**European Journal of Information Systems (2007) 00**, 1–17

www.palgrave-journals.com/ejis

# Towards agent-oriented model-driven architecture

Liang Xiao[1] and
Des Greer[2]

*[1]School of Electronics & Computer Science, University of Southampton, U.K.; [2]School of Electronics, Electrical Engineering & Computer Science, Queen's University Belfast, U.K.*

**Correspondence: Liang Xiao, School of Electronics & Computer Science, University of Southampton, Southampton, SO17 1BJ, U.K.**
**E-mail: lx@ecs.soton.ac.uk**

**Abstract**
Model-Driven Architecture (MDA) supports the transformation from reusable models to executable software. Business representations, however, cannot be fully and explicitly represented in such models for direct transformation into running systems. Thus, once business needs change, the language abstractions used by MDA (e.g. object constraint language/action semantics), being low level, have to be edited directly. We therefore describe an agent-oriented MDA (AMDA) that uses a set of business models under continuous maintenance by business people, reflecting the current business needs and being associated with adaptive agents that interpret the captured knowledge to behave dynamically. Three contributions of the AMDA approach are identified: (1) to Agent-oriented Software Engineering, a method of building adaptive Multi-Agent Systems; (2) to MDA, a means of abstracting high-level business-oriented models to align executable systems with their requirements at runtime; (3) to distributed systems, the interoperability of disparate components and services via the agent abstraction.
*European Journal of Information Systems* (2007) **0,** 000–000.
doi:10.1057/palgrave.ejis.3000688

**Keywords:** adaptive agent model; agent-oriented model-driven architecture; agent-oriented software engineering; business knowledge model; model-driven architecture; multi-agent system; requirements engineering; software adaptivity; UML



## Introduction and background

Business environments and business needs are often changing rapidly. Progressive change and adaptation of the supporting software systems is inevitable but the maintenance and evolution of traditional OO systems is difficult because (1) Objects inherently have static structure and behaviour. (2) Object-oriented requirements and design models in the form of UML diagrams lack the capability to describe behavioural semantics (Fowler, 2004), for which reason implemented systems cannot be directly transformed from models and so they rapidly lose their value, if maintenance changes are done at the code level only.

The Object Management Group's (OMG) Model-Driven Architecture (MDA) (Fowler, 2004; Object Management Group; Kleppe *et al.*, 2003; Meservy & Fenstermacher, 2005; France *et al.*, 2006) promotes the production of models with sufficient detail that they can be used to generate executable software (Mellor & Balcer, 2002). MDA proposes a Platform Independent Model (PIM), a highly abstracted model, independent of any implementation technology. This is translated to one or more Platform Specific Models (PSM), which in turn are translated into code.

Executable UML (Mellor & Balcer, 2002) that relies on Action Semantics (AS) (Object Management Group, 2002b) or Object Constraint Language (OCL) (Object Management Group) for specification of actions makes such

**Received: 31 November 2006**
**Revised: 25 April 2007**
**Accepted: 27 July 2007**

model transformation possible. However, these action languages do not significantly raise the level of abstraction above that provided by programming languages (France *et al.*, 2006). The understanding of both a semantic-oriented language and a programming language is required, and the interpretation of the statements in OCL/AS into specific language constructs is manual, error-prone and effort intensive.

Moreover, these language statements represent constraints for design decisions rather than capturing business rules at the requirements level. Business representations, such as business rules, if not specified explicitly as business level constraints but hard-coded in systems would add to the maintenance burden (Xiao & Greer, 2005). Capturing business requirements in model abstractions and then transforming them into running software systems is not possible by using object-oriented MDA. Further, whenever code relevant to business needs has to be changed, running systems must be interrupted, regenerated and redeployed.

Agents have been credited as an advance in Software Engineering abstractions (Wooldridge *et al.*, 1999). In general, agents are reactive and proactive and dynamically perform actions to achieve their goals. 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. Coupled with knowledge models, agents have the potential to dynamically adapt their behaviour at runtime when models are changed (Xiao & Greer, 2006a).

However, in current practice, Interaction Protocols (IPs) (Foundation for Intelligent Physical Agents; AUML web site) that model agent conversations have to be turned into program code by developers, manually (Ehrler & Cranefield, 2004). Agents cannot behave dynamically or be configured to do so when the development is completed. One approach, Plug-in for Agent UML Linking (PAUL) attempts to allow agents to execute IPs by attaching application-specific code to the appropriate points of the protocols (Ehrler & Cranefield, 2004). The problems here are that the use of separate code fragments making management and maintenance difficult, the lack of support for agents changing roles and the use of Java statements tying the method to a specific platform.

Mainstream agent-oriented methodologies such as *i** or Gaia concentrate on requirements or early design modelling. In these, strategic actor dependencies or roles are modelled, but cannot be transformed later to platform-specific models or running code, the gap between these methodologies and IP-based development on top of existing platforms inhibiting their full-scale usage.

An Agent-oriented Model-Driven Architecture (AMDA) is therefore put forward. AMDA uses a set of business models under continuous maintenance of business people to reflect the current business needs, models being associated with adaptive agents that interpret the captured knowledge to behave dynamically, always fulfilling current requirements. Consequently, the maintenance of the models is the maintenance of the actual software system. This provides a means of model-based adaptation rather than code-based adaptation.

The next section will describe the starting point of the AMDA approach: capturing requirements in a Computation Independent Model (CIM), making use of a case study. In the following section, we illustrate the building of a PIM, centred on its hierarchical knowledge models and the agent model. The penultimate section discusses the transformation of the PIM to a PSM as well as code for a specific agent development platform. The contributions of the AMDA to the existing body of knowledge are reviewed in the concluding section.

## Capturing requirements in a CIM

To demonstrate the efficacy of the AMDA approach, we have investigated how the AMDA approach might be applied to an actual system, a British railway management system. The system monitors train running with regard to incidents and also ensures the safety of the train services by conveying issues to relevant parties for resolution. Figure 1 presents an extract from the original 250+ page specification.

In the specification, a large number of standardised functional requirements tables have been documented for each domain. A specific requirement, *IMI–HandleFault*, is given in Table 1.

Figure 2, based on Figure 1, demonstrates the CIM for the handling of faults, the subsequent imposing of restrictions and the rescheduling of train services that would involve faulty assets.

Table 2 summarises the roles of major business domains and actors through the interaction of which faults are managed and rectification is made. Domain roles are collective functions such as *IMI–HandleFault* given in Table 1. This conceptualisation in the CIM directs the later modelling in PIM of agents as representing actors/domains and agents playing roles as defined by domain functions.

When object-oriented systems are being implemented, functional tasks shown in Figure 2 will be statically assigned to objects with fixed message passing patterns between objects. The new paradigm of AMDA raises the level of abstraction. For example, when an agent representing 'Train Operator' at the business level invokes a service 'Reschedule Train Service' in an interaction with another agent, that service will be able to be exchanged with alternative ones supplied by in-house developers or service providers for operation, externally, and configured for various use rather than functionally fixed, internally. This is enabled by the agent abstraction without requiring business experts to understand the underpinning objects or services implementation while configuring their business models.





PPL.EJIS.3000688

*Case background*: The specification comprises 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 or domains are closely linked. For example, an infrastructure fault (Infrastructure Management) may block the access to track and cause rescheduling of train services (Train Running).

Briefly, Train Running Business domain supports the principal service to customers, including delivery of planned train paths and response to requests for further train paths. Relating 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 as a result of a request from a customer (re-planning). Railway asset faults/incident will cause train service re-planning as part of the case study. Although the running of train services itself is not within the scope of the case study of this thesis, it has been studied thoroughly in [14].

