Available online at www.sciencedirect.com
ScienceDirect
Information and Software Technology 51 (2009) 109–137
INFORMATION AND SOFTWARE TECHNOLOGY
www.elsevier.com/locate/infsof

Adaptive Agent Model: Software Adaptivity using an Agent-oriented Model-Driven Architecture

Liang Xiao [a,*], Des Greer [b]

[a] School of Electronics and Computer Science, University of Southampton, Southampton SO17 1BJ, UK
[b] School of Electronics, Electrical Engineering and Computer Science, Queen's University Belfast, Belfast BT7 1NN, UK

Received 19 December 2006; received in revised form 29 January 2008; accepted 9 February 2008
Available online 16 February 2008

Abstract

Model-Driven Architecture (MDA) promotes the development of software systems through successive building and generation of models, improving the reusability of models. Applying the same principles to the area of Agent-Oriented Software Engineering (AOSE) advances the ideas behind MDA even more significantly, due to the inherent adaptivity of software agents We describe an appropriate set of models originating from requirements specification and transformable to models understandable and executable by agents thus demonstrating an Agent-oriented Model-Driven Architecture (AMDA) approach. In AMDA, agents use hierarchical business knowledge models with business process rules at the top, business rules to control policy and logic in the middle and a base layer defining business concepts. Being externalised, knowledge is easily configurable by human beings and applied by software agents. A real case study is used to illustrate the process. The main advances over the object-oriented MDA are (i) the addition of component dynamics (ii) the use of agent-executable rule-based business models and (iii) a proposed higher level of abstraction with the direct representation of business requirements.

Keywords: Agent; Business knowledge modelling; Model-Driven architecture; Software adaptivity

1. Introduction and motivation

In a perfect world, a good engineer builds a perfect system, the customer is satisfied, and the maintainer of the system has little to do to keep the system up and running [1]. In the real world, business environments and business needs are changing rapidly and progressive change and adaptation of the system is inevitable if customer satisfaction is to be maintained.

Although it is expensive and difficult, software engineers and developers today must keep maintaining IT systems to align them with the changing business needs, if they are to remain useful [2]. Software adaptivity and maintainability must therefore be taken into account throughout the full software life cycle rather than only at the end of the software production phase.

Many software systems used in business environments nowadays are object-oriented (OO). Maintenance and evolution of OO systems are difficult because they are often not built to be adaptive or flexible, and so are resistant to frequent modification. Efforts have been made towards adding to software adaptivity including the use of the Strategy Design Pattern [3], Coordination Contracts [4] and Adaptive Object Model (AOM) [5]. All of these approaches and others attempt to add adaptivity to systems but offer only limited adaptation and end up creating complexity, unjustifiable presumptions, or introducing side effects that inevitably impede their use [6]. For example, the code that implements adaptation is often tangled and hard to manage or comprehend [7]. In fact the OO paradigm facilitates design by use of such principles as modularity and information hiding, but these imply ease of re-design rather than adaptation during operation.

* Corresponding author. Tel.: +44 7881894360.
E-mail addresses: lx@ecs.soton.ac.uk (L. Xiao), des.greer@qub.ac.uk (D. Greer).

0950-5849/$ - see front matter © 2008 Elsevier B.V. All rights reserved.
doi:10.1016/j.infsof.2008.02.002

The difficulty in ensuring adaptivity is also highly related with the software development lifecycle. Typically, 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 behaviour in the required system. The initially captured and elicited requirements models are typically documented in textual descriptions, while design models documents are in UML models. These documents and models provide a high level view of the system and guide developers in producing running systems from the specification. However, on one hand, the original requirements models, being largely textual descriptions of system functions, are separated from the developed design models, which lack the capability to capture the exact behavioural semantics from what is stated in the functional requirements [8]. On the other hand, related to this, the UML artifacts cannot be straightforwardly turned into running systems and would rapidly lose their value if, as is often the case in practice, maintenance changes are done at the code level only. The manual and error-prone transforming of requirements specification to design models and then to code is a major limitation of traditional software system development, especially with regard to high cost of software maintenance.

We want to break the tradition of having static component structure/behaviour characteristics as in OO systems but we also want to reuse models. Our hypothesis is that, if we can capture business requirements in a set of models, and use system components to interpret dynamically their behaviour from the models, then the maintenance of the models becomes the maintenance of the actual software system.

2. Background: MDA and Agent-oriented MDA

Model-Driven Architecture (MDA) [10–12,8,55] promotes the production of models with sufficient detail so that they can be used to generate or be transformed into executable software, running on target systems [43]. In MDA, models are central rather than an overhead in the development process. Change to models can be synchronised in code automatically without redevelopment. 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). The translation is based on a particular technological implementation including specific constructs and features of the implementation [42]. PSM is translated into code in a similar pattern.

2.1. Insufficiency of MDA

One difficulty with MDA is that the process of PIM → PSM → code starts from the design products rather than requirements models. Consequently, it requires highly creative work [11] to build a PIM from narrative requirements documents. This results in high costs in requirements change due to the need of highly skilled professional engineers for the process. Moreover, recognising that UML alone is not able to capture some semantics in its diagrams [13], a combination of UML and OCL [10] is used in MDA. However, OCL constraints are static and are external 'add-ons' to UML [14], used in the design stages rather than the requirements stages. Although the new UML 2.0 enhances the previous versions and adds some semantics, its numerous modelling concepts, poorly defined semantics, and lightweight extension mechanisms makes its application to MDA difficult [55]. Furthermore, MDA relies heavily on the tools which are supposed to have strong transformation capabilities from PIM to PSM and then to code. The reality of vendor inertia in implementing standards, and supporting the inherently complex transformation of a generic model to their various target platforms (e.g. J2EE or .Net) makes such auto-transformation especially hard to achieve [15].

Though MDA is confronted with various difficulties, it does impact the Software Engineering research community significantly with the idea of (re-)using models to drive the development [42,11]. MDA can reproduce OO systems despite the inherently static nature of object structure and behaviour, code being regenerated from models. However, changes cannot be made to mission-critical systems at runtime without interruption. More importantly, some business representation cannot be straightforwardly or even appropriately formed as objects, such as business rules. Additional maintenance burden would be otherwise added to systems if business rules were hard-coded [9]. These weaknesses in object technology have lead to the exploration of an alternative component technology at a higher level abstraction, being capable to retrieve, understand, as well as interpret business knowledge directly and dynamically.

To complement the running components, models must be built representing actual business needs and configurable by business people. New requirements must be easily supplied by business experts without IT intervention. This leads to a responsibility shift in that business people will get more involved in software maintenance, an adaptive system infrastructure being built to accommodate new business needs. While generating software systems from models is a method used by MDA to update system behaviour, actual running components interpreting models dynamically is even better. Ultimately, such a software system might never need re-delivery and therefore suffer no downtime or lost business opportunity due to unavailability caused by waiting for the next release. The system is under maintenance by customers, the current requirements being brought to the system persistently and engineers are freed from routinely maintaining code.

2.2. Software agents

The agent concept is conceived as an alternative to the object concept with agent as the main modelling and

execution component. Agents have been credited as an advance in Software Engineering abstractions, after the appearance of other abstractions such as procedures, data types, and objects [46]. Attempts that cast agent-orientation as the next major Software Engineering paradigm include [47,48]. Agents are useful for requirements modelling [49] as well as implementation [50]. In general, agents are reactive and pro-active, they have intentional behaviour, and try to achieve their goals by performing actions dynamically [47]. In contrast with 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. Models can provide the knowledge sources to agents for driving such flexible behaviour and thus can be reused continuously. Intelligent/autonomous agents have been proved useful for bringing dynamics, flexibility and adaptivity to many different domains [16–21]. The combination of externalised requirements (which become the maintenance target) and an agent system (which will be the maintenance actor) is also a promising solution for adaptivity.

2.3. Agent-oriented Model-Driven Architecture (AMDA)

An Agent-oriented Model-Driven Architecture (AMDA) is put forward. The hypothesis is that, the use of a new software development paradigm compatible with MDA with agent as the first class system component, will dramatically enhance software system adaptivity and maintainability. AMDA considers requirements modelling from the beginning, with its models being coupled with adaptive agents. This paradigm is not subject to the difficulties faced by object-oriented MDA. Since agents interpret from models dynamically rather than systems being generated again and again under IT support, business need not being interrupted to accommodate changes.

In AMDA, business models which represent business requirements are integrated into a set of Agent-oriented UML and become the PIM. They capture knowledge on agent structure and behaviour. Mutable software specifications are allowed and are recognised as the norm throughout the system lifecycle. Agent behaviour is driven by the knowledge model transformed from the specifications. Since requirements are in the first place unpredictable, models are re-configurable and directly link requirements to executable agents. An adaptive modelling structure is used, that gives the system a clear and comprehensible division into business units [22], and that is able to easily accommodate and facilitate business changes. Business models are the directly impacted entities after changes are required, thus their use as a PIM is appropriate. The amendment of these models is carried out directly by business people, while the execution of them is performed by agents, so reflecting deployed requirements. Therefore, changes coming from the requirements, rather than the design, can be passed to the software system without redelivery by developers. This allows newly arising specifications to be re-interpreted and renders recoding unnecessary. In fact, integrated requirements and design models get reconfigured constantly to reflect the required business changes. Consequently, the business knowledge modelling phase becomes the essential and primary step in software development, and effort spent on it will never be wasted.

The proposed approach is aimed to: (1) define easy to interpret business models that represent actual business requirements; (2) integrate business models as a Platform-Independent Model that agents can execute on various agent platforms and (3) establish a link from early software development phase of requirements to the later design and implementation phases, an area where MDA is lacking.

3. Models and approach in AMDA

This section discusses our approach. The main business model elements are *Business Concept*, *Policy Rule*, *Reaction Rule* and *Business Process Rule*. They are associated with an agent platform neutral Agent Model, together forming a Platform-Independent Model. An actual British rail track management system will be used as a case study as a demonstration.

3.1. Overview

Models are as important in AMDA as they are in MDA, the construction in both driving the development of the software systems and the runtime instances in both driving the running system architectures. Table 1 outlines the similarities and differences of the two paradigms.

