional requirements tables from the original requirements document

Identifier	AcceptLateAddition
Domain	TrainRunning Business
Description	To handle a late <i>request</i> for a <i>train journey</i> .
Cause	Receipt of a <i>request</i> for a <i>train journey</i> directly from a Train Operator or from the driver entering the production function's area. The <i>request</i> is provided in the form of a combination of relevant <i>train details</i> , <i>locations</i> and desired <i>timings</i> .
Assumption	The crew is competent for the route requested.
Information Used	Relevant <i>locations</i> .
Outputs	A new <i>train journey</i> , to Train Operator and others.
Required Effect	A new <i>train journey</i> is created from the <i>request</i> , and validated (ValidateTrainPlan). If the <i>train journey</i> is acceptable then it is distributed to all interested parties; otherwise the <i>request</i> is rejected or renegotiated. Having been accepted, the new <i>train journey</i> is known to the Production Function.
Identifier	ValidateTrainPlan
Domain	TrainRunning Business
Description	To validate a <i>train journey</i> .
Sub-Req of	AcceptLateAddition
Information Used	<i>Train journey</i> and relevant train details, sectional running times, locations and track restrictions.
Required Effect	The Production Function checks that the <i>train journey</i> : <ul style="list-style-type: none"> – complies with the Rules of the Route and, where relevant, the Rules of the Plan; – does not conflict with other train journeys and planned possessions; and – neither introduces unacceptable disruptions to this or other services nor increase the sensitivity of the Train Plan to disruption beyond “an acceptable level”. <p>The Production Function must also balance the needs of all its customers.</p>

of functions in our standardised format. A similarly structured specification in a standard form of Inputs & Source, Outputs & Destination, Action & Requires, and Pre-condition & Post-condition is presented in [30].

3.2. Agent/Rule conceptual modelling

Typically agents represent actors or information systems, identified in the knowledge domain, with business goals relating to business processes. Our agent-oriented development approach aims for an integrated development process, starting from requirements modelling, where agents are used to organise the requirements. Functional requirements tables, two examples of which are shown in Table 1, are differentiated in their identifiers, and thus are organised by the corresponding responsible agent units. In the generic case, an extra step to structure the requirements by their domain may be required. These requirements describing system features are collectively arranged around agents as the basis to direct agents on how to behave. Later, this organisation structure is used for agent-oriented design diagrams, reflecting relationships between agents and their responsibilities. Finally, the conceptual agents will be mapped to running agents. In Table 1, the requirement, “AcceptLateAddition”, states that additional train journeys would be requested to be deployed

if they are acceptable. Such a requirement must be assigned to an agent. In later sections it will be discussed how the agent acts in a business process to realise it. Before that, agent identification and organisation of requirements around agents will be presented.

The particular requirements document for the case study is organised by business domain. This is an alternate of an IEEE recommended template in which specification is organised by features [15]. A feature is an externally desired service by the system that may require a sequence of inputs to affect the desired result. A domain usually consists of multiple related features and domains interact regularly for shared goals. In Table 1 “AcceptLateAddition” belongs to the “Train Running” business domain, one of the domains that have been documented in the original “Production Function” specification. Also “AcceptTimetable” and “AcceptTimetableChange” are in the same domain. Thus, falling into the same knowledge domain, these requirements should be managed by the same agent, which we name “TrainRunning” agent. This agent is responsible for business domain of “Train Running”, including the deployment of train journey timetable, the change of the timetable, and additional train journeys, corresponding to these three pieces of requirements. All these services provided by the system have related features, so that it is appropriate to organise them into the same domain.

Table 2
Semantic meanings of symbols in definitions

Symbols	Semantic meanings & examples
<i>ownerAgent</i>	A relationship between a function requirement and an agent which the function is organised into. e.g. Function “Accept Late Addition” in Table 1 has agent “TrainRunning” as its <i>ownerAgent</i>
<i>causeAgent</i>	A relationship between a function requirement and an agent which causes it to function. e.g. Function “Accept Late Addition” has agent “TrainOperator” as its the <i>causeAgent</i>
<i>sourceReq</i>	A relationship between a business rule/function with the requirement it originates from. e.g. A business rule transformed later has “Accept Late Addition” in Table 1 its <i>sourceReq</i> , and a business function has “Validate Train Plan” its <i>sourceReq</i>
<i>isInvolvedIn</i>	A relationship between a business function and a business rule, one involved in the function of the other. e.g. A business function transformed from requirement “Validate Train Plan” <i>isInvolvedIn</i> a business rule transformed from requirement “Accept Late Addition” to assist its function
<i>connectsTo</i>	A relationship between two business rules, indicating the production result of one is sent to the other for processing. e.g. A business rule transformed from “Accept Late Addition” <i>connectsTo</i> another rule which accepts the train journey created by it.
<i>isDecodedFrom</i>	A relationship between a business object and a message, indicating that the object is constructed using the message as an information source, supported by associated classes for the construction. e.g. A “TrainJourney” object <i>isDecodedFrom</i> a late journey request message
<i>isEncodedTo</i>	A relationship between a business object and a message, indicating the object is embedded into the message to be sent. e.g. The “TrainJourney” object, once gets validated, <i>isEncodedTo</i> a message to be distributed to interested parties

The concept of agent should be distinguished between the requirements modelling and the implementation. Agents during this phase are used to organise the requirements. When implemented as software, they are responsible for meeting their corresponding requirements. What follows is a formal guide for agent identification in the requirements modelling process. Prior to that symbols used in our definitions and their semantic meanings are given in Table 2, illustrated with examples.

Definition 1. Agent Identification

Let F be a set of f that constitutes the specification, ID be a set of f . Identifier, and A be a set of delegated agents. There exists a function T , from F to A , given that $f_1, f_2 \in F$, $a_1, a_2 \in A$, where $a_1 = T(f_1)$, $a_2 = T(f_2)$, we have:

$a_1 = a_2 \Leftrightarrow f_1$. Domain = f_2 . Domain.

f . *ownerAgent* is defined as $T(f)$.

For the case study, let f_1, f_2 be “AcceptTimetable Change” and “AcceptLateAddition” respectively. Both belong to the TrainRunning Business domain. This indicates they should be organised together in the same agent.

On completion of the grouping of requirements and assigning them to agents according to their nature, the requirements would be further re-structured, forming rules and functions that guide the behaviours of the agents, this mapping from the original requirements to the rule/function elements and the construction of the elements lay a foundation of our agent-oriented design

diagrams and serve as the transition from requirements modelling to designing.

Definition 2. Rule & Function Distinction

Let f_1, f_2 be two functional requirements tables, P_1, P_2 be property sets of f_1, f_2 . Let s_2 be a set of nouns that appear in f_2 . (Required Effect).

If “Sub-Req of” $\in P_1$, “Sub-Req of” $\notin P_2$, f_1 . (Sub-Req of) = f_2 . Identifier, and f_1 . Identifier $\in s_2$, then: f_1 makes a business function F , and f_2 makes a business rule R ,

F . *sourceReq* = f_1 , R . *sourceReq* = f_2 , and F . *isInvolvedIn* (R).

As “ValidateTrainPlan” has a “Sub-Req of” section, with the identifier of “AcceptLateAddition” in it, indicating that it is a sub-requirement of the latter piece of requirement. Also in the “Required Effect” section of “AcceptLateAddition”, it mentions about “ValidateTrainPlan”, indicating it uses the latter piece of requirement to assist its own function. Therefore “AcceptLateAddition” is represented in a business rule, “ValidateTrainPlan” is represented as a business function, and: “ValidateTrainPlan”. *isInvolvedIn* (“AcceptLateAddition”).

3.3. Rule modelling (for requirements modelling)

Rules are central in our requirements modelling approach. They provide structured requirements informing agents how to behave, ultimately becoming executable requirements. Rules represent functional requirements, and the use of rules makes these require-

ments explicitly inter-related, since they dictate the action and reaction of agents in their collaborations. In this sub-section we will demonstrate the rule modelling and the transformation of rules from the functional requirements tables.

A rule essentially captures a system function requirement in terms of the receipt of an event (normally modelled as a message), a condition that is satisfied or not and (after some processing) what action an agent should take. The overall structure of this resembles a function input, function use context, function process and function output. From this basic structure which is shared by all rules, we split each rule into several related compositional parts and each captures one aspect of the requirement for the function. For such a function described in Table 1, its “Cause” section is used to make the rule “event”; its “Information Used” and “Required Effect” sections are used to make the rule “processing”; and its “Required Effect” and “Outputs” sections are used to make the rule “condition” and “action”. Thus, events cause agents to execute rules. If certain conditions are satisfied, then some actions are triggered, which in turn includes generated events sent to other agents. We also conceive “belief” as an integral part of the rule structure, being a collection of knowledge that an agent can learn from the messages received from other agents. Overall, a rule can be modelled as Fig. 1.

Figure 1 shows that an agent processes a rule using the following steps.

1. Check event – find out if the rule is applicable to deal with the perceived event.
2. Do processing – decode the incoming message, construct business objects to be used in later phases.
3. Check condition – find out if the (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 (possibly the triggering one). If c_i is not satisfied, then go back to Step 3 and check the condition c_{i+1} .
5. Update beliefs – according to the information obtained from the message just received, the knowledge of the agent to the outside world is updated.

Business rules, as we specify here, make agent another abstraction over object. An agent uses a dedicated rule for a specific task and, in turn, a rule uses business classes to complete it. What and how classes are to be invoked can be specified in rules and the con-

figurability of them brings adaptivity. Mutable requirements on components collaboration can be externalised in rules, and the agent knowledge is made updatable for the collaboration partners, events processing, and response messages production. Different actions can be set in rules as reactions to different conditions, in an order of user preferences/priorities. Details on rule configuration are discussed in Section 4.

The two components of rules, event and action, allow rules to be inter-connected, and therefore also the agents that own the rules, the communication of which is through passing messages defined in event and action components.