The selected excerpt of the specification is concerned about fault management of the railway system. Involved domains are: Infrastructure Management - Incident (abbreviated **IMI**), being responsible for passing of information about faults between the system and contractors; Infrastructure Management – Execution (abbreviated **IME**), being responsible for granting of isolations; Train Running - Incident (abbreviated **TRI**), being responsible for refinement and corrections of planned train journeys. External entities with respect to fault management are: **Train Operators**, who initiate train running requests, and have to be consulted when dealing with perturbations, and **Contractors**, who carry out maintenance.

*Case terminology*: The infrastructure of the railway system consists of the assets necessary to run the trains. Their condition is a major constraint on train running. An *infrastructure asset* is any identifiable item of interest within the infrastructure. Examples of assets are points, bridges, or electrification equipment. An infrastructure asset may have a number of asset *faults*. Asset faults may either cause an *incident* or may be caused by an incident. Asset faults may also occur independently of an incident. Examples of incidents are accidental damage, spills, or faults themselves. An incident may cause a *track restriction*. For example, a broken rail may cause a line blockage. The condition of an infrastructure asset may also cause a track restriction. For example, deterioration in track quality may cause a temporary speed restriction. Track restrictions include isolations, temporary speed restrictions, line blockages, and reduced loading gauge. Under a *contract* or a variation to a contract with a contractor, infrastructure assets are maintained and asset faults are fixed.

*Case description*: An asset fault is either reported to the system (Requirement: *IMI-AcceptFaultReport*) or detected directly by the system (Requirement: *IMI-NoticeFault*). The handling of both cases is the same (Requirement: *IMI-HandleFault*). If the fault has already been cleared no further action is needed immediately. Otherwise the system notifies the Contractor responsible for the fault and agrees a priority for fixing the fault. The fault may not require immediate attention and may have no immediate impact, in which case nothing further is done. *However, if the fault is located at capital cities, it has impact and needs to be fixed immediately*. If the fault does have some impact an incident is recorded. It may be necessary to put in place immediate track restrictions (Requirement: *IME-ImposeSuddenRestrictions*), and this will involve changes to forecast train journeys (Requirement: *TRI-RespondToIncident*). Affected train journeys are amended for re-scheduled services to the Train Operator. Those concerned may be notified of the details (*Requirement: CCI-NotifyIncident*). As time passes or work progresses, further information may be received about the fault (Requirement: *IMI-UpdateFaultInformation*). This may result in changes to the priority of the fault or imposition or removal of track restrictions. A special case of this is the final fixing of the fault, when the restrictions will be removed.



**Figure 1** Extract from a railway management system specification.

## Creating a PIM

Related requirements are structured in domains and are delegated to agents, who have knowledge concerned with their corresponding domains, becoming actors that are responsible for realising domain functions and collaborating with each other by message passing for cross-domain interactions. The domain requirements, already captured in the CIM, must be organised in the PIM, in which responsibilities are assignable to conceptual agents and later transformable to the PSM, which



PPL.EJIS.3000688

**Table 1 Functional requirements table *IMI–HandleFault***

| *Domain* | *IMI* |
|---|---|
| Identifier | HandleFault. |
| Description | To maintain information about faults so that they can be fixed in a way that minimises the overall impact on the business. |
| Cause | A fault becomes known to the Production Function, either from people reporting information about a fault (*IMI–AcceptFaultReport*), or directly from the infrastructure asset, via infrastructure monitoring equipment or from failure to operate when commanded (*IMI–NoticeFault*). |
| Information | Information about infrastructure assets and their contracts. |
| Used | Information about *train journeys* to assess the impact of *the fault.* |
| Outputs | Fault information to *contractors.* |
| Required Effect | The fault is recorded.<br>Unless the fault has already been cleared, the appropriate contractor is identified and agreement is reached about a priority for fixing the fault.<br>If the fault is associated with an existing *incident* then that is recorded; otherwise, if it has some impact then a new incident is established with the fault as its cause.<br>If necessary, *track restrictions* are put in place. If so, there is an impact on the train service handled by *TRI–Respond-ToIncident.*<br>Anyone affected by the fault is notified (*CCI–NotifyIncident*). |

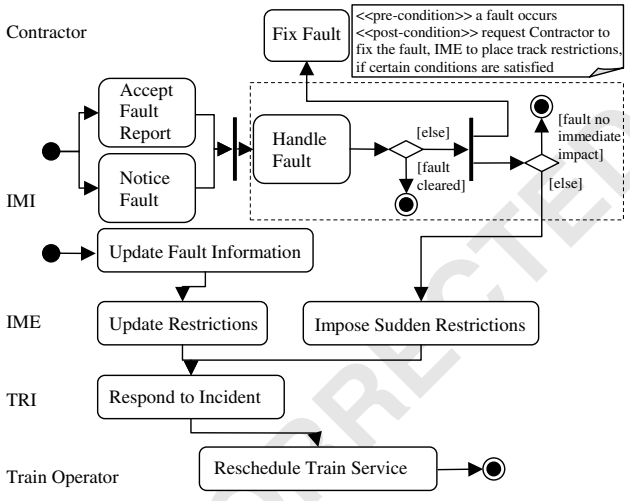


**Figure 2** The CIM for the case study.

**Table 2 Roles of actors and domains**

| *Actor/Domain* | *Role* |
|---|---|
| IMI | Detect and handle asset faults. |
| IME | Place track restrictions. |
| TRI | Handle the impact of incidents on train journeys by the amendment of those affected. |
| Contractor | Fix asset faults. |
| Train Operator | Reschedule train services. |

agents use to dynamically interpret (rather than being hard-coded with) their behaviour while running upon specific platforms. The PIM of AMDA consists of hierarchical business knowledge models and associated with them, a platform-independent agent model.

Using an AMDA creates a knowledge-driven semi-autonomous agency. Agents use the knowledge models to dynamically interact with each other externally, as well as to compute and make decisions internally, supporting components that are helpful for agent behaviour also being instructed by the knowledge models to agents for their runtime invocation. As illustrated in Figure 3, two combined hierarchical structures are proposed: (i) the hierarchy of business knowledge models (Business Process Rules, Reaction & Policy Rules, Business Concepts & Facts) and (ii) the hierarchy of computing components (Agents, Classes/Services). In brief, business knowledge is accumulated from models at various levels to drive agents to behave in order to meet business needs. The agents dynamically select appropriate business objects or web services and use them to fulfil their responsibilities. Collectively, these hierarchies are developed from previous work on the Adaptive Agent Model (AAM) (Xiao & Greer, 2006b) and represent the PIM for AMDA.