Fig. 1 provides an overview of our overall system architecture used in AMDA including the artefacts used in model building (development and adaptation), their roles in the running system, as well as the knowledge and data exchanged among them.

The *Conceptual Model* provides a definition for all business concepts in use. These are determined from the requirements specification at build-time and referred to by agents at runtime, when it remains editable by business experts. The *Fact Model* constructed and used at runtime where previously abstract concepts are established as concrete facts which are then available to agents for decision making. The *Policy Rule* model contains business policies and strategies as initially determined from the requirements specification but remains editable at runtime. While the system is running, policy rules are applied to automatically update the Fact Model with inferred facts, new facts being accumulated and invalid ones demolished dynamically. Business classes associated with concept schemas built in the Conceptual Model may facilitate the inference of additional facts, by constructing business objects from communicating message contents and invoking them to manipulate existing facts. The *Reaction Rule Model* defines rules for individual agents as identified in the requirements specification constraining their reactions to events. Again, these remain editable at runtime where events drive agents'

Table 1
The similarities, differences, and relationships between MDA and AMDA

Similarities and differences	Main constructs used	Major development emphasis	Final system production	Reused entities in the development process	Applicable environments
Approaches					
Model-Driven Architecture	Objects	Model building	Code generation from models (once per change of model)	Models	(Large-scale) Systems being produced and running locally
Agent-oriented Model-Driven Architecture	Agents (business objects and services can be used to facilitate agent function)	Model building	Agent behaviour driven by models dynamically, models being under continuous maintenance	Models as well as existing components and services developed in house or supplied from elsewhere	Making use of a wide range of disparate entities and running in distributed environments

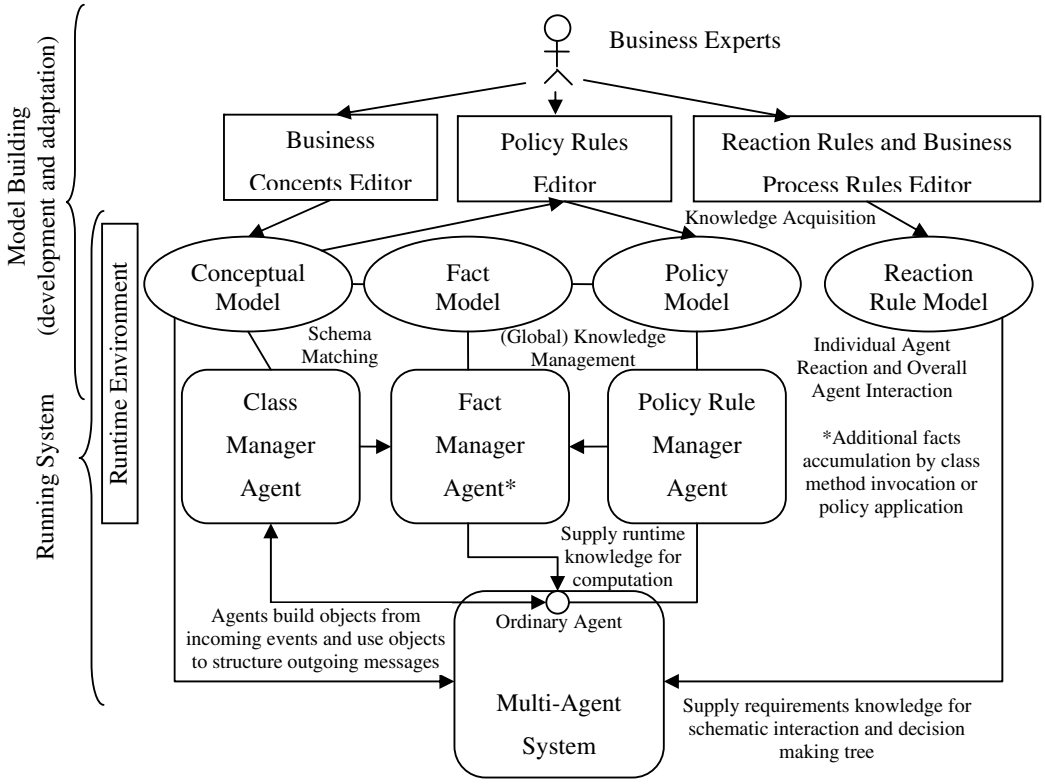

Fig. 1. Overview of models used for developing and adapting agent systems.

behaviour. Using reaction rules, when event messages are received, agents become aware of their collaboration partners, reaction patterns, and decision making procedures to be carried out. The *Business Process Rule Model* defines business processes in terms of the interaction of multiple agents with a shared aim of delivering a particular business goal. At build-time these are defined sequences of successive events processing via interconnected rules which then execute at runtime. The facilitating agents (Class Manager Agent, Fact Manager Agent and Policy Rule Manager Agent) shown in Fig. 1 and their role-playing behaviour among various models and the Multi-Agent System will be described in the following sections and illustrated altogether further in Fig. 18 of Section 4.2.

Policy Rules, Reaction Rules and Business Process rules are differentiated by their scope and composition. 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 specific agents should use and describe reactions that must be performed when triggered by external events. Business Process Rules (BPR) realise business processes aimed at corresponding goals, a collection of PRs and RRs may be involved when a BPR is applied.

To make AMDA work, these modelling elements are hierarchically structured in business knowledge models, complemented by hierarchically structured computing components, as illustrated in Fig. 2. In brief, agents dynamically behave, the behaviour of which is driven by the captured business knowledge models in order to meet business needs at various levels. In model execution processes, objects support agent behaviour. As a result, the

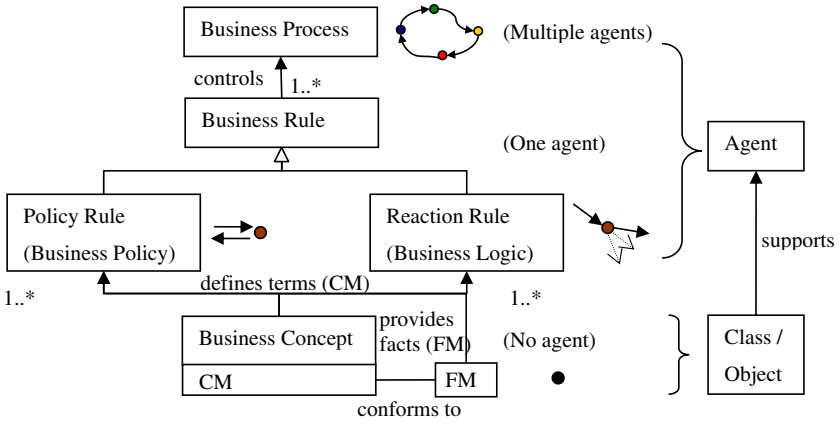

Fig. 2. AAM components structure: two interactive hierarchies.

interaction of the two hierarchies achieves the required business goals. Collectively these hierarchies are termed the Adaptive Agent Model (AAM) and it is this model that forms the basis of our AMDA approach.

3.2. The meta-model

Fig. 3 shows a meta-model of the AAM approach, starting from Agent-oriented Requirements Modelling [56]. In our approach, *business processes* realise *business goals*, *agents* collaborate towards business goals, and *business rules* collectively support business goals through their control of business processes. Agents, therefore, have the responsibility of achieving the business goals. Individual agents collaborate by playing *business roles*, which are, in turn, constrained by business rules, which represent the fundamental functional requirements. *Business functions* are distinguished from business rules. Business functions represent stable blocks of requirements that are owned and later implemented by the lower level *business classes*. Conversely, business rules represent volatile blocks of requirements that are attributed directly to the higher level agents. It could be specified in business rules what, when, and how particular business functions of business classes are used by the agents.

The meta-model provides a perspective from a requirements model as well as an execution model. The proposed AAM is aimed at an integrated development process. In requirements modelling, agents are used to organise requirements from different domains. In the execution system, agents represent actors that act upon the domains with respective domain knowledge. Usually, requirements are structured by domains, in each of which a collection of requirements with the same nature are grouped together. Naturally, one business domain can be delegated to one agent, who has knowledge concerned with that domain. Agents represent actors that are responsible for the function of their respective business domains. When the domain is required for different purposes, the corresponding agent responds and plays several roles in order to realise several aspects of domain functions, so fulfilling its responsibilities. Interactions among domains are delegated to message

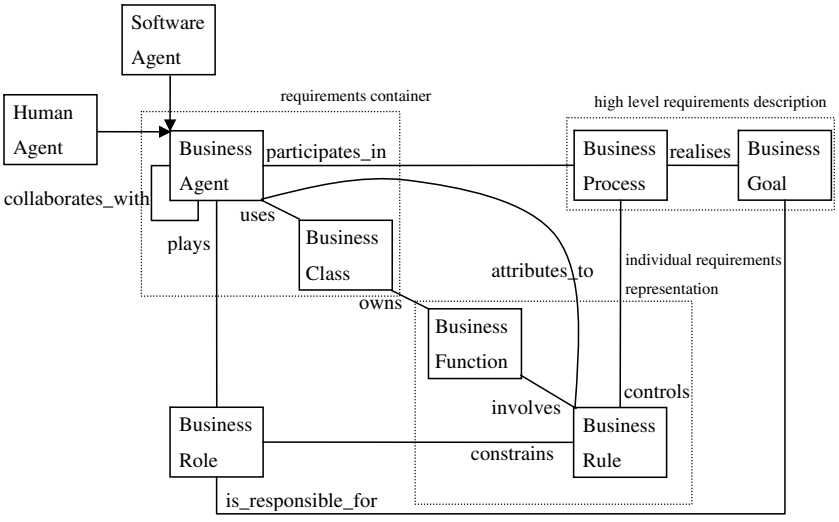

Fig. 3. Meta-model for requirements modeling.

passing among respective agents. Such cross-domain interactions require collaboration of agents, and the collaboration pattern of agents is decided by the interactivity of functions of the involved domains. Domain requirements are assigned to agents in the form of rules or class functions. Later, this organisational 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. Principle modelling elements in this scheme are given below.