Definition 3. Rule’s eventMessage and actionMessage

Let R be a rule, R . eventMessage represent the message information that causes R to function, and R . actionMessage represent the message information that is created by R , then:

R . eventMessage $\in R$. sourceReq. Cause $\cup R$. sourceReq. (Information Used), and

R . actionMessage $\in R$. sourceReq. (Required Effect).

Let $R1, R2$ be two rules, and $R1$. sourceReq. ownerAgent = $R2$. sourceReq. causeAgent, then:

$R1$. connectsTo ($R2$), and $R1$. actionMessage = $R2$. eventMessage.

In the case study, the request for an additional journey, the content of which includes relevant train details, locations and desired timings constitutes the eventMessage of “AcceptLateAddition”, as “Cause” and “Information Used” describe, this message comes from “TrainOperator” as the result of an actionMessage of one of its rules. On the other hand, the actionMessage of the rule consists of a newly created train journey, as “Required Effect” describes, and this message becomes an eventMessage to the rule that receives and processes it.

Each rule specification can be derived from a transformation process guided by the following function.

Definition 4: Rule Transformation $FunctionalReqTabToRule (FunctionalReqTab, Rule)$ {

sourcef: $FunctionalReqTab$;

targetr: $Rule$;

source condition 1

f. “Sub-Req of” = $NULL$

mapping

r. Event. messageFrom := f. Cause. getActor ();

r. Event. messageFromContent := f. Cause. getContent ();

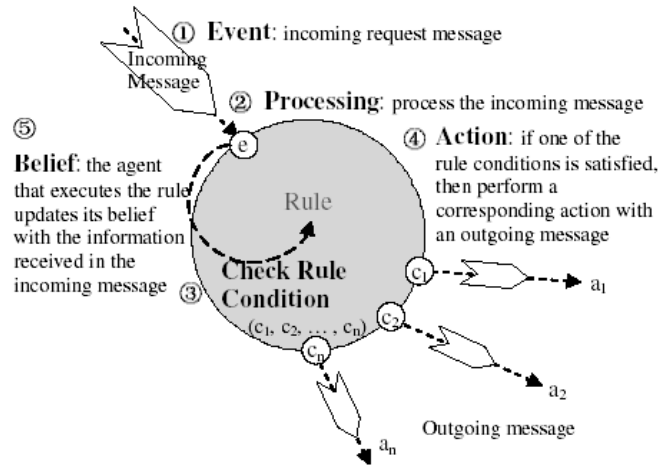


Fig. 1. Rule Model.

```

inv: r. Event. messageFromContent == f. InformationUsed;
r. processingMethod := f. RequiredEffect. getProcessingMethod ();
r. Processing := (f. RequiredEffect. productions = r. processingMethod (f. InformationUsed));
switch (f. RequiredEffect. Case (i))
{
  Case (i): {
    r. Condition := f. RequiredEffect. Case (i). (ConditionStatement);
    r. Action. messageTo := f. RequiredEffect. Case (i). getActor ();
    r. Action. messageToContent := f. Outputs. Case (i). getContent ();
  }
}
source condition 2
f. "Sub-Req of" ≠ NULL
mapping
Add f to f. involvedRule (). associatedClasses (f). processingMethods ();
}

```

To illustrate this transformation process for the case study, we shall take the function, "AcceptLateAddition". This will be transformed to a rule as it does not have a "Sub-Req of" section. It does have an "Event" component, which will trigger the rule to function. The event message comes from the "TrainOperator" agent ("Receipt of a request ... from a Train Operator" in the "Cause" section) and the message content would be a request, including relevant train details, locations

and desired timings ("The request is provided in the form of ..." in the "Cause" section). On receipt of an event, a rule processes it by consuming the "Information Used" and produces some productions that will be used later, which forms a rule's "Processing" component. In this case, "AcceptLateAddition" decodes the request message, obtains relevant location information ("Information Used") and so on, and creates a new "TrainJourney" business object ("A new train journey is created from the request" in the "Required Effect"), the production in this case as given in the "Outputs" field. A rule will react differently in different conditions, and the conditions are related to the productions produced in the previous "Processing" procedure. For "AcceptLateAddition", two conditions are considered. In the first case ("If the train journey is acceptable" in the "Required Effect"), the created "TrainJourney" ("Outputs") object is encoded into a message, and sent off ("then it is distributed to all interested parties"). In the other case, another reaction will be taken ("otherwise the request is rejected or renegotiated"). Methods of business objects will assist the condition checking: "ValidateTrainPlan" is such a function ("... and validated (ValidateTrainPlan)"). Correspondingly, by using the transformation in source condition 2, the functional requirements table "ValidateTrainPlan" would be transformed to a business function due to its presence of "Sub-Req of" section, and it is added to the business class "TrainJourney", which is associated with "AcceptLateAddition", the rule that "ValidateTrainPlan" is involved in.

Thus the rule "AcceptLateAddition", transformed from the functional requirements table in Table 1,

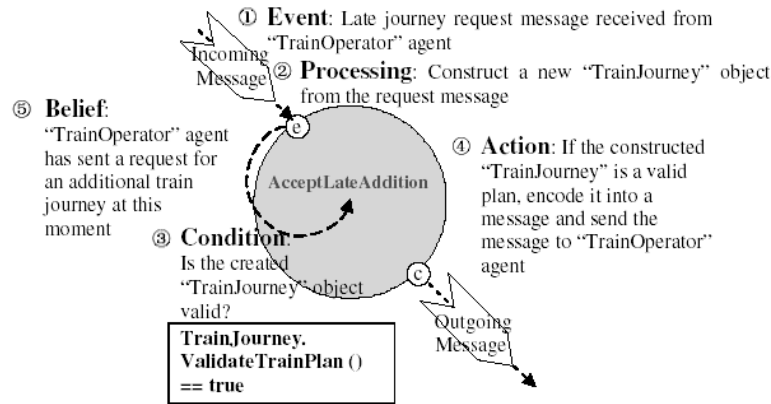


Fig. 2. Rule Model of "AcceptLateAddition".

would be specified as follows and as shown diagrammatically in Fig. 2.

1. Receive a late journey request message from "TrainOperator".
2. Construct a "TrainJourney" object using the information contained in the message.
3. If the created object can pass the "ValidateTrainPlan ()" evaluation method.
4. Then send a message with the created "TrainJourney" to "TrainOperator" and other relevant agents (the alternative condition is omitted for simplification), and
5. Add the belief that the "TrainOperator" has made such a request at this moment.

Rules can be documented as such with five-section descriptions for requirements specification. In Section 4.1, they are further formatted, and in Section 3.8, they are shown their collaboration in business processes, making their role easily understandable in a big picture.

The rule specification in Fig. 2 tells the agent "Train-Running" to follow a process, in the sequence numbered from 1 to 5, for deploying an additional train journey, when this rule is found by the agent to be the appropriate rule to execute in the context.

The Rule Model decomposes the functional requirements into the compositional components of: {event, processing, condition, action, belief} and so facilitates the transition from the requirements to the final implementation in the agent-oriented development process. These components can be translated into programming like statements. Once granted translation capability, agents would be able to interpret these statements, their performance conforming to the rules which capture the requirements.

While agents are running, they perform rules in an event-handling way, reacting differently to different conditions. Business classes managed by the agents can be invoked to decode incoming messages, check conditions, and encode outgoing messages. The separation of classes, the passive entities, from the agents, the active entities, and business functions from business rules, establishes an agent/rule and class/method hierarchy. While the requirements modelling is being completed, the system structure becomes evident, and the design models follow for construction. The Structural Model and the Behavioural Model are the two design models that enhance the traditional UML models. Rules are integrated into these for the purpose of agent-oriented system design, based on the ARC framework.

3.4. Structural modelling

An ARC Structural Model diagram has the Class Diagram, the backbone of UML, as its counterpart in the OO models. Agents are regarded as superior to classes in this model, just like classes are regarded as superior to attributes in OO models. Referring to Fig. 3, 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 behaviours of the agent.

In such a model as Fig. 3 shows, two agents are connected by a "collaborate" line, if they interact with each other, by passing messages. Standard methods invocation between classes is replaced by rule collaboration between agents, where one rule produces a message and the other processes it. What and how components

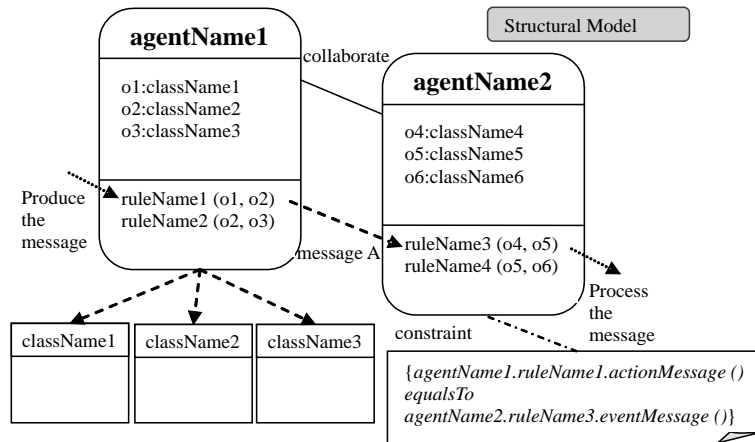


Fig. 3. ARC Structural Model.

are to be invoked can be configured with rules and such configuration information is obtained by agents at runtime to ensure the deployment of the most up-to-date requirements. Eventually, the requirements on component interactions are replaced by agent interactions and such knowledge is formatted in business rules. Another layer of abstraction is hence achieved.