The knowledge captured in the business models has two building blocks: (a) a *Concept Model* (CM) used for





PPL.EJIS.3000688

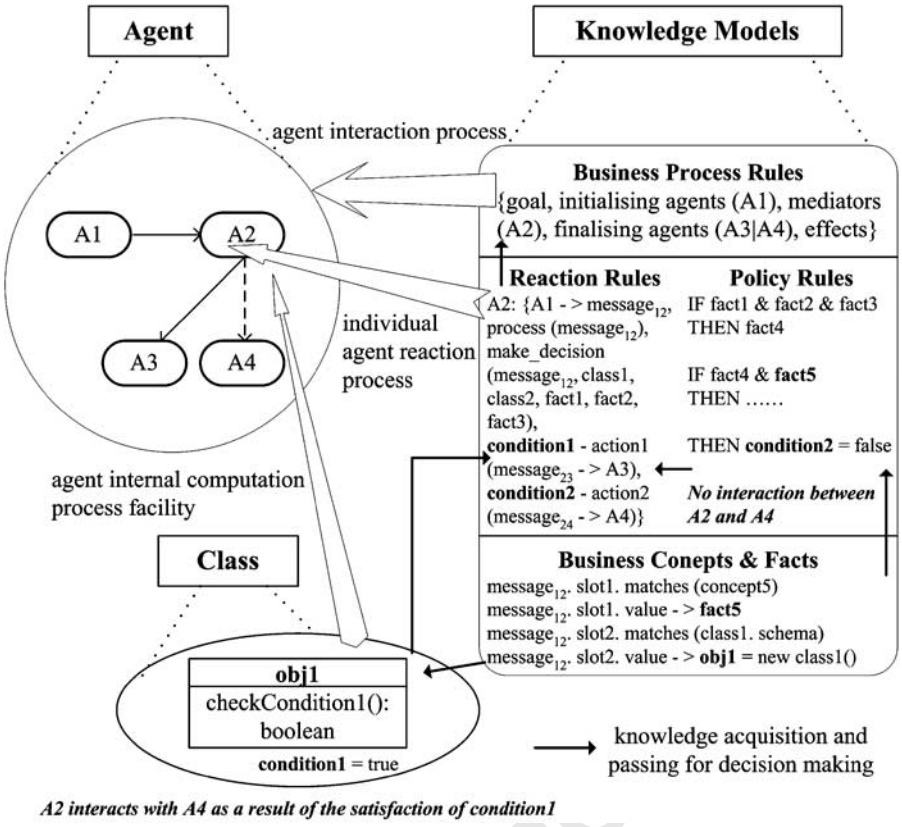


**Figure 3** The PIM overview demonstrating the hierarchical knowledge models and the agency.

vocabulary definition and referred to by all agents and (b) a *Fact Model* (FM) conforming to the CM, constructed at runtime according to a given agent's current knowledge. Two-rule models are in turn used to model agent behaviour. *Policy Rules* (PR) are global rules that all agents should obey and describe policies that must be enforced. *Reaction Rules* (RR) are local rules that agents should use individually and describe reactions that must be performed to other agents when triggered by external events, or messages passed in by some agents. *Business Process Rules* (BPR) realise business processes aimed at corresponding goals, through the interplay of multiple agents. A collection of PRs and RRs will be applied at various execution points in the bigger picture of a BPR, pulling together multiple local reaction processes and global policies. In the computing components hierarchy, objects and services support agent behaviour. The interaction of the two hierarchies realises business requirements. Principle elements in this scheme are as follows:

*Agent*: Agents are conceptual units that organise requirements in models and software units, driven by the models to realise assigned responsibilities.

*Business rule*: Rules are externalised functional requirements that capture agent behaviour and are configurable at runtime. A collection of rules compose and define agent interaction models.

*Class component and web service*: These are traditional passive components. They respond to active agents when invoked, as defined by rules. Our example will mainly use class components.

*Message*: A message is a string, XML portion, or object container passing between agents. For example, it can be defined in rules that an object will be encoded at the sending side, and that it will be decoded and recovered at the receiving side.

## CM and FM

Business concepts and their properties can be identified in the case study using a grammatical analysis. As an example, 'fault' has properties such as 'location', 'immediate impact' and 'priority', that is, fault {type [Enumeration], location [String], immediate_impact [Boolean], priority [Integer], description [String], cleared [Boolean]}.

Together, all business concepts, their constraints and concept–property relationships form the CM. Concrete facts are established at runtime with concept properties populated with values such as, a fault reported as taking place in 'London' and being of a classified type of 'rail broken', and so on, that is, fault {type (rail broken), location (London), …}



PPL.EJIS.3000688

Facts like this accumulate and are stored in the FM. One dedicated agent, the Fact Manager Agent (FMA) manages all facts and interacts with a Policy Rule Manager Agent (PRMA, detailed later in 'Policy rule' section) to add new deduced facts after the application of PR. For example, one policy may say that a fault occurring at London will have an immediate impact. The FM will get updated correspondingly, that is, fault {type (rail broken), location (London), immediate_impact (True), …}

The concept of 'fault' and its related properties can be represented in AMDA as XML, shown in Figure 4. A corresponding business class 'Fault' has later been implemented for agents to operate upon. Their instantiated business objects can be encoded in agent messages for their communication to announce new facts that will be made known by the FMA. By combining the power of PR application using existing facts and the computation of business class methods using existing property values, additional knowledge will be revealed. This supports agents to do decision making and guide their behaviour. New facts are accumulated and invalid ones demolished dynamically at runtime as a result of continuous message passing. Business models, as agent knowledge, are always up-to-date.

### Policy rule

Business policies naturally change over time, and thus externalisation of them as executable rules is desirable. PRs are typically embedded in specification descriptions. One sentence (underlined) in the case study reads: 'if the fault is located at capital cities, it has impact and needs to be fixed immediately'. This is represented in Figure 5. Relationships among business concepts are thus associated and constrained in values or logical relationships in a form that reflects the business needs or strategies. Such rules must be made explicit, as well as the representation of them in models. Otherwise, the embedment of them in code exacerbates the maintenance burden.

Suppose a 'fault' fact is initially established in the FM with its 'immediate_impact' unknown but 'location' known as 'London'. Being aware of the example of Rule1, the PRMA would apply this PR and then update the FM with the property 'immediate_impact' for the 'fault' set to 'true' through interaction with the FMA at runtime. This process can be iterative so that the application of one PR may trigger others, leading to additional facts being established, and eventually the formation of a chain of PRs. Rule2 (Figure 6) uses the term 'immediate impact' as

```
- <concept>
    <name> fault </name>
  - <properties>
      <property> type </property>
      <property> location </property>
      <property> immediate_impact </property>
      <property> priority </property>
      <property> description </property>
      <property> cleared </property>

      <!-- … more properties … -->

    <properties>
  </concept>
```

**Figure 4** Business concept 'fault' representation for the case study.

```
Rule1.
If fault is located at the capital cities
Then it has "immediate impact"
-------------------------------------------
- <policy>
    <id>100</id>
    <condition>
      fault.location == "London" OR "Edinburgh" OR "Cardiff" OR
"Belfast"
    </condition>
    <action>
      fault.immediate_impact = true
    </action>
    <priority>5</priority>
  </policy>
```

**Figure 5** PR representation for the case study.

```
Rule2.
If fault has "immediate impact"
Then it has a high priority
```

**Figure 6** Rule 1 triggers Rule 2 to function.





PPL.EJIS.3000688

its condition and the same term is defined in Rule1 as its consequent action. The execution of the Rule1 triggers the execution of the Rule2.

### Reaction rule

RRs define agreements that are bound between agents for their interactions, constraining what and how agents should perform in a reactive and proactive manner in business processes. Driven by events, agents use RRs to make business decisions. The left-hand side of Figure 7 represents the schematic decision-making tree. Each RR's decision-making element can be decomposed into multiple {condition, action} couplets, actions performed following conditions evaluated as satisfactory on the selected tree branch while decision making, supported by FMA and PRMA supplying facts. The right-hand side of Figure 7 is the decision-making tree that *IMI–HandleFault* uses to handle faults when they are reported. A branch will be selected as condition2→condition2.2 to request the fixing of fault and place track restrictions if corresponding conditions are met. If additional conditions need to be considered, the selected tree branch may be extended with: {condition2.2.1, action2.2.1}, {condition2.2.1.1, action2.2.1.1}, and so on.

This representation models in a platform-independent manner the part of the requirements that are captured in the dashed box area of the CIM shown in Figure 2. The PIM models with regard to agent decision making during interaction lose no generality from their ancestors and at the same time capture sufficient semantics to guide agent behaviour and support later model transformation to PSM.