3.2.1. Agent

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

3.2.2. Rule

Rules are captured functional requirements that are configurable at runtime. Rules constitute externalised agent knowledge and inform agents their behaviour at runtime. 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 models. An agent chooses various rules to play various roles in various interactions.

3.2.3. Class

Classes are traditional passive components. 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.

3.2.4. Message

A message is an object 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 the related rules have been 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.

This sub-model is the core of the meta-model shown in Fig. 3 and actually structures Agent, Rule and Class in a hierarchy [26]: (i) agents are used to model conceptual domain units and are guided by collections of rules in domains; (ii) rules are used to capture requirements and guide agent behaviour and (iii) classes are employed by rules to support the function of agents. This provides a perspective on AAM composition from an execution viewpoint in contrast with the viewpoint of the AAM knowledge composition, presented previously. Here, rules represent both Reaction Rules and Policy Rules which compose Business Process Rules. Classes represent the implementation of Business Concepts. The two viewpoints complement each other and form the blueprint of AAM systems.

3.3. Case study

An actual British railway management system specification has been investigated as a case study. The system is mainly responsible for the running of the railway on a daily basis, monitoring train running with regard to incidents and ensuring the safety of the train services by conveying issues to relevant parties for resolution. Being a very complex system, the specification document has more than 250 pages and contains a large number of standardised form-based function descriptions.

Case background: The specification comprises three main areas: *Train Running and Performance*, *Infrastructure Management and Performance* and *Common Communications*, each of which is subdivided 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 [26].

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.

A large number of specific functional requirements tables have been documented for each domain. One of them, *IMI-HandleFault*, is given in Table 2, a focus of the case study. Formatting requirements in template tables lets requirements be expressed in a unified manner and allows engineers to manage them easier and to find their relationship and missing pieces quicker.

3.4. Requirements Model: Goal decomposition

Goals must be identified and refined so that they can be turned into specifications of operational services and constrains, assigning to agents as their responsibilities [23]. Goals are usually explicitly stated in the user requirements document, informing the purpose of the supporting business. A goal-decomposition technique is employed, similar to [57,58], starting from analysis of high level strategic concerns and ending up with the requirements specification broken down and low-level functional requirements exposed and related.

However, the technique to be applied here has its own distinction due to the fact that the developed business models must be mapped from goals and shall be interpretable to agent software at runtime. Many approaches and techniques alike refine goals until they are assignable to single agents and assign a goal to an agent only if the agent can completely realise the goal. For example, the assertion that a goal is realisable by an agent, as in [59], may be interpreted as an agent finding a path from its initial state to reach the next state, as required by the goal, by means of observing its monitored variables and manipulating its controlled variables. Thus, a lack of monitor-ability or controllability is usually the cause of unrealisable goals. Refinement tactics that introduce intermediate goals or agents or objects may be used to solve this problem. Such a goal-oriented RE strategy requires a one-to-one mapping of goal to agent, often related with formal refinement pattern and logic-based agent specification. In a business and IT context, however, it is probably not the best means for business/software modelling and development, where the aim is producing executable agent software systems and turning goals into tangible business models and maintainable agent knowledge. Moreover, in reality, many business entities must cooperate towards the same objective. Thus, such a modelling practice does not naturally match to an operational business model and cannot take full advantage of MAS development.

In our framework, organising functional requirements in terms of their goals can build a hierarchal requirements structure, where those at the bottom support those at the top. Top-level requirements are those most valued by the business people and reflect the final business goals. Subordinate goals can be derived iteratively using a goal-decomposition technique down to the lowest level goals, which map to individual functional requirements. The top level business goal, in the first place, can be decomposed into a set of intermediate goals. The decomposed goals, called sub-goals to distinguish them from the original goals, can be further decomposed into still smaller sub-goals. In the process they are considered just like the ordinary goals in the subsequent decomposition process. Finally, when the business goal is decomposed to the smallest granularity,

Table 2
Functional requirements table IMI-HandleFault

Domain	IMI
Identifier	HandleFault
Description	To maintain information about faults so that they can be fixed in a way which 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 used	Information about infrastructure assets and their contracts Information about *train journeys*to assess the impact of the *fault*
Outputs	Fault information to *contractors*
Required effect	The fault is recorded Unless the fault has already been cleared, the appropriate contractor is identified and agreement is reached about a priority for fixing the fault. 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. If necessary, *track restrictions*are put in place. If so there is an impact on the train service handled by *TRI-RespondToIncident* Anyone affected by the fault is notified (*CCI-NotifyIncident*)

the process terminates and all the leaf nodes are presented as operational functional requirements. This provides a means for incremental elicitation of requirements. Only when all the leaf nodes existing as requirements units presented in the specification and the top nodes being fully supported by the bottom ones, can the business goals be guaranteed to be represented completely in the requirements documentation. Since a business process realises a business goal, it is possible to dedicate a business process to one functional requirement, mapping to a sub-goal in a leaf node or, alternatively and more likely, a combination of these sub-goals aggregated in an intermediate node.

Whatever the case, it is not required here that one agent realises one goal as in many other approaches discussed above. Rather, it is more often the case that many agents participate in a business process, only through the collaboration of which the corresponding goal can be realised. The rationale for this is that, although it is possible to assign a single agent to an atomic requirement, the resulting architecture is probably not that natural or straightforward when realised in a business software system. More appropriately business processes are the direct representatives that organise many agents, whose functions and cooperation are towards the same goal. This is one of the novelties of our goal-decomposition and its distinctions from other such approaches in the Requirements Engineering area.

It is worth noting that, although "AND" and "OR" relationships among goals are both widely employed in goal-decomposition techniques, we omit the use of "OR" due to our motivation in using goals in requirements analysis. In other approaches, "OR" can facilitate the exploitation of alternative goal composition relationships. In requirements engineering, goals are usually used for requirements acquisition or for developing requirements specifications. In the process of elaborating a goal hierarchy, the requirements of a stakeholder are revealed, and stakeholders become more aware of potential alternatives for meeting their substantive goals [60]. As such, the goal-oriented requirements engineering is applied in the early requirements analysis phase to gain better understanding of potential low level requirements. In our case, however, we start from a well understood and fully documented requirements specification shown in Section 3.3, where all processes, rules, concepts involved in the system have already been made clear though embedded implicitly in the case description of the requirements specification. The requirements specification has been developed and agreed upon between stakeholders and developer teams. Nevertheless, the goal-decomposition approach is in use by our approach, for an opposite purpose: not for eliciting refined requirements, but for tracing back to the original top-level goal, thus ensuring requirements completeness and establishing the relationships among low-level requirements.

In the case where alternative goals exist as valid possibilities in the requirements specification this can be dealt with by separate rules (discussion of which follow) given differing priorities, the dominant rule having the highest priority attribute.

Fig. 4 demonstrates the decomposition process for the case study. The top business goal of "deliver train journeys safely" as emphasised in the specification, in the first place, can be decomposed into "deploy planned and unplanned train journeys" and "detect and handle faulty train paths". The former sub-goal can then be decomposed according to the nature of the train journeys. The latter one, being itself the top-level goal under our study, can be decomposed into "manage new fault" and "manage existing fault", since the processes of handling faults differs in different phases of the life cycle of faults. A fault is sent to be fixed and track restrictions are imposed if it is a new incoming one. Fault information gets updated and the corresponding track restrictions get updated as well (in some case removed if the fault has been fixed). In both cases train services are rescheduled. Such smallest granularity produced after the decomposition process are actually documented as functional requirements tables in the specification of our case study, which will be transformed into agent Reaction Rules

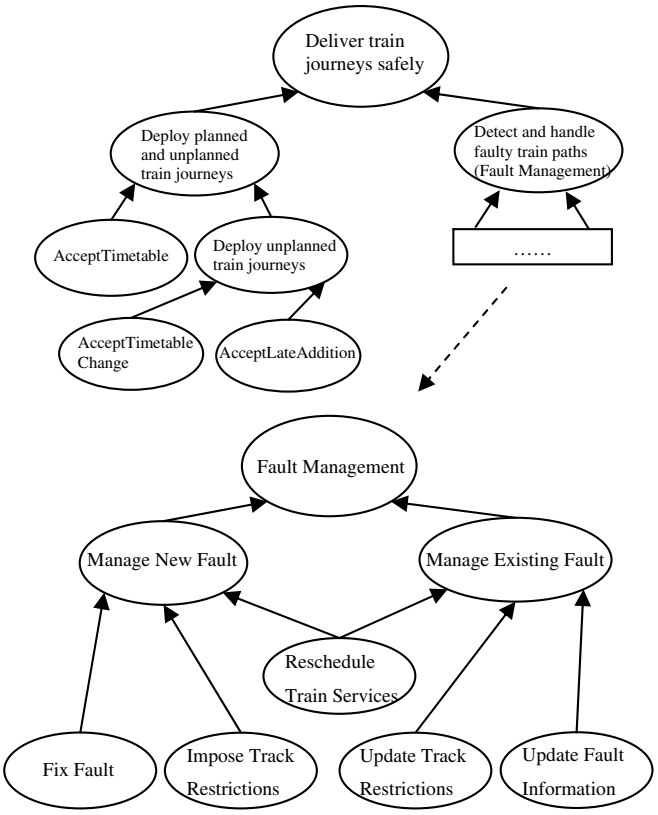

Fig. 4. Goal-mapping graph for the overall system and excerpted case study.

as detailed in Section 3.6.2. Among these is "*IMI-Handle-Fault*" as documented in Table 2. This phase facilitates the production of corresponding BPR, which will be discussed in Section 3.7.

3.5. Conceptual Model and Fact Model

Hierarchical business goal structure having been constructed, there will be a much better idea of the alternative business processes for realising goals. For example, Fig. 4 implies that there could be a business process of "manage new fault" and another business process of "manage existing fault", together composing a super business process of "fault management", which itself is part of the top business process for "deliver train journeys safely". However, it is not certain at this phase if lower level goal should be further decomposed. For example, "reschedule train services" might be a business function that supports two higher level goals of fault management but it might also be composed by multiple lower level business functions in which case it cannot be regarded as a simple function but rather decomposable.