For each agent identified using *definition 1 (Agent Identification)*, there should be a rounded cornered box to represent such a conceptual domain, and a list of rules organised into the bottom compartment of each agent according to their identifiers. A list of classes to be put in the middle compartment of each agent, the relationships between classes with the agents/rules are guided by *definition 5 (Rule's associated Business Class)*, and *definition 6 (Agent's managed Business Class)*, and the relationships between agents are guided by *definition 7 (Agent Collaboration)* given below.

Definition 5. Rule's associated Business Class

Let C_R be a set of business classes that are associated with rule R , S be a set of nouns that appear in R . *sourceReq. (Required Effect)*, then: $C_R \in S$.

There exist two business classes C_1 and C_2 , and their instances O_1 and O_2 , that:

$$C_1, C_2 \in C_R,$$

$$O_1.isDecodedFrom (R. eventMessage),$$

$$O_2.isEncodedTo (R. actionMessage).$$

Definition 6. Agent's managed Business Class

Let agent $a \in A$, C_a be a set of business classes managed by a , rule set $S_R = \{R_1, R_2, \dots, R_n\}$, for any $1 \leq i \leq n$, R_i . *sourceReq. ownerAgent = a*, and there exists no rule $R_j \notin S_R$, that R_j . *sourceReq. ownerAgent = a*.

Let C_{R_i} be a set of business classes associated with rule R_i , then:

$$C_a = C_{R_1} \cup C_{R_2} \cup \dots \cup C_{R_i} \cup \dots \cup C_{R_n}.$$

The following can be inferred:

Let C_R be a set of business classes associated with rule R , C_a be a set of business classes managed by R . *sourceReq. ownerAgent*, then $C_R \subseteq C_a$.

Definition 7. Agent Collaboration

Let $s1$ be a set of nouns that appear in f . *Cause*, $s2$ be a set of nouns that appear in f . *Outputs*. If $a1, a2 \in A$, $a1$ and $a2$ collaborates with f . *ownerAgent* while f is in function is defined as: $(a1 \ \& \ a2). collaboratesWith^f (f. ownerAgent)$, then:

$$(a1 \ \& \ a2). collaboratesWith^f (f. ownerAgent) \Leftrightarrow a1, a2 \in s1 \cup s2,$$

f . *causeAgent* is defined as $a1$, and f . *outputAgent* is defined as $a2$ with an assumption that $a1 \in s1$ and $a2 \in s2$.

A rule's associated business classes enable the rule to function as described in the rule's "Required Effect". "TrainJourney" is an identified class for "AcceptLateAddition", which appears in its "Required Effect" section. It is the real business entity that exists while the rule is being executed. Any intermediate class that is designed for implementation is not considered as a "business class". An instantiated object of "TrainJourney" will be constructed from the event message of the rule, received from "TrainOperator".

An agent's managed classes are the collection of all the associated classes of its rules. "TrainJourney" is a class associated with rule "AcceptLateAddition". Hence, it is managed by the "TrainRunning" agent which owns that rule. The name of this class is to be

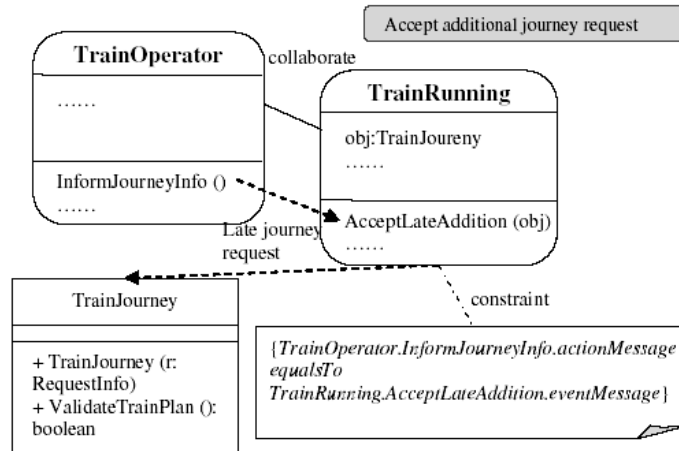


Fig. 4. Structural Model diagram for case study.

declared in the middle compartment of the agent and the box that represents the class is connected with the box that represents the agent in the diagram.

Agent "TrainOperator" and "TrainRunning" are therefore collaborators, since "TrainOperator" not only causes the rule "AcceptLateAddition", owned by "TrainRunning", to execute, but also it is the recipient of the result created by the rule.

In Fig. 4, a diagram for the case study using this modelling technique is illustrated. According to *definition 1 (Agent Identification)*, the *ownerAgent* of "AcceptLateAddition" is "TrainRunning". Using *definition 7 (Agent Collaboration)*, the *causeAgent* and *outputAgent* of "AcceptLateAddition" are both "TrainOperator". These are the only agents involved in this case. The two recognised agents represent the train running planning domain and the train operating company domain respectively. "TrainOperator" has a rule of "InformJourneyInfo" that constructs a request object, packages it into a "Late journey request" message and sends the message to "TrainRunning". This may reflect increasing demand of train journeys on special dates or events in real life. To respond to such requests, "TrainRunning" plans additional journeys using the rule "AcceptLateAddition". The required function of deploying additional train journey is documented in the functional requirements table shown in Table 1.

As it is illustrated in the diagram, there is a collaboration relationship between the two agents, as defined by *definition 7 (Agent Collaboration)*. During the processing of rule "AcceptLateAddition", a "TrainJourney" object is created from the journey request information in the incoming message, as defined by *definition 5 (Rule's associated Business Class)*. A construction

function is used for decoding. The constructed object should pass a validation check before being put into use. Both the construction function and the validation check function belong to the business class "TrainJourney", which is managed by "TrainRunning", as defined by *definition 6 (Agent's managed Business Class)*. "ValidateTrainPlan", presented in Table 1 as one of the functional requirements tables, defines functions to support the rule "AcceptLateAddition" to operate. It is not made as an independent rule, as defined by *definition 2 (Rule & Function Distinction)*. There is also a constraint between the collaboration agents, as dictated by *definition 3 (Rule's eventMessage and actionMessage)*.

In the Structural Model, a system structure with agents, rules, classes, messages, and their relationships are organised. Agents are higher level entities, their behaviours being governed by rules, and business classes with functions are in a lower level, used by the agents. After the essential element identification and their structural organisation, system behaviours can be modelled based on this. The next three sections provide further explanation of the relationships among modelling elements. It will also be demonstrated that they make the implicit relationships among the traditional functional requirements tables explicit, and if such an approach is adopted, they support the requirements consistency check.

3.5. Rule dependency: hierarchical encapsulation

There is a direct dependency between two rules where there is a collaboration relationship between the agents they belong to, because the event of one rule is dependent on the action of the other. Assume R1

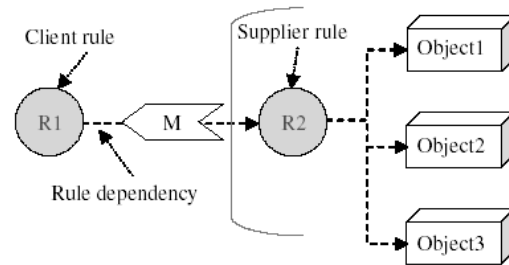


Fig. 5. Rule dependency relationship.

has sent a request message to R2, asking for a service. Assume also that R2 has done some computation and now is sending a reply message back to R1, as shown in Fig. 5. Rule R2, which provides the service is termed the *supplier rule*, while rule R1, which receives the service, is termed the *client rule*. This client rule will become the supplier rule if/when it sends a message to another rule and provides its own service. In the above case, there is no direct dependency from the client rule R1 to the three business data objects that supplier rule R2 has used to fulfil the service. If these business objects change, corresponding change may be necessary for R2 when it invokes the objects, for example while it checks rule condition or constructs `actionMessage`, etc. In most cases, the change is in the computational logic, internal to the objects, not the produced message format, so that the change stops at R2. If the business objects maintain the same interfaces while their implementation is changed, then the change stops at the object level and there is no effect on either of the rules. In the lower level, by using a good OO design methodology, this can be achieved without much difficulty.

Consequently, the relationships between agents resemble the relationships between objects in the OO world, while internal business classes of agents are encapsulated and known only by their owner agents, analogous with private methods of objects, which are encapsulated and only known internally by their owner objects [19]. Changing private methods of an object has no direct effect on its associated objects, and likewise, changing business classes of an agent has no direct effect on its collaborated agents. These patterns of design model in agent/rule and class/method hierarchy promise better control of dependencies. Because agent is another level of abstraction over class, an advanced “hierarchical encapsulation” is achieved.

3.6. Rule-Rule relationship: a message serves as a rule connector

The communication between two agents, through two designated rules, is by message passing. This message can be viewed as a connector [28]. It connects two rules in the sense that the message is produced as an action of one rule and processed as an event of the other. In Fig. 6, such a connector between Agent1.Rule1 and Agent2.Rule2 is shown.