Based on the tree structure, the scheme of a RR is defined as RR: {event, processing, $\{\text{condition, action}\}_n$, belief}. When an event message is received by an agent (Step 1), business objects are decoded from it and facts are then known to the recipient (Step 2). To respond, the agent makes a decision and performs actions that are associated with the satisfied conditions (Step 3). The result of the actions could be producing event messages to other agents (Step 4). The agent's beliefs are updated with the new information (Step 5). Figure 8 shows in natural language the procedure agent IMI uses to process its RR *IMI–HandleFault*.

A RR acts like a contract between agents. For example, *IMI–HandleFault*, as a fault handling RR in this fault management domain, will respond if and only if an event message with a pre-agreed information structure representing an asset 'fault' is received. In addition, it promises pre-agreed information structures will be sent to the

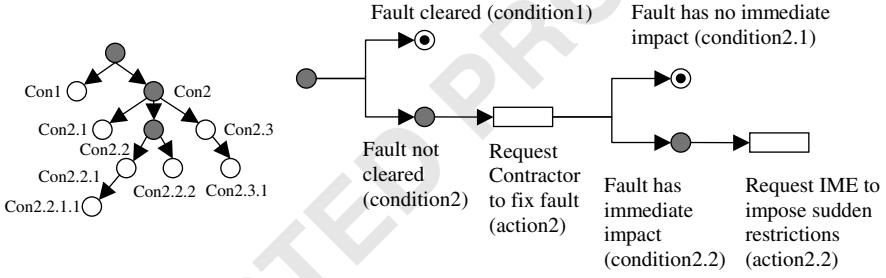


**Figure 7** Schematic decision-making tree of a RR and the tree structure of *IMI–HandleFault*.

| | |
|---|---|
| Step1: | Receive fault report message from "AcceptFaultReport" or "NoticeFault" from the same agent. |
| Step2: | Construct a "Fault" and an "Asset" object using the information contained in the message. |
| Step3&4 | {condition, action} couplet1: If the created "Fault" object is evaluated by its "cleared()" method as FALSE (Condition1), Then send a message with the created "Fault" to "FixFault" owned by Contractor agent (Action1), and |
| Step3&4 | {condition, action} couplet2: If the created "Fault" object is evaluated by its "immeImpact()" method as TRUE (Condition2), Then send a message with the created "Asset" to "ImposeSuddenRestrictions" owned by IME agent (Action2). |
| Step5: | Add the belief that a fault occurs at this moment with a potential incident related with it. |

**Figure 8** Brief steps of RR *IMI–HandleFault* processing by agent IMI.





PPL.EJIS.3000688

pre-agreed partners, as defined by the RR. A sample XML specification for this particular RR is shown in Figure 9. A guideline for transforming functional requirement tables (as in Table 1) to RR structures and then XML specifications is provided in Xiao & Greer (2006b). Each agent reacts to the receipt of a message by executing a rule using a process presented in Xiao & Greer (2006b).

The RR model is convenient for evolving software architecture through rule configuration. Compositional parts of rules separate computation (<processing>) from coordination (<event> and <action>). Like other

```
- <reaction>
    <name>HandleFault</name>
    <business-process>Fault Management</business-process>
    <owner-agent>IMI</owner-agent>
  - <global-variable>
    - <var>
        <name>asset</name>
        <type>Asset</type>
      </var>
    - <var>
        <name>fault</name>
        <type>Fault</type>
      </var>
    </global-variable>
  - <event>
    - <message>
        <from>IMI.AcceptFaultReport</from>
      - <content>
        - <report>
          - <reporter>Henry</reporter>
          - <fault>
              <type>rail_broken</type>
              <location>London</location>
            - <asset>
                <id>10015</id>
                <type>rail</type>
                <contractor>Contractor_A</contractor>
                ...
              </asset>
            </fault>
          </report>
        </content>
      </message>
    </event>
    <processing>
      asset = new Asset (reportMsg)
      fault = new Fault (reportMsg)
    </processing>

    <condition>
      fault.cleared () == false
    </condition>
  - <action>
    - <message>
        <to>Contractor.FixFault</to>
      - <content>
        - <fault>
            ...
          </fault>
        </content>
      </message>
    </action>

    <condition>
      fault.immeImpact () == true
    </condition>
  - <action>
    - <message>
        <to>IME.ImposeSuddenRestrictions</to>
      - <content>
        - <asset>
            ...
          </asset>
        </content>
      </message>
    </action>

    <priority>5</priority>
  </reaction>
```



**Figure 9** RR *IMI–HandleFault* specification for the case study.



PPL.EJIS.3000688

interface languages such as the OMG's Interface Definition Language (IDL) (Object Management Group, 2002a), RRs define the component communication interfaces via universal message travel over the network, regardless of their platform, operating system, programming language, and so on. The models set contracts for agent interaction, enabling the interoperation of agent systems across the network through a technology-independent interaction model. At the same time, the agent functions are abstracted away, the use of specific objects or services being continuously configured to provide the required functions but the interaction interfaces maintained.

### BPR model

The execution of collections of RRs, sequentially and conditionally following event message flow, forms business processes termed BPR. *IMI–HandleFault,* as discussed in the previous section, is a constituent of a BPR called 'Manage New Fault', which has the intention of handling new faults. This BPR is shown in Figure 10, with only the default conditions considered and assumed to be true for simplification.

The agent IMI initialises the BPR (called IAs) using either of its two RRs: 'IMI–AcceptFaultReport' or 'IMI–NoticeFault', in the interest of solving newly detected faults. The agents that finalise the BPR (called FAs) are Contractor and Train Operator, the completion of whose functions fulfils the goal of managing new faults, with faults fixed and train service rescheduled. Figure 11 illustrates this BPR.

### PIM knowledge models and semantic web

The CM and FM of PIM can be equally represented in RDF/RDFS with compatible semantic meanings and comparable expressiveness. The 'case terminology' section of the case study description can be structured in the RDF/RDFS-layered semantic net shown in Figure 12.

The following RDF description in Figure 13 expresses that a 'Fault' concept has a property of 'location', and so on, equivalent to the CM of PIM.

Figure 14 shows a concrete fault with 'London' as the value of its property 'location', equivalent to the FM of the PIM.

It is easy to document all concepts and facts in the PIM in established ontological languages, and AMDA does not place any constraint in that respect. However, they do not provide any further support other than expressing concepts, properties, their relationships and instances of these as facts. What is more important is abstracting higher level of knowledge on top of ontology that can be made use of to support computation, interaction and global business process. For example, although through the navigation of the graph we can infer a contract in association with an asset to which a fault is related, a responsible Contractor must be notified via some

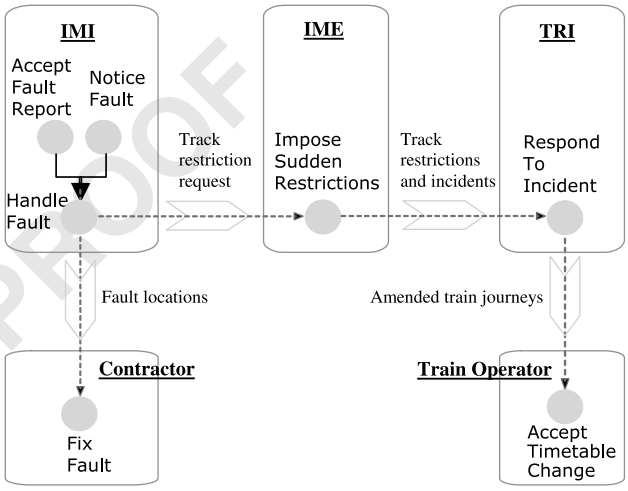


**Figure 10** BPR 'Manage New Fault' for the case study.

```
- <process>
    <name>Manage New Fault</name>
    <goal>a new fault is managed</goal>
  - <IAs>
      <IA>IMI</IA>
    </IAs>
  - <FAs>
      <FA>Contractor</FA>
      <FA>Train Operator</FA>
    </FAs>
    <cause>a new fault is reported</cause>
  - <effects>
      <effect>
        A Contractor will fix the fault
      </effect>
      <effect>
        Train Operator will re-schedule train services
      </effect>
    </effects>
  </process>
```