Nevertheless, if graphs as such being constructed have exhaustively exploited the goal compositions as reflected in the original requirements specification, they should be sufficient to guide us in refining this goal model and in finding out the business process actually required and more importantly, the lower elements that compose individual business processes. Later (Section 3.7), we will see the business processes eventually built from low-level model elements but this top–down method of business model construction is now complemented by a bottom–up method, where as Fig. 2 indicates, we will start from the identification of the fundamental business concepts and facts presented in the specification, which are referred by business rules that further control the running of business processes. In doing so, the modelling process merges from two ends to business processes, the ultimate business assets and entities that define how business is working.

Business concepts should be treated as assets, externalised from the applications that use them, for the purpose of easy maintenance (agents in the same system can speak extendable vocabularies) and easy interoperability (agents in different systems can communicate if a mapping between their vocabularies is given). To achieve this, conceptual schemas must be made explicit. Further, having concepts related in structures makes information described by them interpretable and understandable by computers, or agents running on computers. One may use a grammatical analysis of natural language description of a system to discover objects to build the system. For example, objects may be identified from nouns, attributes from adjectives, operations from verbs, and so on [25]. This is also applicable in finding business concepts and their attributes. Business concepts identified in the fault management domain described in the case study using the technique include "fault", "incident", and "restriction". A "fault" has properties indicating its location, immediate impact, and priority, these themselves being business concepts and properties of the business concept "fault", i.e. 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 Conceptual Model. Additional relationships between business concepts may be enforced as required by business rules for business needs. One business rule in the domain may specify, by implication, a relationship between two of the identified business concepts, e.g. "immediate_impact" and "priority". More business concepts may arise as well as relationships among them when requirements change. Business concepts must be explicitly identified if they are later to be extended or if new business rules are to be defined using them.

At runtime, concrete facts are established as instantiations of abstract concepts documented as shown by "fault". For example, when a report arrives, one fact may be established that states a fault has occurred in London, being of a classified type of "rail broken", and so on. Properties of this business object are thereafter populated with values, i.e. fault {type (rail broken), location (London), ...}.

Facts like this accumulate and are stored in the Fact Model. One dedicated agent, the Fact Manager Agent (FMA) [24], is responsible for the management of all facts. It interacts with a Policy Rule Manager Agent (PRMA,

detailed in Section 3.6.1) to add new deduced facts after the application of Policy Rules, and can acknowledge all agents the facts available to the system. For example, one policy may say that a fault occurs at London will have a high priority of 10 and immediate impact. The Fact Model will be updated correspondingly, i.e. fault {type (rail broken), location (London), immediate_impact (True), priority (10), ...}.

The concept of "fault" and its related properties are represented in XML as shown in Fig. 5. This concept will be referred to by the PR shown in Fig. 6 and RR shown in Fig. 12. A corresponding business class "Fault" has later been implemented for agents to operate upon. Business objects can be instantiated from the class. These can be encoded in agent messages for their communication, announcing among themselves the occurrence of faults and requesting their repair. This lets FMA establish facts that will be known by agents. New facts are accumulated and invalid ones demolished dynamically at runtime. Business models are updated dynamically and reflected in agent knowledge.

The use of Conceptual and Fact Models is supplemented by a lower layer class facility which enables the use of existing OO infrastructure to represent and infer further knowledge. The facility is based on an agent–class hierarchy for Agent-oriented modelling and development [26], where dynamic agents invoke static class methods as required. At runtime, established facts are mapped to business objects instantiated from business classes, the schemas of which are as defined in the Conceptual Model. Methods defined on the business objects are invoked for the manipulation of facts as per the business rules. This leads to the availability of additional knowledge, supporting agents in their reasoning and behaviour. Reusable class libraries and exchangeable class methods may be used. It is up to the flexibly defined business rules that specific class methods are selected and determined to be invoked in various conditions to achieve dynamic effect.

3.6. Business Rule Model

Business rules constrain and connect concepts in the Conceptual Model and describe semantic relationships among them. These rules are stated in machine understandable form but also allow human comprehension and edition. Reactive Rules (RRs) are rules about processes that are in reaction to some events. They constrain individual agents' behaviour based on conditions and available business objects. Policy Rules (PRs) are rules concerning business policies. They constrain global relationship among business concepts that all agents must respect when they use them. Underpinned by the Conceptual Model and the Fact Model, both contain meta-data knowledge and they are applied by individual agents at various execution points of business processes, which involve multiple agents. The advantages of externalisation of agent behaviour in business rules are described in [44,45].

3.6.1. Policy Rule

The justification for using business rules is that they provide a means for implementing business policy [28]. Business policies, naturally, change over time and thus

```
- <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>
```

Fig. 5. 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>
```

Fig. 6. Policy Rule representation on fault impact for the case study.

externalisation of them as executable rules is desirable. It could be conceived that a Policy Rule (PR) captures a constraint of invariant type, while a Reaction Rule (RR) captures pre-condition and post-condition constraints.

Policy Rules are normally implicit and embedded in requirements specifications. The underlined sentence in the case study descriptions reads: *if the fault is located at capital cities, it has impact and needs to be fixed immediately*. Such rules must be made explicit, as well as represented in models. Otherwise, the embedment of them in code increases the maintenance burden.

Taking the above PR as an example, it says faults in the fault management system must be handled in accordance to the nature of the emergency. This category of rules is defined for classification of business concepts. Business concepts are sometimes required to be classified and treated differently because of their different nature/attributes. Fig. 6 shows a description of the rule and its XML representation, stating that any fault found at capital cities has immediate impact, and so it possibly needs to be handled differently from faults of no immediate impact.

In the knowledge of Rule1, the Policy Rule Manager Agent (PRMA) [26] would update the Fact Model through its interaction with the Fact Manager Agent (FMA) at runtime. Suppose a "fault" is informed to the system. A fact is then established in the Fact Model with its "immediate_impact" initially unknown but "location" known as London. It is through the PRMA's evaluation of the latter property using the PR currently set by the business people that the Fact Model is updated, so that the former property "immediate_impact" of the "fault" is set as "true".

Another category of PR is defined on the relationship of (classified) business concepts or their properties, triggering the formation of a chain of PRs. Rule2 (Fig. 7) uses the term "*immediate impact*" as 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 and a PR chain is thus formed. It is easy to infer that a fault located at capital cities has a high priority, as a PR chain is formed. In an iterative means, new facts are progressively known and other PRs may come into play, leading to additional facts being established, based on existing facts. The process proceeds until no more PR conditions are satisfied. Such relationships as concept reference, logical inference, and collaborations among PR contribute to the establishment of a PR chain. Other PRs may contain computational formulas that can be used to calculate the concrete values of certain elements at runtime.

A third category of PR is defined on the different system behaviour that can be expected depending on the (classified) business concepts/properties. Rule3 and Rule4 (Fig. 8) are actually used in combination with and to contribute to a RR (detailed in Section 3.6.2), distinguishing different means for fault handling in different conditions, as a result of different policies.

PRs are built from the requirements elicitation, analysis, decomposition, and reconstruction, providing models with traceability from their origin, so assisting maintainability. Systems using PRs can be implemented independently from any particular technology. Tools [6,29] have been developed to facilitate viewing, addition and edition of PRs as well as RRs. The combinational use of PR and RR, working in BPR towards a common goal is described in Section 4.

3.6.2. Reaction Rule

The overall architecture of a software system is important to the efficiency and effectiveness of maintaining it [30]. The AAM has an event-driven architecture, its agents being reactive to events in the knowledge of Reaction Rules. Event-oriented decomposition, based on events that the system must handle, is a design strategy that enables modular architecture, easy design, independent unit development, and eventually effective maintenance [35]. Events, when becoming part of our models, can drive agent behaviour, according to actions and conditions related with the events described in the models. Thus, agents react to events and through sequences of successive events, business processes are formed and executed logically. Naturally, not all future events are predictable and so enabling the evolution of agent reactivity locally and business process interactivity globally is essential. This evolution is possible since Reaction Rules are continuously editable and variation of events can dynamically drive agents to choose various external interaction patterns or internal event processing procedures.

Consisting of events, RRs define agreements that are bound between agents for their interactions, constraining what and how agents should perform in a reactive manner. Driven by events, agents use RRs to make business decisions using condition and action pairs. When an event message is received by an agent, business objects are decoded from it and facts become known to the recipient. To respond to this new knowledge, the agent may make a decision and perform an action, possibly produce event messages to other agents in order to accomplish a business process.

Since different decisions can be made by individual agents in different situations upon receiving dynamically generated events, individual agent reaction and multi-agent interaction can be adaptive according to configurable rule models.

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

Fig. 7. Policy Rule of the second category from the case study.