If we use “Agent1.Rule1.actionMessage” and “Agent2.Rule2.eventMessage” to represent the sent message for Agent1.Rule1 and the received message for Agent2.Rule2 respectively, we have the following.

```

Agent1.Rule1.messageDestination == Agent2.Rule2 &
Agent2.Rule2.messageSource == Agent1.Rule1
⇒ Agent1.Rule1.actionMessage equalsTo Agent2.
Rule2.eventMessage
  
```

This has been formally described in *Definition 3 (Rule’s eventMessage and actionMessage)*, the outcome of which is, *Rule1.connectsTo (Rule2)*. Despite the straightforward and obvious relationship, such a constraint is not trivial. Implicit requirements are therefore captured and the consistency of the original specification can be checked. Assume that two domains are associated at Function1 and Function2 and each belongs to one of the domains. If Function1 causes Function2 to be effected, then there is a direct relationship between the <Required Effect> & <Outputs> section of the first function and the <Cause> section of the second function. In other words, there is a relationship between Agent1.Rule1.action and Agent2.Rule2.event, provided Function1 maps to Rule1 and Function2 maps to Rule2. In this scenario, a message must pass from Agent1 according to its Rule1, to Agent2. Likewise, Agent2 expects to receive a message from Agent1, according to its Rule2, the two messages being the same one in fact. For our case, a rule called “InformJour-

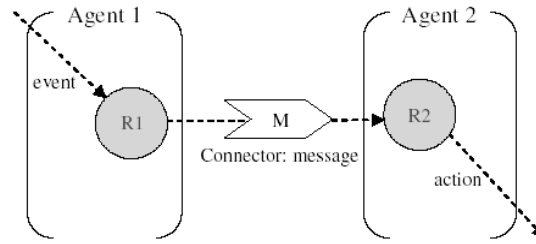


Fig. 6. Rule-Rule relationship.

neyInfo” that belongs to “TrainOperator” sends a message to “TrainRunning”, and the rule “AcceptLateAddition” that belongs to the latter agent receives and processes that same message. The event of the latter rule must conform to the action of the former rule. This implies that the corresponding sections of InformJourneyInfo and AcceptLateAddition need to match; otherwise there would be conflicts in the original requirements document. The risk of failing to change a requirement to reflect a change in a related requirement is thus avoided. Since any natural language based specification can rarely be guaranteed free of inconsistency, a supporting tool can be used to check the formatted rules for consistency. This is described in Section 4.2, using the captured implicit relationship described here.

3.7. Agent-Agent relationship: A rule serves as an agent interface

A rule is an interface between agents in the sense that the message passing between rules describes the collaboration relationship between two agents. It is through the definition of a rule that the owner agent knows how to interact with some other agent. Changing an agent’s rule can change which agent it collaborates with and/or their collaboration pattern. We state that a rule *isAssociatedWith* an agent if it serves as an interface between this agent and the agent it belongs to. The rule *isAssociatedWith* a business class if an instantiation of the class is used by the rule to fulfil its service as an interface.

Figure 7 illustrates the interaction between rules (represented as circles) and therefore agents (represented as rounded rectangles) and also their use of objects. Agent2 is collaborated with Agent1 at rule R. Rule R of Agent1 makes uses of Object2 for providing a service for or requesting a service from Agent2. Whatever the case, such activities as eventMessage decoding, condition checking and actionMessage encoding are involved while rule R functions. There is a message

shared between Agent1 and Agent2, the outcome of which is that this message is decoded as an eventMessage or encoded as an actionMessage, depending on the direction of message passing or their dependency relationship described in Section 3.5. Therefore we have the following.

Agent1.Rule1.isAssociatedWith (Agent2) at Agent2.
Rule2
 \Rightarrow (*Agent1.Rule1.actionMessage equalsTo Agent2.*
Rule2.eventMessage OR
Agent1.Rule1.eventMessage equalsTo Agent2.Rule2.
actionMessage);
Agent1.Rule1.isAssociatedWith (Agent1.Business
Class1)
 \Rightarrow *Agent1.Rule1 invokesOn Agent1.InstantiationOf*
BusinessClass1

As described in Section 3.5, this constraint also captures implicit requirements and enforces consistency in and with the original specification. A rule is associated with an agent either because its event is triggered by that agent or its action triggers that agent to process it as an event. A rule is associated with a business class for analysing the eventMessage, checking if its condition is satisfied, or constructing an object for the actionMessage. In this case, the rule “AcceptLateAddition” of “TrainRunning” is associated with “TrainOperator” because that agent triggers it to function, as described by the rule event in Fig. 2. Rule “AcceptLateAddition” is associated with “TrainJourney” because that class is used by it to construct an object from the request information and check its validity, as indicated by the rule condition and rule action depicted.

Thus, a rule in the model has two distinctive interfaces: an external interface between the rule and its associated agent and an internal interface between the rule and its associated class.

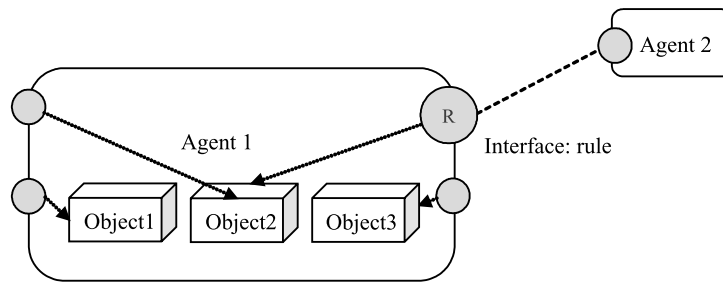


Fig. 7. Agent-Agent relationship.

3.8. Rule language

Syntax and semantics of rules have been introduced in the previous sections, the specification of which is like a language. The rule language plays a similar role to other interface languages like the Object Management Group's Interface Definition Language (IDL) [26]. IDL defines the interface only through which client objects can communicate with server objects in a distributed environment. This enforces the encapsulation of the internal structure and mechanism of the server objects. An interface definition specifies operations to be performed, inputs and outputs, allowing clients and servers to encode and decode values for their travel over the network, regardless of their platform, operating system, programming language, and so on [26]. Our rule interface specifies messages that can be passed between agents and this sets a contract through which the interaction pattern between communication components related by it can be enforced. A client agent is not aware of how a server agent processes its actionMessage. However, that message is the eventMessage to the server agent, so that a rule of it tells it how to react. This ensures the encapsulation of agent functions, and the message passing over network is also technology independent.

3.9. Behavioural modelling

Of the two design models of the ARC, the Structural Model is the foundation, with the Rule Model being based on this and the Behavioural Model built on top of both of them. The Behavioural Model smoothes the transition from design model to implementation.

Structural Model diagrams represent system structure, and the static relationships of the compositional elements of the system. In the OO world, UML Class Diagrams are complemented by Sequence Diagrams for behavioural modelling, and likewise, ARC Behavioural Model is designed to capture the behavioural

scenarios. Inspired by the Agent-Object-Relationship model [31], we group associated agents/rules/classes and show their behaviours and interaction processes for achieving certain goals. Rules are event-driven processing units for agents, working by invoking classes. A Behavioural Model diagram corresponding to Fig. 3, assuming that the response message is sent back to the initial agent, is shown in Fig. 8. It visualises the actual system function in a sequence of actions, with only a satisfied {condition, action} couplet shown in this scenario. Figure 9 shows such a model diagram for the case study. In this, the "TrainOperator" agent sends a message ("Late journey request") to the "TrainRunning" agent, initiating some (sequence of) other rule(s) resulting in, finally, a "New train journey & Confirmation of its acceptance" message being received by the "TrainOperator" agent.

In traditional OO systems, objects are 'aware' of which other objects they will pass messages to, but are unaware of which objects will pass messages to them. Conversely, the ARC framework achieves full architecture independence (two-way encapsulation), in that the details of where objects will send messages is also hidden [25]. This achievement results from managing agent collaboration with rules, which are not hard-coded but configurable. For each agent, not only its collaboration partner, but also the way it collaborates with other agents can be amended, according to the changing requirements. The following sections will demonstrate the configurable rules and a tool for the configuration.

3.10. Hierarchical agent organisation in ARC

Grouping agents together provides an abstraction mechanism that reduces the complex interdependency between agents during analysis and design, an essential requirement when the number of agents increases. This is analogous to the use of packages in Java for class organisation, and Nested Protocols [4] and Leveling [27]

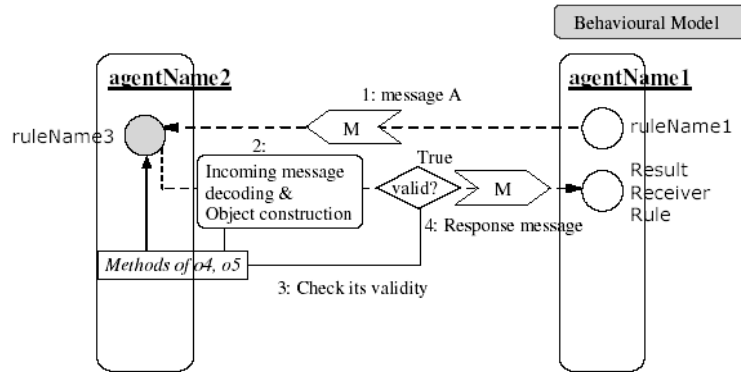


Fig. 8. ARC Behavioural Model.

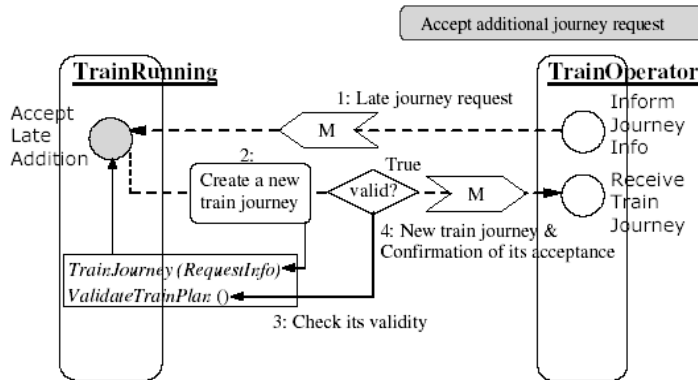


Fig. 9. Behavioural Model diagram for case study.

in AUML for interaction protocol organisation. Using the ARC framework, we do not expect a large number of agents, although we may expect many rules. This is a result of many requirements falling into only a few distinct domains. However, it is possible to decompose agent responsibilities and thus build a hierarchical agent organisation and so facilitate management of ARC systems.