**Figure 11** XML representation of the BPR 'Manage New Fault'.





PPL.EJIS.3000688

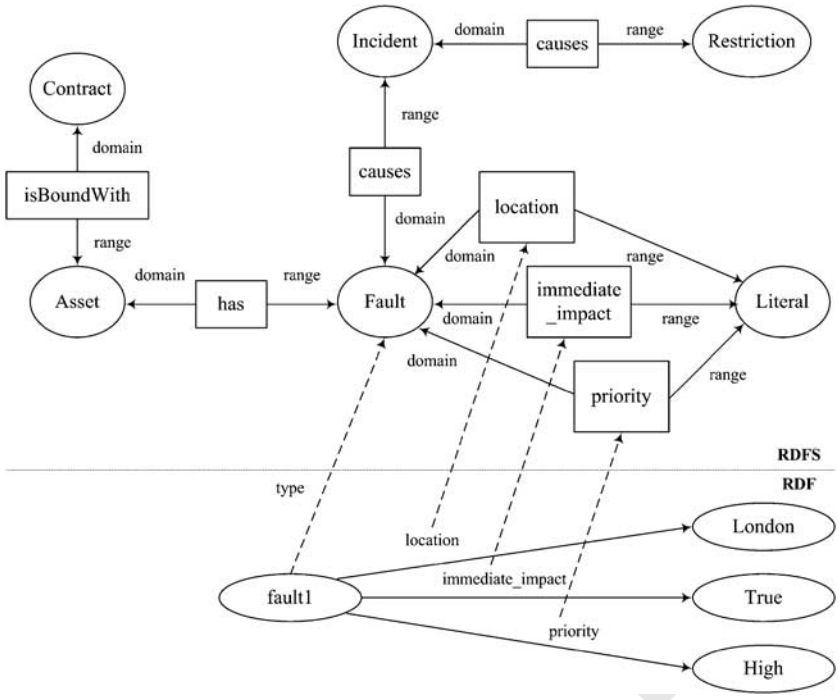


**Figure 12** RDFS and RDF CM for the case terminology.

```
<rdf:RDF……xmlns:PIM="……">
……
<rdfs:Class rdf:ID="Fault">
  <rdfs:comment>The class of faults</rdfs:comment>
</rdfs:Class>

<rdf:Property rdf:ID="location">
  <rdfs:domain rdf:resource="#Fault"/>
  <rdfs:range rdf:resource="&rdf;Literal"/>
</rdf:Property>
……
<rdf:RDF>
```

**Figure 13** RDF schema of 'Fault' and its properties.

```
<rdf:Description rdf:ID="fault1">
  <rdf:type rdf:resource="&PIM;Fault"/>
  <PIM:location>London</PIM:location>
</rdf:Description>
```

**Figure 14** RDF description of an instance of 'Fault'.

mechanism to come to fix the actual instance of the ontologically documented fault, and this is out of the scope of the CM and language expression. Further, the specific fault expression as shown in Figure 14 should be able to be updated automatically with the values of its two other properties, if the location is given as 'London', to reflect the policy that a fault occurs in the capital city has immediate impact and high priority. These requirements naturally point to the modelling of PR, RR, and BPR, as well the Agent Model in our PIM. Fortunately, research linking Agents and Semantic Web is growing in quantity (Bruns *et al.*, 2005) and likely to be of use to AMDA in the future.

### Agent model

Many specific agent platforms should be able to utilise the PIM knowledge models. An Agent Model is used as a vehicle that drives knowledge models interpretation by





PPL.EJIS.3000688

defining agent capabilities that need to be supported by a chosen platform. This not only maps knowledge models into agent behaviour in practice but also places minimum constraints on the agent implementation phase, thus allowing platform-independent modelling, as well as interoperation.

Three categories of class methods have been suggested for method design: query, mutation and helper (Riehle & Perry, 2002). In the context of agent-oriented systems, classification of agent roles/acts for runtime interpretation and execution of business models using runtime data as required by AMDA's PIM is shown in Table 3. A simple lexicon of agent acts, three falling into each one of the three categories, is used to specify agent behaviour. In spite of the straightforward mapping from these acts to OO-based programming statements (get, set, equal, if, and so on), the combination and composition of these fundamental acts make up of all required interactions among agents and business models, and manipulation of runtime data. Agents are the subjects that uses these acts to operate upon the business models and data, being the object. The separation of agents and business models means that changes in the externalised models are interpreted immediately using the semantics of these acts, rather than fixed code.

The combinational use of these acts by agents is flexible, decided when rules are dynamically retrieved as statements about the use of these acts performed on objects or concepts. Business knowledge is little by little known to agents on the fly, and they interact with each other to fulfil the current business needs. No specific requirements being set upon agent function, beyond these primitives, AMDA's PIM is technology independent.

### The PIM blueprint

The PIM blueprint shows business models combine to drive the behaviour of generic agents. In the rule hierarchy BPR–RR–PR, a BPR is formed by the execution of sequenced subordinate RR units, carried out by agents as primitive activities. In the course of each RR execution, PR chains are further applied in support of RRs for decision making. In this process, the model knowledge can be associated with agents developed from any platforms for execution, and the mapping of generic agent acts to specific platform constructs will only be considered in the next stage of AMDA.

The AMDA PIM is illustrated in Figure 15 for the case study. The upper BPR layer captures agents and their interplay towards the goal of fault management at a very high level. This resembles a UML collaboration or interaction diagram used in specifying a PIM for an OO system (analogously capturing objects and their relationships). The behaviour of agents, both agent interaction and internal computation and decision making, however, is not directly modelled such as method declaration in class diagrams in a PIM for OO systems but rather captured in the middle RR layer. These details guide individual agent behaviour. Two sources of knowledge are useful to support RR execution. Lower level class or service facilities can be invoked. Also a PR chain can be formed. Both can be involved for the RR computation and decision-making process. The bottom-level Business CM and FM are used for establishing new facts when informative events are announced and received by RR, and they support PR to deduce new facts, which will be useful to RR. Overall, the PIM model elements are all annotated using XML. Visualised PIM models can be reconfigured via tools support (Xiao & Greer, 2005). The XML annotations associated with the corresponding model elements contain all the model knowledge that agents can use to deploy up-to-date requirements knowledge on the fly, the precise XML-based specifications supplying semantic descriptions that traditional UML diagrams lack (Fowler, 2004) but present in the AMDA PIM. This is the distinct feature of AMDA, models being interpretable rather than requiring the generation of fixed behaviour code. The dynamic interpretation of agent behaviour from PIM will be illustrated in the next section via a PSM and code bridging.

Agent acts described in the abstract agent model can be associated with business models and runtime data for interpretation and processing, the scheme being split up in a set of procedures shown in Table 4, referring to this same blueprint PIM diagram.