```
Rule3.
If fault has no "immediate impact"
Then IMI-HandleFault does nothing

Rule4.
If fault has "immediate impact"
Then IMI-HandleFault establishes a new incident associated with the
fault AND request IME to place track restrictions
```

Fig. 8. Policy Rule of the third category from the case study.

Fig. 9 shows the Reaction Rule Model and how an agent processes a RR 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.

Reaction Rules, as specified here, make agent an 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. The configurability of rule models accommodates mutable requirements including collaboration partners, events processing, reaction selection so eventually driving changing system behaviour.

In the case study specification, a set of functions are required for managing faults. Three functions of special concern are reconstructed in Fig. 10. They constrain business domains to their expected function in different aspects. *IMI-HandleFault*, for example, has its prefix indicating that it belongs to the IMI business domain, and constrains IMI in its handling of faults in reactions. This function is documented in a functional requirements table in Table 2. RRs specify domain functions as well as respective constraints with behavioural semantics.

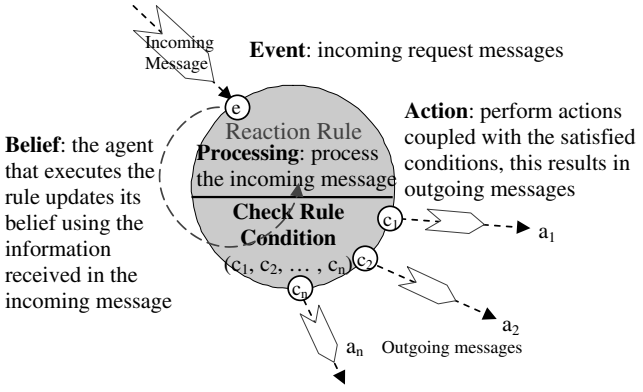

Fig. 9. Reaction Rule Model.

Agent IMI processes its RR *IMI-HandleFault*in the manner shown in Fig. 11.

A RR acts like a contract between agents. For example, *IMI-HandleFault*, as a fault handling RR in this fault management domain, will only respond if an event message with a pre-defined information structure representing an asset "fault" being received. In addition, it promises predefined information structures sent to the pre-agreed partners as defined by the RR. The XML specification of the RR is shown in Fig. 12. A guideline for transforming functional requirement tables as shown in Table 2 to RR structures and then XML specifications is provided in [26].

XML-based rules provide precise definitions of agent behaviours to the Design Models, something UML diagrams lack [8]. In general, each agent reacts to the receipt of a message by executing a rule using the following process shown in Fig. 13.

RRs can play the role of a connector [32,33] or a contract [4], as they have the capability of evolving software architecture via rule configuration. Compositional parts of rules separate computation from coordination. ⟨event⟩ and ⟨action⟩ parts serve as interfaces, connecting one agent with its coordinated agents through incoming or outgoing messages and so model architectural interfaces. The ⟨processing⟩ part serves internal computation. This part captures the invocation of existing components and so models internal computational structure. This separation of concerns makes software evolution easier. Agent collaboration patterns and the main processing component can be maintained individually. From this perspective, the ⟨event⟩ section models the pre-condition of agent behaviour, the ⟨action⟩ section models the post-condition of agent behaviour, the ⟨condition⟩ section models the guard and controls the rule execution path. At runtime, agents check these mutable constraints separately and dynamically.

It should be noted that class methods such as "cleared()" and "immeImpact()" may arise from the need of function facility of the corresponding classes. These can be invoked to facilitate agents to operate, judging conditions and performing actions. Other methods may directly originate from functional requirements tables, which are sub-requirements of others and support common functions rather than specific business tasks. One example is that the "ValidateTrainPlan" requirement is a sub-requirement of the "AcceptLateAddition" requirement

IMI-HandleFault *is informed by* IMI-AcceptFaultReport or IMI-NoticeFault about an asset fault,

IF the fault has been cleared *THEN* DO_NOTHING,

ELSE

Inform the responsible **Contractor** about the fault with an agreed priority,

IF the fault has no immediate impact *THEN* DO_NOTHING,

ELSE

Create an incident related with the fault AND

Create and put in place track restrictions *using* **IME-ImposeSuddenRestrictions** AND

Inform concerned parties using **CCI-NotifyIncident**.

IMI-UpdateFaultInformation *is informed* about further information of a known fault,

Update fault information AND

Inform the responsible Contractor about the re-prioritised fault AND

Update (or remove) track restrictions using **IME-UpdateRestrictions** (or **IME-RemoveRestrictions**, respectively) AND

Inform concerned parties about the update *using* **CCI-UpdateIncidentInformation**.

TRI-RespondToIncident *is informed by* **IME-ImposeSuddenRestrictions** or **IME-UpdateRestrictions** (or **IME-RemoveRestrictions**) about the track restrictions and incidents, which have impact on the delivery of the train services,

Create amended train journeys for re-scheduled services and *inform* **Train Operator**.

Fig. 10. Reconstructed specification for the case study.

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 the "cleared()" method as FALSE (Condition2), Then send a message with the created "Fault" to "FixFault" owned by Contractor agent (Action2), and

Step3&4 {condition, action} couplet2: If the created "Fault" object is evaluated by the "immeImpact()" method as TRUE (Condition2.2), Then send a message with the created "Asset" to "ImposeSuddenRestrictions" owned by IME agent (Action2.2).

Step5: Add the belief that a fault occurs at this moment with a potential incident related with it.

Fig. 11. Brief steps of RR *IMI-HandleFault* processing by agent IMI.

[26]. The former requirement becomes a class function that helps the later requirement which becomes a RR to validate additionally required train journeys. Since "AcceptLateAddition" is a particular business task assignable to a single agent responsible for that task, it needs to be owned by the agent as a RR. Also because "ValidateTrainPlan" supports many RRs to function and not assignable to any particular responsible agent, it needs to be shared by many agents as a class method. Such a distinction differentiates business rules and business functions as described in the meta-model section.

The interaction of the example RR with other RRs and business classes can be documented in two design models, the Structural Model and the Behavioural Model shown

```
- <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>
```

Fig. 12. Reaction Rule *IMI-HandleFault* specification for the case study.

in Figs. 14 and 15, respectively, the basis for building Business Process Rules described in the next section.

In a Structural Model, 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. This regards agents as superior to classes, just like classes are regarded as superior to attributes in a Class Diagram. Agents are connected by "collaborate" lines, the collaboration among which is through one agent produces a message using one of its rules and another agent processes it using one of its rules. In the scenario, IMI collaborates with Contractor and IME using the rule *HandleFault* to fulfil its goal of handling fault. "Asset" and "Fault" are the business classes IMI uses to produce action messages for that collaboration. A message containing fault information will be sent to Contractor. As the definition of its rule FixFault indicates, it is its responsibility to process that message to fix the fault using a "Fault" business class. Likewise, IME uses its rule *ImposeS-*

1. Get a list of its managed rules that are documented in a XML rules repository, 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>.

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.

Fig. 13. Reaction Rule in XML processing procedures.