The organisational structure of MAS provided by the ARC framework is natural and straightforward. An agent represents a business domain. Requirement subsets within this business domain are organised and dedicated to sub-agents. Collectively, the sub-agents share the responsibilities of the original agent from which they are derived. In Fig. 10 “Train Running” domain is depicted, being split up and three sub-domains with different types of functions distinguished: one receives and delivers planned train services; another responds to requests for unplanned train journeys; and the last handles the impact of incidents on train services by re-scheduling. Since the three sub-domains have distinct

duties in nature, sub-agents can be dedicated to them respectively: TrainRunning Planned (TRP); TrainRunning Unplanned (TRU); and TrainRunning Incident (TRI). When events come to the original agent, a first level agent named TrainRunning, they are forwarded to various second level agents to process. For instance, the working timetable (e1) is forwarded to TRP, the updated timetable (e2) and additional journey requests (e3) to TRU, and incidents (e4) to TRI. When a late journey request comes, for example, the TrainRunning agent forwards it to TRU. After the processing, an action message is produced and sent to another first level agent through TrainRunning agent.

This mechanism is fully supported by the existing ARC framework, simply by defining rules. Being in a simpler form with no need of condition evaluation or object invocation, each such rule specifies the receipt of a certain type of events from the same level of agents as itself, and forwards it to one of its sub-agents in a lower level, as shown in Fig. 10. Analysts and designers of MAS can use this mechanism to break down the busi-

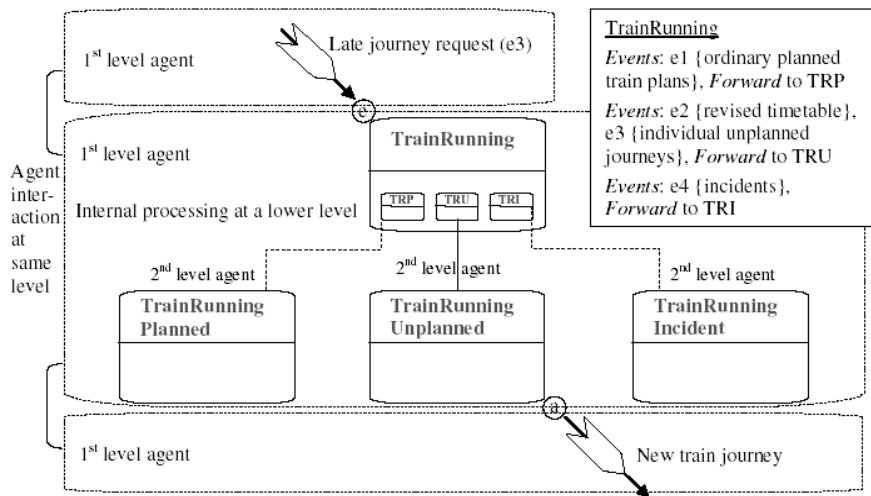


Fig. 10. Hierarchical agent organisation.

ness domains and build more levels until a manageable number of lower level agents appear in each agent one level higher, each bottom level agent has a manageable number of functions, and the complexity of independency is sufficiently reduced. Analysis can be carried out step by step, top down with no concern about the details of all lower level agents inside the current level agents. In this way the architecture can be abstracted at different levels.

4. Implementation & deployment

4.1. XML model for rule implementation

As it is shown in the rule specification in Fig. 1, one rule is composed by five elements. These are encoded using respective tags in XML. They include details about the triggering event, the processing of the event, a series of {condition, action} couplets and the priority. The XML representation (the XML Schema can be inferred) for rule "AcceptLateAddition" is shown in Fig. 11.

XML-based rules add precise definitions of agent behaviours to ARC models, when the model elements are mapped to these definitions. This is something UML diagrams lack [11]. In general, each agent reacts to the receipt of a message by executing a rule using the following process.

1. Get a list of its managed rules that are documented in a XML 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, 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, check the next condition, 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>.

```

- <rule>
  <name>AcceptLateAddition</name>
  <business-process>Late train journey request handling</business-process>
  <owner-agent>TrainRunning</owner-agent>
  <global-variable>
    <name>trainJourney</name>
    <type>TrainJourney</type>
  </global-variable>
  - <event>
    <type>receipt of message</type>
  - <message>
    <from>TrainOperator.InformJourneyInfo</from>
    <to>TrainRunning.AcceptLateAddition</to>
    <title>Late journey request</title>
    - <content>
      - <requestInfo>
        - <trainDetail> </trainDetail>
        <locations>
          <from>Belfast</from>
          <to>Dublin</to>
        </locations>
        <date>2005/07/28, 10:00 a.m.</date>

        </requestInfo>
      </content>
    </message>
  </event>
  <processing>
    trainJourney = new TrainJourney (requestInfo)
  </processing>
  <condition>
    trainJourney.ValidateTrainPlan () == true
  </condition>
  - <action>
    <type>send a message</type>
    - <message>
      <from>TrainRunning.AcceptLateAddition</from>
      <to>TrainOperator.ReceiveTrainJourney</to>
      <title>confirm the late journey request is accepted</title>
      - <content>
        - <responseTo>Late journey request</responseTo>
        - <result>accepted</result>
        - <trainJourney>
          - <journeyId>200510010100</journeyId>
          - <journeyDetail>
            <from>Belfast</from>
            <to>Dublin</to>
            <date>2005/07/28, 10:00 a.m.</date>

            </journeyDetail>
          </trainJourney>
        </content>
      </message>
      (Also send this message to other interested parties)
    </action>
    <priority>5</priority>
  </rule>

```

Fig. 11. The XML definition for rule “AcceptLateAddition”.

8. Analyse the business object which has 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 go to Step 3.
10. Wait for the next event.

The definition of rule “AcceptLateAddition” in XML is based on its Rule Model transformed from requirements given in Section 3.2, and is executable to agents. To illustrate its execution, suppose all previous rules

managed by the agent “TrainRunning” are not applicable and now it checks the applicability of this rule. The agent parses the message just received and finds out that the message has come from an agent with the name “TrainOperator”. The agent knows its rule “AcceptLateAddition” is defined to deal with the message received from that agent, because according to the rule definition, the content of XML element <event>/<message>/<from> matches with that agent in the name. Also the message content has the same structure as specified in the rule: In the rule definition shown in Fig. 10, the content between tags

in italic is the possible message content, where the rule specifies it will only accept events with that kind of content structure. Then, by invoking business classes managed by the agent, a business object of “trainJourney” can be constructed, the name of which is declared and shared as a global variable, and its validity can be evaluated. The “trainJourney” object can be built from the content of the `<event>/<message>/<content>` structure of the received message. Its validity can be checked using the “ValidateTrainPlan ()” method of the business class. If the “trainJourney” is valid then the condition for executing the rule is satisfied. A corresponding message will then be structured using the created “trainJourney” object, the name of which refers to the global variable, and sent to all interested agents, including “TrainOperator”, again as it is specified in `<action>/<message>/<to>`. Finally, the “TrainRunning” agent adds the knowledge that the “TrainOperator” agent has sent a late train journey request at this time to its own beliefs. When enough such information is collected, business reports can be built for analysis for the later use. In the whole rule execution process, if the event does not match or the condition is not satisfied, the next candidate rule with the highest priority will be tested for applicability and executed in the same way.

4.2. Tool support

A tool has been developed to support the specification of ARC Behavioural Models and relate the rule model elements with their XML model. Visual inspection of the completeness and consistency of the Behavioural Models is possible and easy. Figure 12 captures a window from this tool showing the construction of a model diagram in its main panel and the definition of a rule via a user-friendly tree structure.

Each rule specified in the tool maps to one agent behaviour. XML-based rules are translated by agents at run-time. Pseudo code of the agent behaviour represented by “AcceptLateAddition” in the case study is shown in Fig. 13. A shared module called “Rule” (It is part of the JavaBeans in Fig. 14 to be discussed later) facilitates agents to translate XML-based rules as their behaviours. The module accesses the XML definition of rules and can be used to assemble corresponding objects. The methods `getPriority ()`, `getEvent ()`, and `getAction ()` are provided in a “Rule” module.

4.3. Deployment

As soon as ARC models are specified graphically in the supporting tool with the business rules specification in XML, agent interaction models, rule reaction patterns and message flows are established accordingly. JADE [16] (Java Agent DEvelopment) framework is one of the platforms that conforms to FIPA [10] standards for developing agents. The actual agent system that will run on the JADE platform in a distributed network is initially generated from the tool. A central XML-based business rule repository is deployed in the network, containing the rule definitions and the registered business classes that are used by rules. A JavaBeans component is implemented, 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 business knowledge. The edition through the tool for the knowledge change is saved in the XML repository using the same JavaBeans.

As shown in Fig. 14, all agents access the repository via the JavaBeans as well, in order to obtain the most up-to-date knowledge. 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 Knowledge Model can be changed using the tool. Then the generated Agent Model can be immediately updated with new requirements knowledge. Eventually agents can always get the desired behaviours as soon as they have been specified through the tool, and can be continuously updated.

Such deployment provides a viable solution for complex applications in the dynamic environment of internet. The supporting tool for maintenance of the business model, the business rules database and the agent system are deployed in a distributed internet environment. The rule database for dynamic agent behaviours is stored at the back end, the configuration of which is through the tool. Agents are the autonomous components, communicating through online message passing, serving the application layer. A front end GUI allows business analysts to query and maintain business knowledge anywhere, anytime. Each layer can evolve independently. The maintenance of rules brings adaptivity to the whole complex application.

5. Contributions

In this section we will consider the benefits arising from applying the ARC framework.