**Table 3 Agent role in PIM**

| | *Role name* | *Role function* |
|---|---|---|
| Query | Get | Query the incoming message queue and get new messages. |
| | Comparison | Query two entities for equality. |
| | Select | Query a set of entities and pick out the one of special value. |
| Mutation | Initialisation | Mutate entities and set initial values. |
| | Set | Mutate the outgoing message queue with new messages. |
| | Finalisation | Mutate entities and set original values to finish up. |
| Helper | Conversion | Help the encoding or decoding of messages. |
| | Assertion | Help the check of conditions. |
| | Factory | Help the production of messages. |





PPL.EJIS.3000688

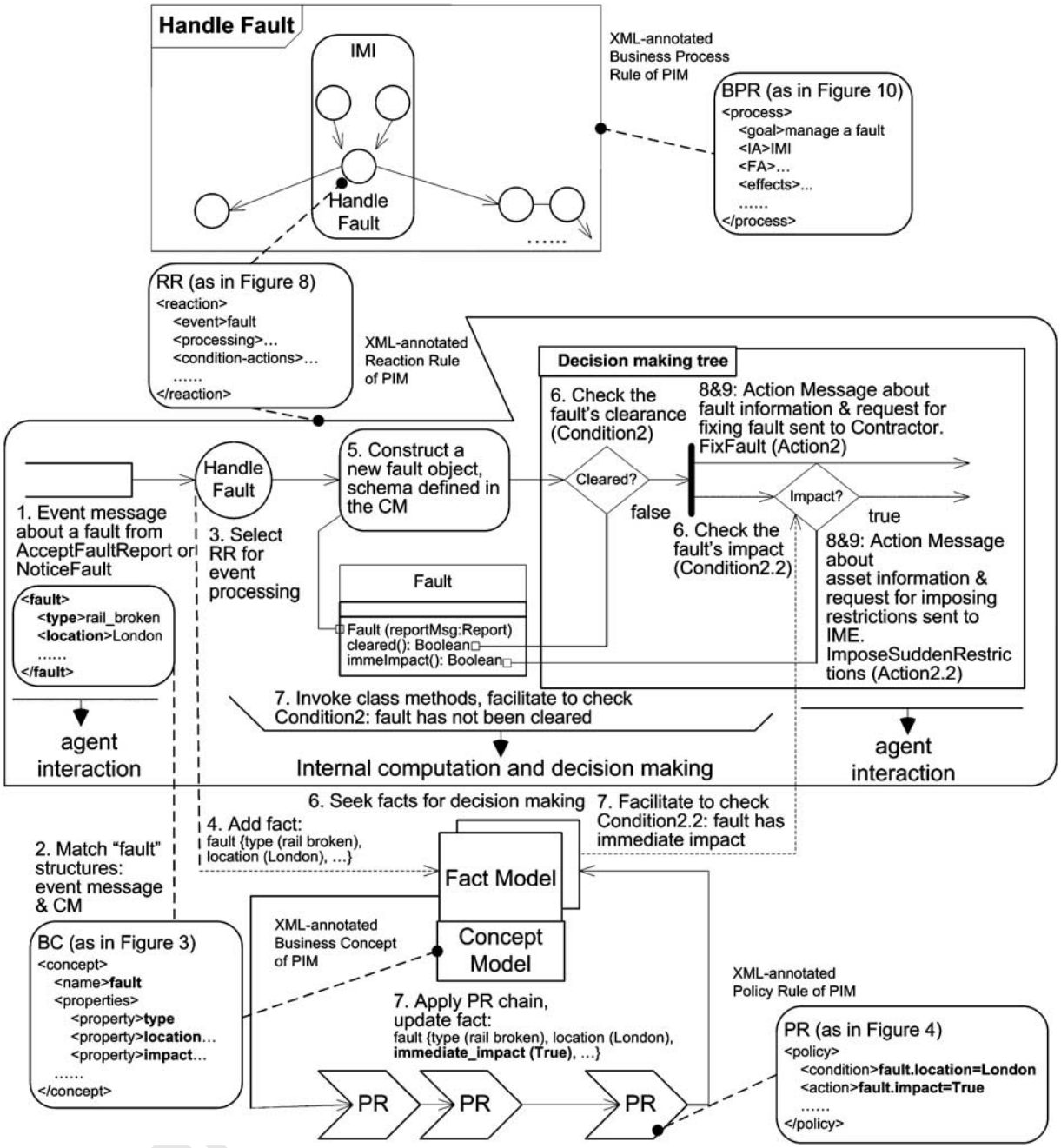


**Figure 15** A portion of the PIM for the case study (centred on RR 'HandleFault' of the BPR).

Consider the constituents of AMDA systems as initial Requirements (R), Models (M) being built and final Systems (S), then the following scheme expresses the propagation of changes in AMDA for some cases. None of those requires explicit (manual) changes to the final systems (Figure 16).

## Transforming to PSM and code

Figure 17 illustrates the technology-dependent target models transformed from PIM, specifically for the Java Agent DEvelopment Framework (JADE) (JADE platform). Platform-specific details not present in the PIM are now included, with agent classes, business object classes as well as other supplementary classes required by the platform. As shown in the figure, each agent class is associated with several role classes that capture the roles the agent will play. Each role class is further associated with one or more behaviour classes that implement the action() method, the actual operations to be performed by JADE agents when the behaviour is in execution. This



PPL.EJIS.3000688

**Table 4 PIM driving agent behaviour**

| *Steps* | *Generic role playing* | *Operations in case study* |
|---|---|---|
| 1 | Agent plays its GET role and gets an incoming message from its incoming message queue. | A fault is reported to IMI. |
| 2 | Agent plays its COMPARISION role and validates the encoded object structure using the Concept Model (CM). | The 'fault' structure encoded in the message matches with the one defined in CM. |
| 3 | If the object structure is equally defined with one in the CM, then Agent plays its SELECT role and finds the Reaction Rule (RR) from the RR Model that is defined to deal with this event. | The RR 'IMI–HandleFault' is selected in this context as its $<$event$>$ section is specified to handle reported faults. |
| 4 | Agent plays its INITIALISATION role and populates the object structure in the Fact Model (FM) with values seen in the message. | A fact about a 'fault' is established in FM with its location of 'London' as well as other information. |
| 5 | Also Agent plays its CONVERSION role, decodes the message and constructs a new business object available to the Class Manager Agent. | A business object 'fault' is constructed using the same schema as defined in CM. |
| 6 | Agent plays its ASSERTION role and checks if conditions specified in the RR are satisfied using the Fact Manager Agent (FMA). | Facts in FM are looked for in relation with the conditions of the RR. |
| 7 | In this process, the interactions of FMA with Policy Rule Manager Agent (PRMA) and Class Manager Agent produce facts to evaluate conditions. | FMA interacts with PRMA/CMA to seek additional knowledge either by applying relevant PR or invoking related class methods. The fault is known as having impact as a result of its location, indicated by a PR. |
| 8 | Agent plays its FACTORY role and produces a message as the result of the action coupling with the satisfied condition as defined in the RR. Prior to that, Agent plays its CONVERSION role and a business object available to the Class Manager Agent is encoded into the message. | The business objects of 'fault' and 'asset' established previously are retrieved and encoded in messages. The messages are prepared to be sent to responsible agents to fix faults and impose restrictions as defined in $<$action$>$ of the RR. |
| 9 | Agent plays its SET role and puts the message to its outgoing message queue. | Two potential messages are ready for sending. |
| 10 | Agent plays its FINALISATION role. Temporary facts are demolished, and FM knowledge is restored its original state. | Facts about the particular fault are cleared after their use. |



AMDA {R, M, S}: {R [functional requirements], M [Agent Model, (BC, PR, RR, BPR) in UML&XML], S [agent instances & model repository]}
Propagation of changes:
Computation change: {R(c), M(RR's internal processing using objects), S(agent-object links implicitly reinterpreted)}
Interaction change: {R(i), M(RR's event receiving and action sending), S(agent-agent links implicitly reinterpreted)}
Policy change: {R(p), M(PR), M(RR's internal processing using PRs)}

**Figure 16** AMDA scheme for propagation of changes.

involves event message processing, decision making and action performing. Instances of ACLMessage classes will be decoded or encoded in communicating event and action messages. Business classes will facilitate the internal decision making via invocation by the behaviour class. RR structure can thus be mapped to an agent behaviour class in which a decision-making tree is represented for internal computation, and event and action message classes represented in the standard ACL formalism for agent interaction.