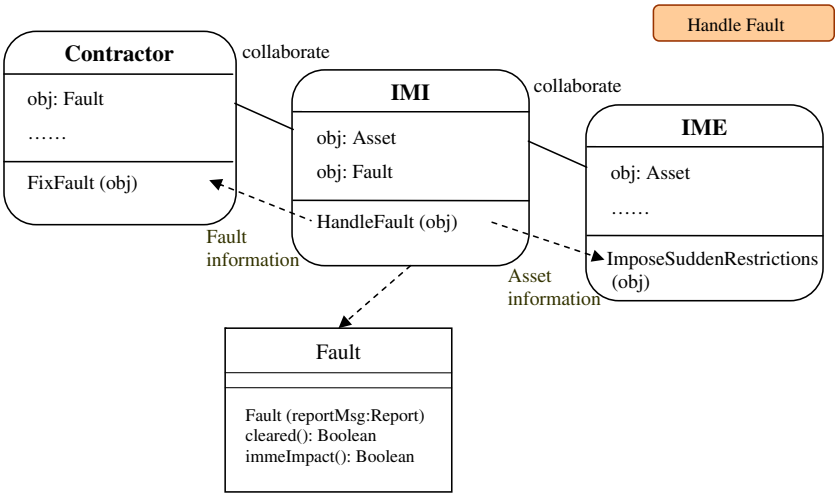

Fig. 14. Structural Model for the case study centred on *IMI-HandleFault*.

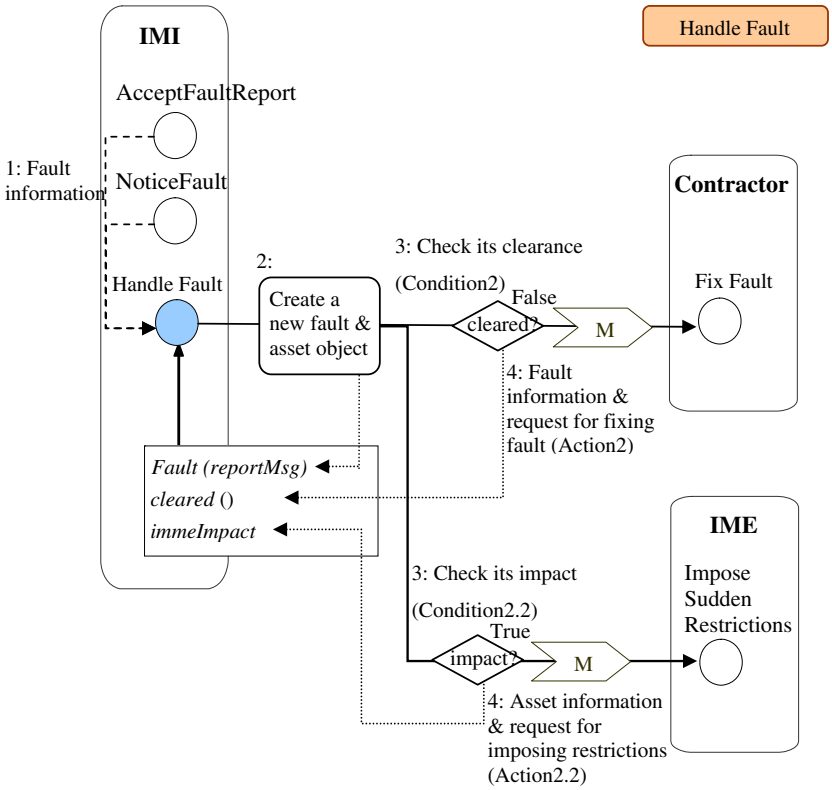

Fig. 15. Behavioural Model for the case study centred on *IMI-HandleFault*.

uddenRestrictions to process a message containing asset information to impose restrictions at a corresponding asset. The collaboration of the three agents towards a common goal of handling faults is captured in the RR models.

The Behavioural Model shows the same but from a perspective of capturing behavioural scenarios. It complements the Structural Model just like in the OO world Class Diagrams are complemented by Sequence Diagrams. As demonstrated in the design models, the use of business class components and the collaboration details among agents are in models and reconfigurable with rules. The XML definitions of the RRs are associated with the elements in the design models. This provides the agent a means to carry out computation using the models, and at the same time offers business people visual presentation and a method to modify models. The model configuration information is obtained by agents at runtime to ensure the deployment of the most up-to-date requirements. A business rule model architecture for agents is hence achieved.

The Reaction Rule language plays a similar role to other interface languages like the OMG's Interface Definition Language (IDL) [34]. IDL defines the interface only through which client objects can communicate with server objects in a distributed environment. This facilitates the encapsulation of the internal structure 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 [34]. The rule interface here 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 action message. However, that message is the event message 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.

3.7. Business Process Rule Model

Given the RR structure described in the previous sections, each RR has an internal processing component, and an external interface of event message receiving and action message sending, through which agents interact. The execution of collections of RRs following event message flow sequentially and conditionally forms business processes, combining inter-related Behavioural Models built upon Reaction Rule Models, and thus forming the blueprint of the system. These interconnected models constrain business processes and form higher level rules, termed Business Process Rule (BPR). Thus, one RR is about how a given task is to be performed following a process, being a goal internal to one agent, while a given BPR is about how one business is achieved by a process composed of interconnected RRs. A BPR thus delivers, a goal shared by many agents.

IMI-HandleFault is activated by the business process for managing new faults. It is one of a group of RRs that comprise the corresponding BPR, called "Manage New Fault". This BPR is shown in Fig. 16, with only the default conditions considered and assumed to be true for simplification. This is one of the two sub-goals of a higher level goal "Fault Management", as a result of the goal decomposition shown in Fig. 4. Two aspects are involved towards that goal: (i) new faults are reported and then handled and (ii) existing faults are handled in an alternative way or eventually removed as information about them is updated. The illustrated BPR extends the two Design Models shown in the previous section, connecting all related agents and their RRs that collaborate towards the shared goal.

The agent IMI initialises the above BPR using either of its two RR: "*IMI-AcceptFaultReport*" or "*IMI-NoticeFault*", in the interest of solving newly detected faults. The agents that finalise the BPR are Contractor and Train Operator, the completion of whose functions fulfils the goal of managing new faults. Fig. 17 shows the BPR "Manage New Fault" in XML, expecting a new fault as input/cause/pre-condition, and "fault fixed" and "train service re-scheduled" as output/effect/post-condition.

In the XML representation of BPR, Initial Agents (IAs) are referred to by those that initiate the BPR and Final Agents (FAs) are referred to by those that finalise BPR. IAs act spontaneously without request by other agents and FAs complete the BPR and request no further action of other agents. Intermediate agents participate in BPR between the activities of IAs and FAs during the execution of BPR. For every input from the IAs, output can be expected from the FAs, indicating that the goal of the business process has been accomplished. An IA is seen as the initiator actor and a FA could be a benefactor in use case terminology. As long as the input, output and goal of a BPR are all met, it is a black box and there is no need to be concerned about the selection/substitution of intermediate agents participating in the BPR, class invocation and decision making in individual agents or policy application. The publication of BPRs to external systems allows the invocation of them as services and so enables interoperability.

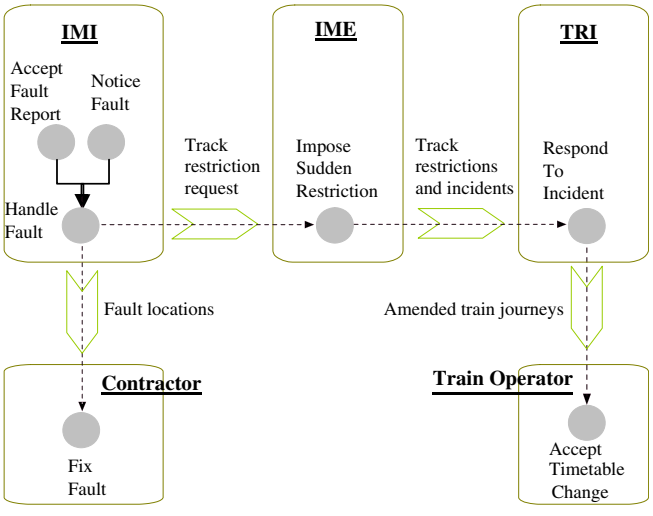

Fig. 16. Business Process Rule "Manage New Fault" for the case study.

The rule hierarchy of Business Process Rule – Reaction Rule – Policy Rule has been established. 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.

Both BPRs and use cases can be structured according to goals. During requirements analysis, responsibilities can be assigned to agents (in BPRs) or actors (in use cases), through the collaboration of which goals can be achieved. But BPRs have several advantages over use cases. A use case will fire business rules to constrain its process, if they are relevant in that use case context. BPRs act in a similar way as an integration unit of RRs. However, once a business rule is changed, all use cases that use the rule have to be changed, manually and individually. In contrast, BPRs are dynamically composed by RRs at runtime, and so such changes are accommodated automatically. Related to this, another important distinction is that BPRs can be used to automate the later development phases and remain reusable, rather than lose their value, like use cases typically do, when the modelling is completed. The insufficient semantics of use cases is even a barrier in its support for model refinement [15]. Moreover, BPRs are also an enhanced form of specification. The hierarchical structure of BPR-RR-PR eases the later mapping to agent knowledge in respect to architectural interaction, individual reaction process, and global policy. This is in contrast with use cases, where objects and their relationships are difficult to directly map to [35]. Finally, such a hierarchical rule structure used in AAM allows agents to automatically interpret the required knowledge from the configurable rule models, as opposed to the situation where object structure, method invocation, and interaction are all fixed after use case models are built and development is completed.

4. Model configuration and execution

We have presented business models, organised in the hierarchical levels of business processes, business rules, and business concepts, represented both in natural language and in XML. The former is business experts-oriented while the latter is Agent-oriented. Both are interchangeable via supporting tools [6,26]. Therefore, once business knowledge is configured by business people in their own language, the changed constructs can be transformed by the tools to corresponding XML portions that are analysable and executable by software agents. For example, a business concept of "fault" may be required with the schema given in Section 3.5 and the tool receives the field content, as filled in by the business people of this particular concept and puts this into the business model repository using