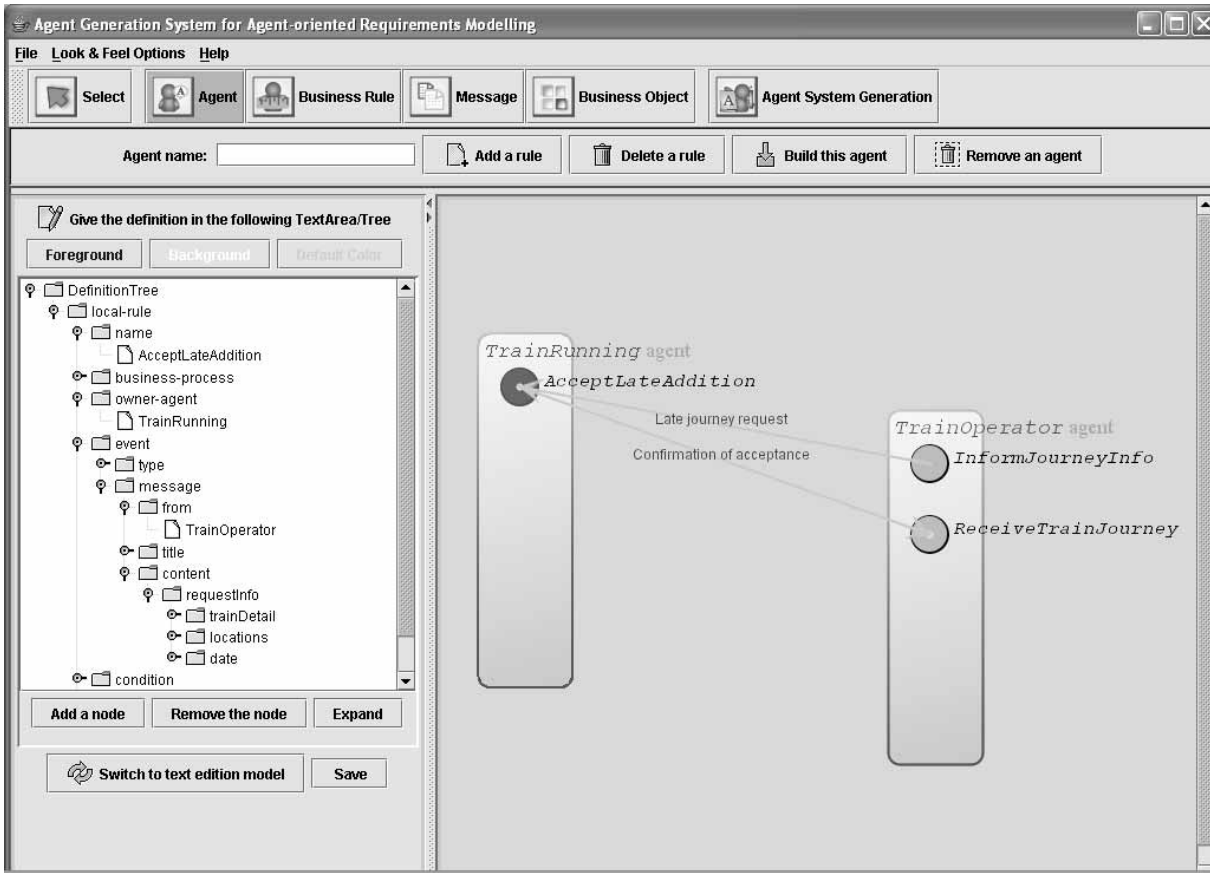


Fig. 12. Supporting tool for ARC framework.

5.1. Adaptivity

We have already described how the ARC framework introduces another layer of abstraction over traditional OO components and achieves full architecture independence. As a consequence, business rules, sitting between agents and business classes, are used to manage system collaboration and become the knowledge source of agent behaviours.

The most attractive outcome of applying the ARC framework to agent system development is the level of adaptivity achieved. As shown in Fig. 15, a collection of {condition, action} pairs can be set for Rule1 specifying, on receipt of an external event such as a request for service, the different actions in different conditions. Several re-configurations can be used for different adaptive purposes.

1. Re-matching of conditions and actions can change the action for a condition to another pre-set action. Supposing condition1 is satisfied initially, the exchange of action1 and action2 for

condition1 and condition2 can make Rule1 to invoke Class2 and Class3 instead of Class1 and Class2, and send the result to Agent3 instead of Agent2.

2. Calling alternative business classes in actions determinately affects the produced result. Supposing method1 of Class1 is invoked by action1, and then another version of Class1 is available with method1 updated. Re-configuring action1 to use the new version of Class1 can bring a new behaviour and produce a different result, without amending code.
3. Re-ordering of the conditions can change the priorities for applying the corresponding actions, if conditions are not all exclusive. If both condition1 and condition2 are satisfied, then the exchange of them in the Rule1 specification can make Agent1 execute action2 instead of action1.

Therefore, by using adaptive rules, agents are both adaptive internally because they can make use of different business classes and also externally because they

```

thisAgent.addBehaviour (Rule thisRule) {
    thisBehaviour.setPriority (thisRule.getPriority ());
    TrainJourney trainJourney;
    Message m = thisAgent.receiveMessage ();
    while (m != null){
        Agent fromAgent = m.getSenderAgent ();
        if (fromAgent.equals (thisRule.getEvent ().getMessage ().getFromAgent ()))
        {
            /* the rule is applicable to the received message */
            RequestInfo requestInfo = (RequestInfo) m.getContentObject ();
            trainJourney = new TrainJourney (requestInfo);
            if (trainJourney.ValidateTrainPlan ()) {
                /* the condition of the rule is satisfied */
                Message m2 = new Message ();
                m2.setContentObject (trainJourney);
                Agent toAgent = thisRule.getAction ().getMessage ().getToAgent ();
                m2.addReceiverAgent (toAgent);
                thisAgent.send (m2);
                /* update this agent s beliefs */
                thisAgent.addBelief (System.currentTimeMillis (), fromAgent, m);
            }
        }
        m = thisAgent.receiveMessage ();
    }
}
    
```

Fig. 13. Pseudo code of an agent behaviour.

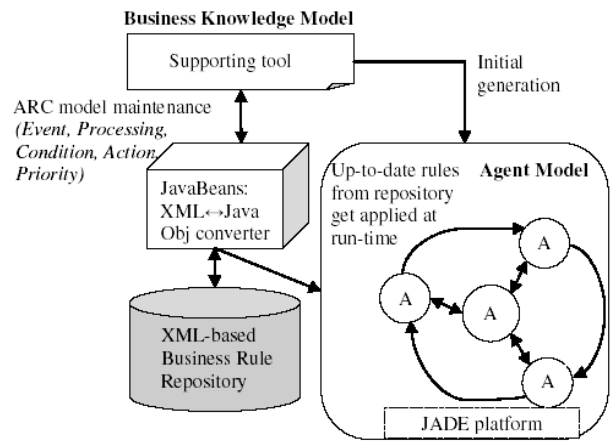


Fig. 14. Deployment of agent systems based on ARC framework.

can speak to different collaborators differently. This provides intra-agent and inter-agent adaptivity, more details being found in [38]. Moreover, all the settings

for adaptivity can be text-based or configured by business people through a simple user interface provided by the supporting tool. Major software maintenance after

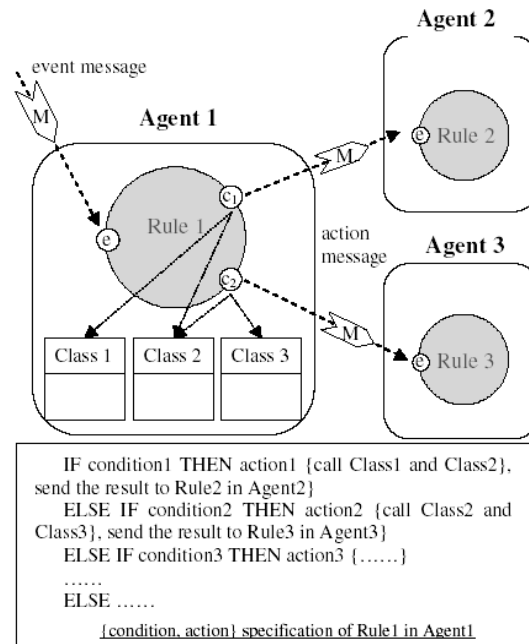


Fig. 15. Using ARC framework for adaptivity.

the deployment of systems is predicted to be focused on: i) Definition of new rules and re-definition of existing ones. ii) Changing of rules assignment to agents. iii) Changing of use of classes and their methods in rules.

One of the other uses of ARC framework is that it can play the role of middleware. Supposing Class 1, Class 2, and Class 3 used by Agent 1 are developed in Java and a set of classes used by Agent 2 and Agent 3 are developed in C. The cross platform agent layer enables the inter-operation between component systems written in different programming languages, as the communication among agents is via XML messages (This is a common requirement of distributed software architectures like CORBA).

Additionally, some agents can be dedicated for modularising crosscutting concerns. For example, an agent with a set of authorisation rules, an agent with a set of security rules and an agent with a set of logging rules can be developed. They are ready to listen to the event messages requesting such services and reply with the results, such as granting authorised access, read/write operation permission or logging of events. The separated central modules of these can minimise code tangling and code scattering. The use by demand feature of these by ordinary agents makes the evolution of them the evolution of crosscutting concerns in the whole system (This meets an aspiration of aspect-oriented programming).