The PSM can be further transformed at the implementation level to code executable upon the JADE platform, agent acts described generically in Table 3 and specifically for the case study in Table 4 being turned into platform-specific constructs. For example, the Get act can be expressed as 'ACLMessage msg = myAgent.receive()', the Set act as 'myAgent.send(msg)', the Comparison act as an 'equals' statement and the Assertion act that evaluates the decision-making tree structure including {condition, action} couplets as 'switch' and 'case' or 'if' (and 'then') statements.

JADE agents executing their behaviour using such specific language constructs can be illustrated by the pseudo code of the sample *HandleFault* behaviour in a single simplified method transformed from the PSM as shown in Figure 18.



PPL.EJIS.3000688

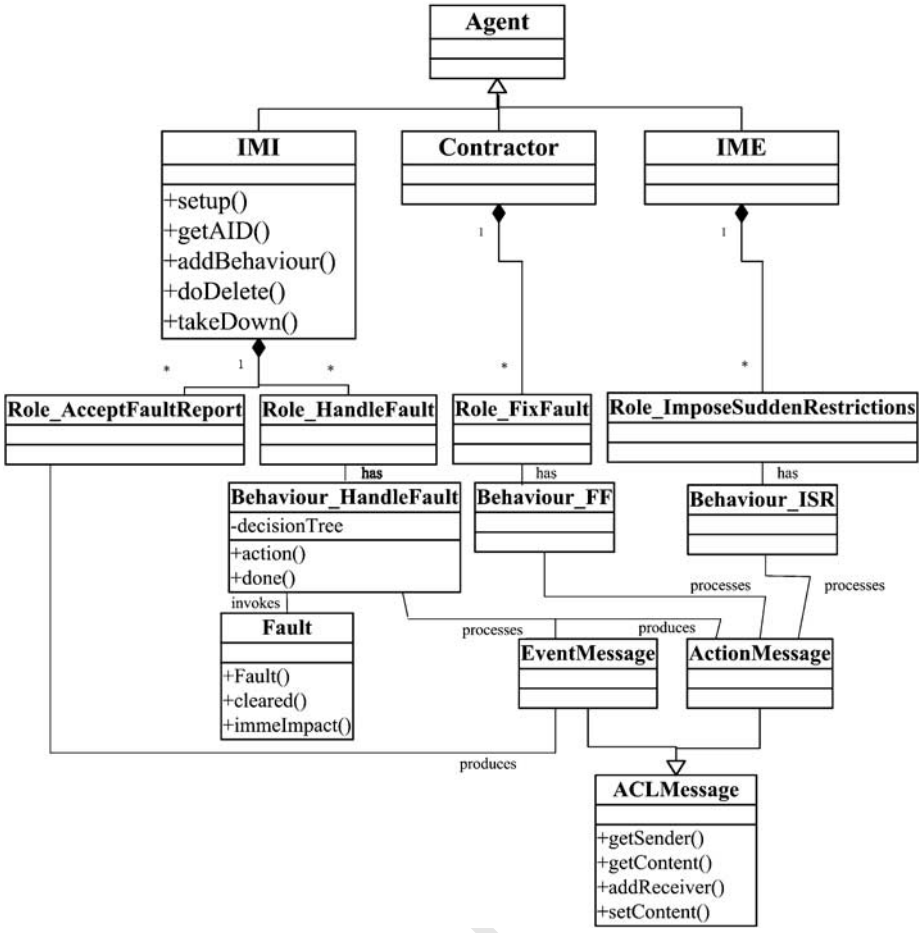


**Figure 17** A portion of the PSM for the case study (centred on the 'HandleFault' of the IMI).

'Rule' is a JavaBeans that retrieves rules from the AMDA model repository and do XML-based rules parsing and Java objects assembling for agents operation. When agent IMI and IME interact using *HandleFault* and *ImposeSuddenRestrictions*, respectively, a schema is shared between the action message of former rule and the event message of latter rule, in this instance an assert structure, that both agents agree for information exchange. During the processing of rules, even the internal object representations cannot be directly understood by sender and receiver, they will be converted to XML structures as FIPA Agent Communication Language (ACL) (Foundation for Intelligent Physical Agents) message contents for information interchange, according to the common schema and vice versa. This enables objects written in different languages for mutual communication and understanding. Data binding techniques such as Java & XML data binding (McLaughlin, 2002) are applicable to convert between Java or other programmed object instances and XML instances. XML data structures from incoming messages can be populated into objects via an unmarshalling process according to the agreed XML schema and conversely, XML instances are produced from objects for outgoing messages via a marshalling process. For example, a Java object can be marshalled by an agent to XML, transmitted over the network and unmarshalled as a C++ object understood by another agent for its internal use.

Apart from the underpinning programmed objects being enabled of interoperable use by AMDA across object-oriented programming platforms, agents developed across agent-oriented programming platforms can also use the universal AMDA models for communication and collaboration, in an adaptive manner. When the dynamic rule selection and execution process formalised in the PIM is transformed into various PSMs or code as ordinary agent behaviour classes suitable for JADE or other agent platforms for their interoperation using common message exchange, the implemented classes could have constrained flexible agent behaviour as they do traditionally. However, the original dynamic characteristics built into the PIM are maintained, agents always executing the rule appropriately configured for the current context at runtime. Specifically, since methods such as getEvent() and getDecisionTree() provided by our 'Rule' module will be invoked once per running of the





PPL.EJIS.3000688

```
thisAgent. addBehaviour (Rule thisRule) {
thisBehaviour. setPriority (thisRule. getPriority ());
    Asset asset;
    Fault fault;
    Message m = thisAgent. receiveMessage ();
    while (m != null)
    {
       Agent fromAgent = m. getSenderAgent ();
       if (fromAgent. equals (thisRule. getEvent (). getMessage (). getFromAgent ()) &
          m. equals (thisRule. getEvent (). getMessage ()))
       {
          XMLSchema schemaIn =
             thisRule. getEvent (). getMessage (). getSchema ();
          for (int i = 0; i < thisRule. getDecisionTree. size(); i++)
          {
             XMLSchema schemaOut [i] =
                thisRule. getDecisionTree. getAction (i). getMessage (). getSchema ();
          }
          ObjMsg reportMsg =
             m. getContentObject (). unmarshal (schemaIn);
          asset = new Asset (reportMsg);
          fault = new Fault (reportMsg);
          /* if (!fault. cleared ()) then send fault's XML serialisation form to Contractor
             if (fault. immeImpact () then send asset's XML serialisation form to IME */
          for (int i = 0; i < thisRule. getDecisionTree. size(); i++)
          {
             if (thisRule. getDecisionTree. getCondition (i))
             {
                XMLMsg xmlMsg = thisRule. getDecisionTree. getActionObj (i). marshal (schemaOut
[i]);
                Message m2 = new Message ();
                m2. setContentObject (xmlMsg);
                Agent toAgent =
                   thisRule. getAction (). getMessage (). getToAgent ();
                m2. addReceiverAgent (toAgent);
                thisAgent. send (m2);
             }
          }
       }
      m = thisAgent. receiveMessage ();
    }
}
```

**Figure 18** The pseudo code for the case study (IMI agent's behaviour 'HandleFault').

behaviour, and the rules are externally maintained and retrieved at runtime for dynamic decision making, the continuous updating of the model via tools guarantees the running agents always execute the required rules and hence requirements captured.