```
- <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>
```

Fig. 17. XML representation of the BPR "Manage New Fault".

XML (as shown in Fig. 5), ready for parsing by agents. When business people have their business needs changed, say, the addition of properties, the reconfiguration of the business concepts is via the same tool and this is reflected in the business models and the knowledge of the running agents immediately. Similarly, business rules and business processes can be described in a constrained natural language manner by business people to the supporting tools and corresponding XML representations formulated so that agents can parse and execute them for action, reaction, and interaction purposes. This includes the dragging and dropping of model elements such as agents, rules and messages provided by the tool and business processes can be constructed visually (as shown in Figs. 15 and 16), corresponding XML being generated. Both the people-friendly business models and the agent-friendly XML models are employed and complement each other. They are equivalent semantically but facilitate easy configuration by people and immediate execution by agents.

To demonstrate this further, consider how business experts can implement a new requirement. Currently, in the case study, infrastructure faults with no immediate impact will not cause sudden restrictions being placed in corresponding locations. Although this is reasonable, the delay of handling such faults in the long run could potentially cause harm. Suppose some arrangements are now required in association with faults without immediate impact. For the AAM approach the supporting tools for the configuration of models from repository can be used to configure models similar to those presented in Figs. 15 and 16. To deal with an additional fault handling requirement, they may add a rule to perform the actual business and link another agent requesting this agent to use the new rule, apart from the existing fault handling requirements as captured by other rules. In fact, this creates an association between IMI's HandleFault rule with a new rule of IME that is defined and assigned for the arrangement of the type of faults without immediate impact. A new message will also be created passing between the two involved agents. Model configuration as shown in Figs. 15 and 16 is straightforward. New model elements as rules, messages, and so on can be dragged and dropped into the tools. Even the reconfiguration of existing models is undemanding. For example, an additional (rule) processing phase can be inserted into two linked rule processing procedures of a business process by redirecting one existing rule to the new rule and the new rule pointing to the other existing rule with the original link removed. More radical requirements change could be, adding completely new business processes or embedding existing business processes in superior business processes. All these can be configured visually in tools, broken into reaction rules at individual agent level, and mapped to counterpart XML constructs. The completion of the model reconfiguration in a visualised form is the only thing business experts need to worry about and no IT intervention is needed in the due course. The tools will regenerate the corresponding XML representations of the models to the model repository. In our case, an additional condition and action pair will be put to the IMI-HandleFault's specification due to the additional requirement of handling ordinary faults without immediate impact. A business condition of fault not having immediate impact is put to the condition part. Directing a message to a new rule handling this type of faults is put to the action part. Complementing this, a new rule will be added to the IME for receiving events from IMI. The configuration of visualised models and conversion to the repository XML models is immediate, as is the interpretation and execution of the newly transformed models by agents. The business configuration and technical implementation and deployment are separated but lead to an integral and adaptive running system. Appendix A describes the model transformation process which is applied to provide a genuine Model-Driven paradigm.

4.1. Agent Model and implementation

Our AAM model example is implemented in the Java Agent DEvelopment Framework (JADE) [52,50], its

deployment being described in [44]. Nonetheless, multiple specific agent platforms should be capable of running the AAM knowledge model. Therefore, an Agent Model is used as a vehicle that drives knowledge model interpretation, by defining agent capabilities that need to be supported by a chosen platform. Since all the main agent platforms conform to the FIPA [51] standards in intercommunication and the models outline the agent capabilities without regard to a particular implementation environment, the common Agent Model is suitable for building any agent from any platform that wants to participate in an integrated AAM system. This on one hand maps knowledge model into agent behaviour in practice and on the other hand puts minimum constraints on the agent implementation phase so allowing a Platform-Independent model, as well as interoperability.

Three categories of class method have been suggested for method design: query, mutation and helper [37]. 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 AAM is listed 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. At runtime, environmental information is accommodated into business models and agents communicate with business models to handle business needs. Agents are subjects and business models are objects. The separation of agents and business models lets the externalised models, once get configured, the change gets reflected immediately in agents that interpret the models using the semantics of these acts, rather than fixed code.

Table 3
Agent role in AAM

Role name	Role function
Query	
GET	Query the incoming message queue and get new messages
COMPARISION	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

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 makes AAM technology independent.

4.2. Runtime Model-Driven Architecture and adaptivity in AAM

BPRs are business processes initialised by the IA, causing a series of agents to react using various RRs, the process finishing with the FA. In this course, an involved agent chooses a RR to react after an event occurs in a particular context, makes a decision, selects collaborators, and requests them to carry on the BPR. While a RR is functioning, a set of PRs may become relevant, so forming PR chains which are applied to assist the RR to make the decision or reflect business policies that must be enforced in that context.

A typical sub-process that makes up a BPR is shown in Fig. 18. Conceptual Model, Fact Model, PR Model, and RR Model are in coordination as the knowledgebase that drives an ordinary agent to behave, assisted by the FMA and the PRMA. The RR Model describing the agent execution of RR as discussed previously is now integrated with others.

The following is the runtime execution process carried out by a typical agent "Agent" using business models and runtime data, conforming to the agent acts in the abstract Agent Model described in the previous section. The application of it to the case study in that concrete situation is discussed below in Fig. 19 referring to Fig. 18. IMI agent and related BPR, RR and PR discussed previously will now work together.

Due to the externalisation of various rules, two distinct features of AAM models that contribute to the adaptivity of AMDA can be identified.

(i) There is no direct link between agents, so that agents are free to select collaborators. This is in contrast to the object-oriented paradigm, in which objects must be aware of where messages are passed to even though they are unaware of which objects will pass messages to them. Two-way encapsulation [36] required by full architecture independence is thus realised.
(ii) There is no direct link between agents and classes, so that agents are free to choose from a selection of such components.

Further, an important concern in software adaptation is that, when change of a certain element is required, whether or not it will affect other elements. Such consideration is useful in evaluating the adaptivity of architectures. The

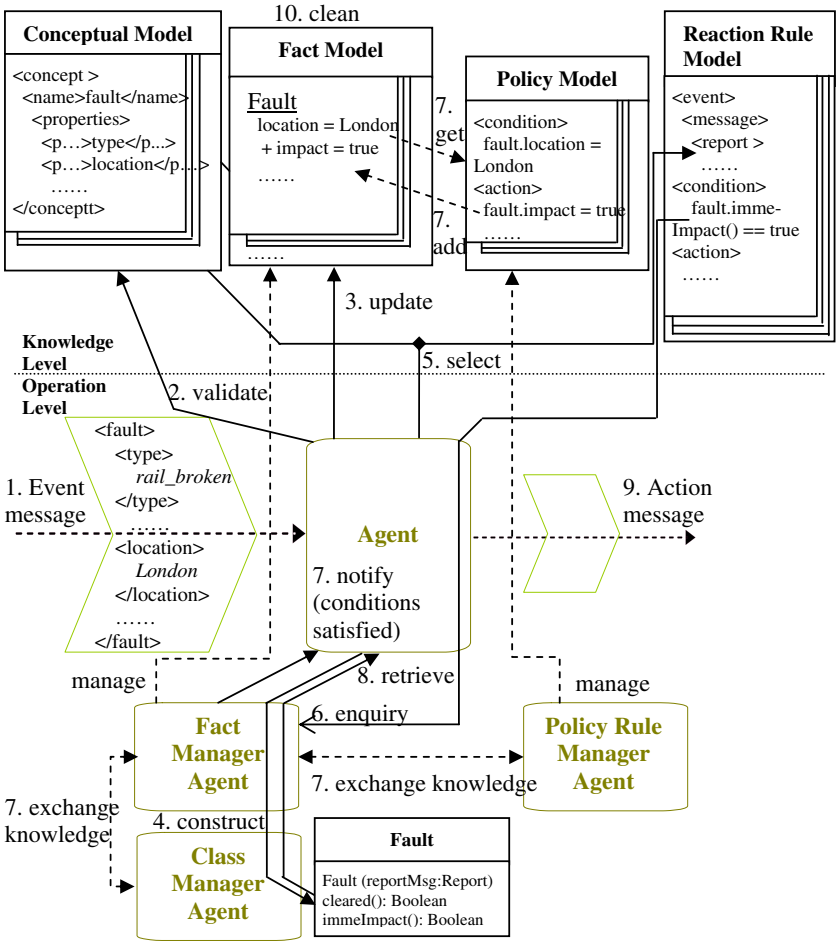

Fig. 18. AAM runtime architecture.

AAM can be used to minimise interdependencies among system elements. Table 4 compares the impact of each single type of typical required change on overall systems developed using AAM and traditional OO systems. Considering this table, there is strong inference that the AAM approach is better than the current known approaches in adapting many aspects of software systems. In AAM, changes to one aspect have less impact on the other aspects of a system and less effort is required to accommodate the changes. With AAM the possibility is that all post-delivery efforts relating to business requirements change can be carried out by business people, developer intervention not being required.

To achieve adaptability in traditional software development, all possible scenarios must be prescribed before the development of a system starts. However, very often limited knowledge or control over the environment can be obtained, and not all contingencies can be anticipated [49]. Hence a pre-determined design results in an implementation of the system which later resists the emergence and accommodation of new conditions and new actions. Such a situation is largely due to the lack of semantics supported in existing modelling languages (UML, AUML, etc.) and the fixed structures and behaviours required by existing programming language constructs (objects, etc.).

Agents have the capability to represent components that have no perfect knowledge but, rather, an engine that uses an extensive knowledgebase to dynamically perform changing tasks. They are components that allow flexible system architecture. With agents there is the potential that something unknown or uncertain can be fulfilled in the future or replaced by updating knowledge. In AAM, agent is an abstraction at the business level and business models are the actual maintenance target that can incorporate new knowledge and contingencies, whenever they become available. Agents commit no restriction to structure, behaviour, relationship, decision making and so on. Instead all these are part of the configurable business models controlled by those who know what they should be and how they should change at different times for different purposes. All in all, the combination of agents and business models is the natural representation of the business needs and is at the right level where the maintenance should be carried out.

5. Evaluation and conclusions

Both the Agent-oriented paradigm and MDA are powerful tools in Software Engineering [42] and if combined, such a methodology as proposed in this paper can make the development of complex systems easier and cheaper

Step1: Agent plays its GET role and gets an incoming message from its incoming message queue. (A fault is reported to IMI.)

Step2: Agent plays its COMPARISION role and validates the encoded object structure using the Conceptual Model. (The "fault" structure encoded in the message matches with the one defined in CM.)

Step3: If the object structure is equally defined in the Conceptual Model, then Agent plays its INITIALISATION role and populates the object structure in the Fact Model 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.)

Step4: 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.)

Step5: Agent plays its SELECT role and finds the Reaction Rule from the Reaction Rule 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.)

Step6: Agent plays its ASSERTION role and checks if conditions specified in the Reaction Rule are satisfied using the Fact Manager Agent. (Facts in FM are looked for in relation with the conditions of the RR.)

Step7: In this process, the interactions of Fact Manager Agent with Policy Rule Manager Agent 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.)

Step8: Agent plays its FACTORY role and produces a message as the result of the action coupling with the satisfied condition as defined in the Reaction Rule. 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.)

Step9: Agent plays its SET role and puts the message to its outgoing message queue.

Step10: Agent plays its FINALISATION role. Temporary facts are demolished, and Fact Model knowledge is restored its original state.

Fig. 19. AAM runtime process.

to align with changing business needs. Before considering the contributions of AAM, some important similarities and differences with other approaches in the use of an agent notion are pointed out.

The notion of agent in AAM does not include full autonomic characteristics as increasingly found in Agent-oriented Software Engineering research. Full autonomy by definition could lead to unpredictable agent behaviour, and uncertain interactions and outcomes. In order to guarantee the fulfillment of system goals, especially important to many mission critical systems, agent behaviour has been restricted by business models. This allows reliable services to be provided by agents, according to reconfigurable models. Other fundamental characteristics of agent are still preserved, e.g. they are reactive and maintain their own threads of control.

Moreover, the concept of agent in requirements modelling and implementation should be distinguished. Agents during the requirements phase are used to organise the requirements. When implemented as software, they are responsible for meeting their corresponding requirements. We refer to agent not as a single notion such as that of the Belief Desire Intention (BDI) agent [61]. Since the aim is an integrated model driven development process, agents in AAM have different but highly related meanings in different phases. The moving from requirements to design and finally implementation using AAM implies the transference of the agent notions and model transformation. We start from an agent notion suitable for capturing and managing requirements, similar to that in the i^* [60] model. Following this, agents for design purposes appear in design diagrams along with object components (such as those shown in Figs. 14–16), realising the responsibilities of corresponding requirements agents. These agents are similar to those of the Gaia methodology by Wooldridge and Jennings, agents being accompanied by roles in designing MAS. Finally the models drive the behaviour of running software agents at runtime (such as shown in Fig. 18) in the sense of which the notion is close to that of JADE, being a MAS implementation platform. Seam-

Table 4
Comparison of ripple effects of changing business needs on AAM systems and traditional systems

Adaptation target	Adaptation	Impacted AAM element	Impact on AAM systems	Impact on traditional systems
Concept	Addition	—	Policy Rules that refer to altered Business Concepts may have to be changed, or even invalidated (if referring concepts do not exist)	New classes must be developed, or existing ones modified
	Modification	Policy Rule	Corresponding business classes may have to be redefined	
	Deletion			
Policy	Any change	—	None (changes will be accommodated automatically)	All system components must be inspected to find code relevant with the particular changed policies
Event	Addition	Reaction Rule	New Reaction Rules must be defined to deal with unforeseen events	New class methods must be added in response to the new events
Decision making	Modification	Reaction Rule	Reaction Rules must be changed of their use of {condition, action} pairs in tree	Fundamental business logic must be re-developed involving comprehension of existing code and tremendous re-implementation