5.2. A new development process model

One of the challenges facing the traditional software development process for MAS is the need for a new efficient process model, suited to the abstraction of agent-based computing [43]. The ARC framework introduces an efficient parallel development and maintenance process for MAS in its hierarchy of Agent-Rule-Class, as shown in Figure 16. In three separate layers a system is developed and maintained by actors playing three types of role, each specialising in one of the layers. i) The AO expert role is concerned with strategic selection of an agent running platform appropriate for their specific domain, such as Jade [16]. This means that systems built using the ARC framework are not limited to a particular platform, since all platforms comply with the FIPA [10] specification and are exchangeable with no effect to the systems running on any specific platform. The currently used platform may be discarded in favour of another with additional features and better performance. ii) The business analyst role concerns those familiar with the domain and relates to the development and maintenance of business models, including organisational structures, policies, and ontologies. Since the upper layer agents and lower level objects run independently from the business rules but reflect the changed requirements captured in rules in system behaviours. This can be done without interven-

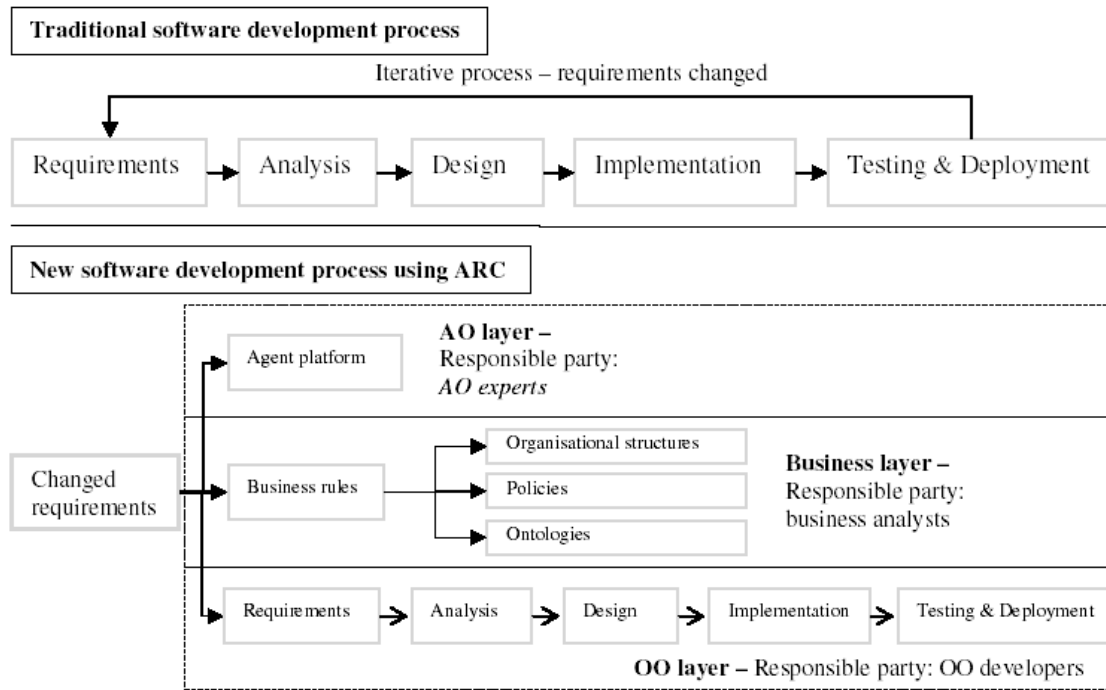


Fig. 16. Parallel development and maintenance of MAS based on ARC framework boosts separation of concerns and efficiency.

tion by developers but by customers directly, requiring no system re-development or re-deployment. This also advances a model driven architecture approach. iii) Once OO developers have implemented a layer of supporting components, the need for maintaining the individual atomic functions is comparatively low, as they capture the most fundamental operations and it is up to the business rules that dynamically use them. Alternatively, these OO components could be COTS bought from third parties and the updating of them is up to the component providers.

5.3. Summary

As discussed earlier, current AOSE research is largely on analysis and design of MAS. Methodologies covering the whole development process along with practical tools to support traditional OO developers to build MAS are not common, especially those that reuse existing knowledge and components. As a result, there is little, if any, work on convenient maintenance on the basis of re-configurable use of objects by agents. The ARC framework complements these shortages, supplying a complete AOSE development process model. Agent-oriented requirements engineering transforms requirements specification into organ-

isational agents, rules being specific requirements related with agents. Structural Model and Behavioural Model provide ARC design models in an extended UML style. Represented in XML and ready to be interpreted by agents, rules providing agent behaviours and the implementation model are being established. Maintenance is supported by a tool for both the blueprint of the whole system diagrammatically and also specific rules textually. In defining or evaluating an AOSE methodology the match with the agent paradigm should be considered, but also how it facilitates software development and maintenance [43]. This position supports the need for a comprehensive process, such as that using the ARC.

Accordingly, the ARC framework contributes to both AOSE and OOSE in three major aspects. i) It allows developers who are familiar with OO system development but new to AO to better understand MAS and their development with the possibility and advantages of adding an abstraction layer over the existing OO components. ii) It guides a full software development process and provides supporting tools for AO system development. iii) Because agents and objects are integrated for use/reuse in ARC framework, which leads to more efficient development and easier maintenance by using rules between them, both communities would be

more convinced of each other's approach (especially for OO proponents to accept AO).

The ARC framework provides a useful means of managing openness in MAS. This is crucial for the agent paradigm, where specifically there is a need at runtime to: accommodate new components in an environment that automatically knows and uses them; allow new and re-organised agent interaction patterns; allow new rules that drive the entire behavioural model of the system.

6. Related work

A useful approach that provides a language facility to support AOSE, called the *caste* [44–46], is closely related to the ARC framework. Both caste and ARC provide modularity and promote agent-orientation as an evolution of object-orientation. Notions of state, action, and behaviour rule are used to describe agent characteristics and agents of the same structural and behavioural characteristics are defined as caste. The approach is useful to define inheritance and instance relationships in MAS, an advancement over simple agent group relationships. Also agents can join or quit castes freely at runtime. Agent classification and characteristics thus can be dynamically changed, whereas objects are instances of same classes at all time. However, the approach has not developed a mechanism to allow agents to make use of existing OO components. In contrast, in the ARC framework, behaviour rules for agents can select current available actions from encapsulated class methods, which allow the MAS to be immediately executable. In addition, in the ARC framework agent interactions are explicitly modelled in the Behavioural Model. This and the associated supporting tool provide a blueprint of the system to be developed and a means to validate its completeness and consistency. Such models and utilities are not offered by caste which instead requires behaviour rules to have been modelled individually for inter-agent communications. This may be difficult and error-prone for system modellers.

A major feature that distinguishes ARC framework from caste and other AOSE approaches, however, is the context. In our opinion, it is more appropriate for an AOSE approach to be considered in the beginning in a context of individual applications, and later the inter-operation of multiple applications in a more complex context, existing technologies being extended wherever possible to support in both contexts. The requirements of each application is already specified from cus-

tomers input and building an executable MAS on top of OO components brings additional advantages such as adaptivity. In contrast, many other AOSE methodologies assume a distributed environment and attempt to solve a few specific issues using the agent technology. AOSE is certainly promising for distributed system engineering due to its capabilities for dealing with complex issues [43]. This is related with the autonomous, pro-active, and goal-oriented characteristics of agents. However, critical problems may arise in many standard applications when AOSE methodologies are applied as it is not clear whether or when a given task can be accomplished in MAS. Major traditional specifications required in each of the customer sites for the stable running of core business might be overlooked or not carried out using an assured implementation, autonomous agents being problematic. The MAS built therefore may not satisfy all of the functions desired by different stakeholders and, in the long term this will hinder the adoption of MAS. In addition, extra cost may also be introduced unnecessarily to the development since a high percentage of the requirements are found in already developed OO components or can be easily provided using existing technologies. In many cases, a more viable solution is to employ and accept a solution which combines the existing reusable components and adds agents on top wherever necessary in applications, and then after the mandatory functions are delivered, additional benefits can be enjoyed, due to the inter-operations of multiple MAS.

For these reasons, in the ARC framework we start from determined agent behaviours restricted by business requirements. Unlike continuously un-predictable MAS arising from many other AOSE methodologies, the MAS built using the ARC framework is defined and visible at any given time, even though the behaviours are changing and unknown in the long term. Using the ARC framework, autonomy is actually limited in agents in order to gain higher visibility and certainty of the system requirements. This is necessary and essential in many systems, especially if real-time processing and safety is of important. An infamous example is found in the crash of Air France's Airbus 320 at an air show in 1988, which was caused by a conflict between the human pilot's instruction and the autonomous control by software agents [44]. In ARC based systems, agent behaviours are adaptable, visible and guaranteed at a given moment. Nonetheless, agents in the ARC framework can be considered semi-autonomic, in that the rules which offer agent knowledge and define agent behaviours can be freely changed and also agents can

choose rules, both of these providing dynamic system behaviours. Nevertheless, in the future development of ARC framework, the option of more autonomy will be introduced but with the proviso that consistency with the global properties of the system is maintained. The intention is to provide intelligent selection of favourable rules and rejection of undesirable ones. This will be necessary for scaling when higher complexity is encountered. Three scales of observation: micro, macro, and meso, are proposed in [43] for analysis of MAS engineering issues. At present the ARC framework is closer to the micro scale, in which a limited number of agents of reliable and understandable behaviours are under control. At the macro scale, a large number of agents run in an uncontrollable environment with only the overall behaviour being important. Therefore, extension of ARC towards the macro scale is a future research direction for additional benefits.

7. Conclusions

The definition of the ARC framework is given in this paper. The application of it is especially beneficial in three areas: i) Building agent systems on top of OO systems using the ARC framework achieves adaptivity by configuring host system components collaboration; ii) Inter-operability is achieved by enabling the collaboration between cross platform components that are written in different languages, in a distributed environment; iii) Easy maintenance is facilitated by separating crosscutting concerns in dedicated agents.

The current work focusing on the contribution of adaptivity shows promise, the achievement of which lies in using business rules between agents and classes, so that, original classes are kept intact most of the time as stable components. An easy way is provided to configure rules, which make use of classes differently according to the configuration. Agents are empowered to behave according to rules specification at run-time. Because only at the time when rules get applied agents know their collaborated agents and classes and then they are bound at that time, flexible system behaviours are achieved through runtime agent-agent and agent-class re-binding by re-configuration of rules. It has been demonstrated in our recent work the suitability and usefulness of ARC for constructing adaptive requirements models [34], design models [35], and implementation [33,37], particularly for agent-oriented systems [36,38]. Since the benefits of the OO paradigm are not abandoned and the features of CORBA and

aspect-oriented programming have been introduced via an extra layer, being agents coupled with knowledge in business rules, the potential application of ARC is extensive.