## Conclusions

This work represents an original attempt to couple the agent-oriented paradigm and MDA, the methodology proposed by this paper can make the development of complex systems better aligned with changing business needs easier and less costly. The major contributions of AMDA are three-fold.

Firstly, AMDA contributes to the Agent-oriented Software Engineering community a method of building adaptive MAS with overall development process support. Behavioural semantics are associated with model constructs and maintained before and after agents translate from these their behaviour to deploy up-to-date requirements. Therefore, AMDA covers the complete MAS development process, filling the gap between major agent-oriented methodologies like *i** or Gaia and agent-oriented development platforms such as JADE. Further, systems built by the method do not enforce a constraint that agent behaviour must conform to statically specified IPs and so are adaptive.

Secondly, AMDA contributes to MDA research as a means of abstracting high-level business-oriented models understandable by business people so that they can bring real time change effects without caring about the implementation of low-level computing technologies. The abstracted business knowledge models are real business assets that business stakeholders value and can keep the formalised requirements validated at the model level. These agent-executable rule-based business models, once associated with corresponding agents, can later guide to the use of specific agent and object language constructs for runtime execution. This is in contrast with UML (or Agent UML, see AUML web site) and AS/OCL, where the notation system has to be interpreted by





PPL.EJIS.3000688

human beings manually during development, and the constraint language being unable to capture high-level business semantics. An important lesson learned from MDA experience is that models should be used to abstract selected elements of the implemented complex systems rather than replicate the abstractions in the programming languages (Schmidt, 2006). AMDA raises the level of abstraction from MDA's low-level object and object-constraint concepts to business-oriented constructs, capturing high-level interaction, decision making, policy application, and so on.

Associated with this shift in level of abstraction, the characterised approach further directs a means to tackle the growing issue of software complexity (Fiadeiro, 2007). Traditionally, the inherent technical complexity of systems is overcome by its decomposition into smaller chunks in size and then statically assembling the whole from the parts. A tougher type of complexity is now being recognised concerning the dynamic nature that allows their entities to individually perform dynamically or collectively interact adaptively under various social contexts to maintain existing properties or satisfy emerging ones. An analogy is that a computer can be used to run an application. Although its components of CPU or memory can be upgraded physically and reconfigured internally by technicians, the computer's social role of running applications is maintained without change of its social position to its user, even though the user could feel the application now runs quicker. The separation of functionality from social role and interaction is necessary to reduce social complexity. AMDA's separated levels of abstraction provides such a paradigm: agents abstract at the business/social-level role playing; business models capture interactions of business/social roles; and the internal functions of agents provided by components or services are separated below the business/social level.

Finally, AMDA contributes to distributed computing. We have not constrained our models in this respect, low-level computing facilities being developed locally for a single closed system. Instead, the layered computing hierarchy used in AMDA implies the paradigm can be used to model the interaction of disparate components and services in a distributed environment via agents dynamically making use of them in an integrated MDA, and so to cope with the challenges brought by the pervasive business settings and the associated so-called Global Requirements Engineering (Damian, 2007). This is a nontrivial feature of AMDA since software is no longer a monolithic system running on a single computer (Fiadeiro, 2007) but rather evolves in line with business acquisition and collaboration, making use of heterogeneous services. Components and services may be ready to use but in a context their original developers have not planned, their integration from various sources in an open environment such as Internet being an emergent need. AMDA agents, being technology-independent, are well suited to the coordination and interoperation of separately developed components and services in our adaptive models. Their individual use or interaction among them is not coded in advance so that emerging business needs can be met via selection and connection of the relevant ones at runtime.



## About the authors

**Liang Xiao** is a research fellow in the School of Electronics and Computer Science at the University of Southampton in England. He obtained his B.Sc. at the Huazhong University of Science & Technology in China, his M.Sc. at the University of Edinburgh in Scotland and Ph.D. at Queen's University, Belfast in Northern Ireland. 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. Specifically, his research work focusses on Software Adaptivity, Agent-oriented Software Engineering and Agent-oriented Model-Driven Architecture. The results of his research have been published at several international conferences and journals. His current research activities in Southampton include working on the EU-funded projects of HealthAgents and OpenKnowledge. **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 career in academia at Queens and previously at Ulster, he worked in industry as an analyst–programmer, and his research is inspired by real problems in software engineering. His particular research interests are in Software Adaptivity, Iterative and Incremental Software Processes, Software Evolution Planning and Software Risk Management.

## References

AUML WEB SITE. http://www.auml.org/.

BRUNS R, DUNKEL J and OSSOWSKI S (2005) Advisory agents in the semantic web. In *Proceedings of the Sixth International Conference on Enterprise Information Systems*, pp 271–278.

DAMIAN D (2007) Stakeholders in global requirements engineering: lessons learned from practice. *IEEE Software* **24(2),** 21–27.

EHRLER L and CRANEFIELD S (2004) Executing agent UML diagrams. In *Proceedings of the Third International Joint Conference on Autonomous Agents and Multiagent Systems*, pp 906–913.

Q2 Q3



PPL.EJIS.3000688

FIADEIRO JL (2007) Designing for software's social complexity. *IEEE Computer* **40(1),** 34–39.

FOWLER M (2004) *UML Distilled: A Brief Guide to the Standard Object Modeling Language* (3rd edn). Addison-Wesley.

FOUNDATION FOR INTELLIGENT PHYSICAL AGENTS. http://www.fipa.org/.

FRANCE R, GHOSH S and TRONG T (2006) Model-driven development using UML 2.0: promises and pitfalls. *IEEE Computer* **39(2),** 59–66.

JADE PLATFORM. http://jade.tilab.com/.

KLEPPE A, WARMER J and BAST W (2003) *MDA Explained: The Model Driven Architecture: Practice and Promise*. Addison-Wesley.

MCLAUGHLIN B (2002) *Java & XML Data Binding*. O'Reilly.

MELLOR S and BALCER M (2002) *Executable UML: A Foundation for Model Driven Architecture*. Addison-Wesley.

MESERVY T and FENSTERMACHER K (2005) Transforming software development: an MDA road map. *IEEE Computer* **38(9),** 52–58.

OBJECT MANAGEMENT GROUP. 250 First Ave. Suite 100, Needham, MA 02494, USA.

OBJECT MANAGEMENT GROUP (2002a) CORBA 3.0 – IDL Syntax and Semantics chapter. OMG document formal/02-06-07, USA.

OBJECT MANAGEMENT GROUP (2002b) OMG Unified Modeling Language Specification (Action Semantics). OMG document ptc/02-01-09, USA.

RIEHLE D and PERRY A (2002) Framework Design and Implementation with Java and UML. Tutorials at OOPSLA.

SCHMIDT DC (2006) Model-driven engineering. *IEEE Computer* **39(2),** 25–31.

WOOLDRIDGE M, JENNINGS NR and KINNY D (1999) A methodology for agent-oriented analysis and design. In *Proceedings of the third International Conference on Autonomous Agents*, pp 69–76.

XIAO L and GREER D (2005) The adaptive agent model: software adaptivity through dynamic agents and XML-based business rules. In *Proceedings of the 17th International Conference on Software Engineering and Knowledge Engineering*, pp 62–67, Taipei, Republic of China.

XIAO L and GREER D (2006a) Externalisation and adaptation of multi-agent system behaviour. In *Advanced Topics in Database Research, Volume 5* (SIAU K, Ed.), pp 148–169, Idea Group.

XIAO L and GREER D (2006b) The agent–rule–class framework for multi-agent systems. *International Journal of Multi-Agent and Grid Computing* **2(4),** 325–351 IOS Press.





PPL.EJIS.3000688