The ARC framework will be made more powerful, but work so far indicates that it will contribute in a novel and substantive way to adaptive agent-oriented systems.

References

- [1] L. Andrade and J.L. Fiadeiro, *An Architectural Approach to Auto-Adaptive Systems*, Proceedings of the 22nd International Conference on Distributed Computing Systems Workshops, 2002, 439–444.
- [2] 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, 11–18.
- [3] AUML web site, <http://www.auml.org/>.
- [4] B. Bauer, J.P. Muller and J. Odell, Agent UML: A Formalism for Specifying Multiagent Software Systems, *International Journal of Software Engineering and Knowledge Engineering* **11**(3) (2001), 207–230.
- [5] F. Bergenti, M. Gleizes and F. Zambonelli, *Methodologies and Software Engineering for Agent Systems: The Agent-Oriented Software Engineering Handbook*, Kluwer, 2004.
- [6] J. Castro, M. Kolp and J. Mylopoulos, Towards Requirements-Driven Information Systems Engineering: The Tropos Project, *Information Systems* **27**(6) (2002), 365–389.
- [7] 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.
- [8] D. Damian, C. Jonker, J. Treur and N. Wijngaards, Integration of behavioural requirements specification within compositional knowledge engineering, *Knowledge-Based Systems* **18**(7) (2005), 353–365.
- [9] S.A. DeLoach, M.F. Wood and C.H. Sparkman, Multiagent Systems Engineering, *International Journal on Software Engineering and Knowledge Engineering* **11**(3) (2001), 231–258.
- [10] Foundation for Intelligent Physical Agents, <http://www.fipa.org/>.
- [11] M. Fowler, *UML Distilled: A Brief Guide to the Standard Object Modeling Language*, 3rd Edition, Addison-Wesley, 2004.
- [12] E. Gamma, H. Richard, R. Johnson and J. Vlissides, *Design Patterns*, Addison-Wesley, 1994.
- [13] T.W. Goh and Z. Zhang, An intelligent and adaptive modelling and configuration approach to manufacturing systems control, *Journal of Materials Processing Technology* **139**(1–3) (2003), 103–109.
- [14] M. Griss, S. Fonseca, D. Cowan and R. Kessler, *SmartAgent: Extending the JADE Agent Behavior Model*, Proceedings of SEMAS, 2002.
- [15] The Institute of Electrical and Electronics Engineers, IEEE recommended practice for software requirements specifications, IEEE Std 830-1998, 1998.
- [16] JADE platform, <http://sharon.csel.it/projects/jade/>.
- [17] Z.H. Jia, K.S. Ong, J.Y.H. Fuh, F.Y. Zhang and A.Y.C. Nee, An adaptive and upgradeable agent-based system for coordinated product development and manufacture, *Robotics and Computer-Integrated Manufacturing* **20**(2) (2004), 79–90.

- [18] D. Kinny, M. Georgeff and A. Rao, A methodology and modelling technique for systems of BDI agents, in: *Agents Breaking Away: Proceedings of the Seventh European Workshop on Modelling Autonomous Agents in a Multi-Agent World, MAA-MAW'96*, W. Velde and J. Perram eds, LNAI 1038, Springer, 1996, pp. 56–71.
- [19] A. Kleppe, J. Warmer and W. Bast, *MDA Explained: The Model Driven Architecture: Practice and Promise*, Addison-Wesley, 2003.
- [20] G.B. Laleci, Y. Kabak, A. Dogac, I. Cingil, S. Kirbas, A. Yildiz, S. Sinir, O. Ozdakis and O. Ozturk, *A Platform for Agent Behavior Design and Multi-Agent Orchestration*, Agent-Oriented Software Engineering Workshop, the Third International Joint Conference on Autonomous Agents & Multi-Agent Systems, 2004, 205–220.
- [21] K. Lieberherr, *Workshop on Adaptable and Adaptive Software*, Proceedings of the Tenth Conference on Object Oriented Programming Systems Languages and Applications, 1995, 149–154.
- [22] 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, 114–124.
- [23] M. Manna, *Maintenance Burden Begging for a Remedy*, Data-mation, 1993, 53–63.
- [24] Object Management Group, Inc., Agent Technology Green Paper, OMG Document agent/00-09-01 Version 1.0, 250 First Ave, Suite 201, Needham, MA 02494, USA, 2000.
- [25] Object Management Group, Inc., Applying UML 2 to Model-Driven Architecture, 250 First Ave. Suite 100, Needham, MA 02494, USA, 2003.
- [26] Object Management Group, Inc., CORBA 3.0 – IDL Syntax and Semantics chapter, OMG Document formal/02-06-07, 250 First Ave., Needham, MA 02494, USA, 2002.
- [27] J. Odell, H. Parunak and B. Bauer, Representing Agent Interaction Protocols in UML, in: *Agent-Oriented Software Engineering*, P. Ciancarini and M. Wooldridge, eds, LNCS 1957, Springer, 2001, pp. 121–140.
- [28] M. Shaw and D. Garlan, *Software Architecture: Perspectives on an Emerging Discipline*, Prentice Hall, 1996.
- [29] V. Silva, R. Choren and C. Lucena, Using the MAS-ML to Model a Multi-Agent System, in: *Software Engineering for Multi-Agent Systems II*, C. Lucena et al., eds, SELMAS 2003, LNCS 2940, Springer, 2004, pp. 129–148.
- [30] I. Sommerville, *Software Engineering*, 7th Edition, Addison-Wesley, 2004.
- [31] G. Wagner, The Agent-Object-Relationship Metamodel: Towards a Unified View of State and Behavior, *Information Systems* 28(5) (2003), 475–504.
- [32] M. Wooldridge, N.R. Jennings and D. Kinny, The Gaia Methodology for Agent-Oriented Analysis and Design, *Journal of Autonomous Agents and Multi-Agent Systems* 3(3) (2000), 285–312.
- [33] L. Xiao and D. Greer, *The Adaptive Agent Model: Software Adaptivity through Dynamic Agents and XML-based Business Rules*, Proceedings of the Seventeenth International Conference on Software Engineering and Knowledge Engineering (SEKE'05), Taipei, Taiwan, Republic of China, 14–16 July, 2005, 62–67.
- [34] 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), Paris, France, 29–30 August, 2005, 28–37.
- In conjunction with the Thirteenth IEEE Requirements Engineering Conference (RE'05).
- [35] L. Xiao and D. Greer, *Modelling Agent Knowledge with Business Rules*, Proceedings of the Seventeenth International Conference on Software Engineering and Knowledge Engineering (SEKE'05), Taipei, Taiwan, Republic of China, 14–16 July, 2005, 566–571.
- [36] L. Xiao and D. Greer, *Modeling, Auto-generation and Adaptation of Multi-Agent Systems*, Proceedings of the Tenth CAiSE/IFIP8.1 International Workshop on Exploring Modeling Methods in Systems Analysis and Design (EMMSAD'05), Porto, Portugal, 13-14 June, 2005, 605–616. In conjunction with the Seventeenth Conference on Advanced Information Systems Engineering (CAiSE'05).
- [37] L. Xiao and D. Greer, *Software Adaptivity through XML-based Business Rules and Agents*, Proceedings of the PREP2005, Lancaster, UK, 30th March-1st April, 2005, 287-288.
- [38] L. Xiao and D. Greer, Externalisation and Adaptation of Multi-Agent System Behaviour, in: *Advanced Topics in Database Research*, (Chapter IX, Volume 5), K. Siau, eds, Idea Group, Inc., 2006, pp. 148–169.
- [39] S.H. Yim, J.H. Ahn, W.J. Kim and J.S. Park, Agent-based adaptive travel planning system in peak seasons, *Expert Systems with Applications* 27(20) (2004), 211–222.
- [40] J.W. Yoder, F. Balaguer and R. Johnson, Architecture and Design of Adaptive Object-Models, *ACM Sigplan Notices* 36(12) (2001), 50–60.
- [41] J.W. Yoder and R. Johnson, *The Adaptive Object-Model Architectural Style*, Proceedings of the 3rd IEEE/IFIP Conference on Software Architecture: System Design, Development and Maintenance, 2002, 3–27.
- [42] F. Zambonelli, N.R. Jennings and M.J. Wooldridge, Developing Multiagent Systems: the Gaia Methodology, *ACM Transactions on Software Engineering and Methodology* 12(3) (2003), 317–370.
- [43] F. Zambonelli and A. Omicini, Challenges and research directions in agent-oriented software engineering, *Autonomous Agents and Multi-Agent Systems* 9(3) (2004), 253–283.
- [44] H. Zhu, SLABS: A Formal Specification Language for Agent-Based Systems, *International Journal of Software Engineering and Knowledge Engineering* 11(5) (2001), 529–558.
- [45] H. Zhu, *The role of caste in formal specification of MAS*, Proc. of PRIMA'2001, LNCS 2132, Springer, 2001, 1–15.
- [46] H. Zhu and D. Lightfoot, Caste: A step beyond object orientation, in: *Modular Programming Languages – Proc. of JMLC'2003*, L. Boszormenyi and P. Schojer, eds, LNCS 2789, Springer, 2003, pp. 59–62.

Authors' Bios

Liang Xiao is a research fellow with the Intelligence, Agents, Multimedia Group in the School of Electronics and Computer Science at the University of Southampton. He obtained his BSc at the Huazhong University of Science & Technology (HUST), 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. His experience on solving real domain problems has stimulated his interests in the area of Software Engineering. Specifici-

cally, his research work focuses on Software Adaptivity, Agent-Oriented Modelling and Requirements Engineering. The results of his research have been presented and published at several international conferences.

Des Greer is a lecturer in Computer Science at Queen's University, Belfast. He is a graduate of Queen's University, Belfast and earned a masters and doctorate at the University of Ulster. Before his ca-

reer 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 adaptivity, iterative and incremental software processes, software evolution planning and software risk management.