			Partnerships may be changed as a result of changed actions in delivering computational results	
			Business classes may also be used differently	
Partnership	Addition	Business Process Rule and Reaction Rule	Business Process Rules must be re-configured	Component interfaces must be redesigned. Effect is enormous, as many components have to be changed in the way of communication
	Modification		Reaction Rules may be added, changed, or removed. In the typical case of modification, their "event" and "action" components are affected, reflecting required new collaborations	
	Deletion			

lessly, the variations of the agent notion used in our approach transfers from one to another, an indication of the progress of the software development in a model-driven fashion. The remainder of this section will enumerate the contributions of AAM.

5.1. Adding component dynamics to MDA

Since normal objects have static structure and behaviour, system adaptivity is inherently hard to achieve. In the agent world, no notion of model-driven agent architecture exists yet. On the contrary, available agent platforms actually constrain agent system developers in prescribing agent behaviour in agent classes during implementation, an inappropriate practice inherited from OO system development. For example, in the popular JADE agent platform, Interaction Protocols (IPs) described in Agent UML (AUML) [53] are used to represent agent conversations in message sequences for agent system development. IPs are specified using informal notations and are manually interpreted by agent developers into program code [41]. This means that: which agents are in partnerships; where messages are expected to be received from and sent to; and what the processing procedures are to be taken in response to events are all static, once the development is completed. Related work in an attempt to execute Agent-oriented UML models includes Plug-in for Agent UML Linking (PAUL), which attaches application-specific code to the appropriate points of the protocols for agent execution [41]. PAUL recognises that the possible sequences of messages that form agent conversations constrained in IPs need to be interpretable by agents. It uses specific operations that instruct agents the methods to receive messages (as operation parameters) and send messages (as operation outputs), which adds some though limited behavioural semantics to the existing IPs. PAUL has similarities with our approach in the sense that agent communication, in particular the message producing and processing as stated in their interaction models has been made explicit for agent execution, where in our approach rules are abstracted for agents instead of operation code. However, due to the use of separate code fragments attached to IPs in PAUL, the management and maintenance of them adds to extra burden to the existing system. Moreover, the approach does not support to execute interactions between more than two lifelines and agents cannot change roles. A further important weakness is its use of Java statements commits the method to a specific platform. AAM addresses the above problems by associating XML-based process-able rules with agents to abstract agent functions in a comprehensive but manageable manner; allowing the definition of additional rules for agents to make it scalable to accommodate any number of agent involvement and agent roles into the system; capturing in rules the invocation of generic classes and methods to ease management so that developers can select any platforms to run the system.

The result is that, AAM breaks the traditional constraint of static component structure and behaviour found in practice in both the object world and the current agent world, without committing to any specific runtime platforms. It allows agents to retrieve and interpret their behaviour with regard to all these aspects dynamically at runtime, entirely driven by configurable business models.

5.2. Agent-executable rule-based business models

This work contributes a set of agent-executable rule-based business models which are independent from platforms or application domains. Their employment is intended to serve as PIMs (Platform-Independent Models). Like PIMs, the models represent the business to be eventually supported. In the normal sense, business models capture business requirements but have no capability/need to describe the constructs, relationships, and architectures of the software system to be delivered, while the goal in creating the PIMs is to generate a high-level dictionary for the project that captures abstract concepts and their relationships [12]. Thus, business models are more requirements-oriented whereas PIMs are design-oriented. However, we have explicitly used UML and XML together to represent business abstracts (processes, rules, concepts, etc.) where business requirements and design structures are integrated in a single set of models. Precisely, in the top level, business process rules capture the required business processes (business requirements level) in a form that fits in the overall system architecture (software design level), as shown in Figs. 16 and 17. The graphical UML model (Fig. 16) and the textual XML model (Fig. 17) together capture the semantics of the business requirements that are presented generically enough and at the same time with sufficient details for guiding the behaviour of agents (interaction sequences in particular), no matter what platform-dependent agent platforms will be chosen afterwards. Similarly, the reaction rules and policy rules sitting in the middle level capture the required business rules (business requirements level) in a form where the constructs of which pull agents together in interactions, inform agents of their internal computations and decision makings individually, as well as policy applications globally (software design level). Finally, business concepts in the bottom level capture vocabularies of the models (business requirements level) in a form that is associated with the fundamental entities used by the system, facilitating the establishment of concrete facts and mapping to business objects (software design level). Therefore, the hierarchical models that our approach aims to build, not only capture information at the business requirements level, but also have sufficient details about the constructs, relationships, and architectures in their combined UML and XML formalisms as required at the software design level. The richness of information captured at both levels as well as their seamless integration by our modelling approach attributes largely to the semantic-rich UML and XML within the Business Knowledge Models and its direct association with the Agent Model.

Some proponents of MDA propose the specification of an interface language such as OMG's Interface Definition Language (IDL) plus some constraints for PIM, while others propose the Executable UML [43]. The AAM's use of rules is equivalent to an interface language plus constraints. However, such a model is also comprehensive in terms of its covering of all business aspects including processes, reactions, policies and concepts. Nevertheless, at the same time it is an integrated rule-oriented model rather than a set of separated interfaces and constraints.

Aiming at being a form of PIM, Executable UML relies on Action Semantics (AS) [38], adopted as an integral part of UML. AS provides specification for actions, such as create/select/delete object, write/read attribute, and relate/select/unrelated objects across association. Object Constraint Language (OCL) is also used to complement UML for expressing semantics. It can express rules that place logical constraints on elements in models, such as invariants, pre-conditions, and post-conditions. AS overlaps with OCL in the semantics they can convey. AS and OCL can be used in combination or as alternatives, both being useful to bridge the semantic gap from high-level UML models to low level code constructs.

Object-oriented MDA can use UML plus OCL/AS to complement the UML's lack of formal foundation on some constructs, e.g. transition guards or method bodies [39]. Such insufficiency could lead to incomplete or inconsistent UML models and error-prone interpretation by developers during implementation, if UML is used alone. Even when OCL or AS are added, together with modelling tools that are capable of code generation, how to insert the right code in a chosen language at the right place of the code skeleton in accordance to the expressed statements remains an issue, which demands developers' good understanding of the target programming language. Actually, these action languages do not significantly raise the level of abstraction above that provided by programming languages [55]. The interpretation of the statements in OCL/AS into specific languages requires the understanding of both a semantic-oriented language and a programming language and the mapping between them is manual, error-prone, and requires extra efforts. Defining a language that uses programming language primitives to build high level constructs can significantly enhance abstraction level and thus support MDA [55].

Our modelling paradigm of AAM is at a business level, the invocation of lower level class methods is abstracted away from the core business needs, and modellers can focus on business logic in modelling. The modelling as such leads to the association of business constructs in models at a very high level of abstraction (concepts, rules, processes, etc.). They embrace editable syntax and understandable semantics for business people. These high level abstractions

together capture sufficient business knowledge about the system under development but are technology-independent. They are later transformed into XML formalisms, capturing the knowledge understandable to agents and guiding their behaviour including: agent interaction processes; agent internal computations and decision makings; and low level component invocations. Such computing details and the employed techniques are hidden away from the business modellers, the business logic being the only knowledge input necessary from them. Even so, the input of the knowledge implies the agent abstractions as entities carrying out the business and the component abstractions as entities facilitating agents with fundamental supporting functionalities but their implementations left to the technical people's decisions and fully configurable. Later (swappable) agents and components are auto-assembled and running in the system. This is made possible thanks to the rich knowledge captured in the business logic (with references to them as well as capability requirements) and the freedom of technical specification (without specific implementation requirements to them) in separation of business modelling. Agents and components can be switched to other platforms or implementations as long as they meet the business needs and fit in the business models.

When the business requirements must be changed, these are mapped into entities in different levels of the BPR, RR, PR, Business Concept hierarchy. This is in contrast with the traditional modelling situation where partial or complete redesign is needed when a new requirement emerges. The underlying supporting objects are a separate concern. Business models only specify the abstract business objects and their methods required by the system but do not entail their specific language features. The common class/method paradigm in modern OO languages allows us to initially avoid commitment to a single language. The use of programming languages selection is free, as long as their functionalities satisfy the needs expressed by the business models. For example, a class method invocation for a rule condition evaluation is not a concern when we establish our models. It is in later phases that concrete classes come into consideration. They could come from a reusable class library. The change of available class versions and the way they are used can dynamically bring new effects to systems to reflect new business needs as the invocation of them is externally modelled to drive dynamic agent behaviour. Obviously, not only the programming languages but also the agent platforms need not to be specified when one is building the models, as an abstract Agent Model has been used. Consequently, the overall component architecture in AAM is technology-independent.

One concern that might be raised is the possibility of conflict between models. In fact, the proposed modelling approach assists engineers and modellers in detecting requirement conflicts. Conflicts among business processes are eliminated using the goal-decomposition graph. Since goals map to processes, we can assume processes are compatible with each other. Further, concepts are atomic entities and have no means to disagree with each other. Reaction rules distributed in different agents cannot conflict with each other since disparate agents make distinct reactions without affecting each other. Therefore, two potential types of conflicts exist, among two or more policy rules that globally applicable to all agents, or among two or more reaction rules that locally applicable to a single agent. In the former case, a solution is found by looking for those policy rules that have the same satisfaction conditions but different actions required to be triggered as the result of satisfying conditions. In the latter case, a solution found by is looking for those reaction rules that are for the same agent, have the same events for triggering the rules, but guide that agent to perform conflict decision-making processes. Therefore if rules are well managed and categorised in our knowledge repository, say for example, under the same conditions (for policy rules) or under the same agent ownership and event triggering conditions (for reaction rules), then the detection of conflicting rules can be almost automatic, this being an avenue for future work.

5.3. Raising the level of abstraction

The use of a set of constructs with direct business meanings such as business process, business policy and business partnership and their direct agent interpretation is an important advantage of AAM. Similarity, the externalisation of low level object component usage is a key technique enabling our architecture to be driven by models in the agent context. An important lesson learned in Model-Driven Architecture from past experience is that models should be used to abstract selected elements of the implemented complex systems rather than replicate the abstractions that programming languages provide [40]. Moreover, it is recognised that existing UML models alone are not sufficient to generate complete implementation, not only program skeletons, but also fine-grained functionality. MDA intends to raise the level of abstraction for building systems and reduce complexity, just as 3GL replaced assembly language and design patterns replaced the need to rewrite code many times [12,55]. A language abstraction is usually proposed when an existing method becomes cumbersome and a new one is needed for modelling systems. The introduction of additional language concepts is more manageable and reusable than design patterns [54]. The object concept is not now the best candidate at the abstraction level of the new MDA paradigm and should be replaced by the higher level abstraction of agent.

This does not mean objects should disappear. Rather, they remain playing the important roles of supplying fundamental business functions and facilitating agents to behave with those functions. Agents make use of them dynamically, on demand. However, these static components are moved down to a lower level of abstraction, below the agent abstraction which is associated directly

with business abstracts. These abstracts are the complementary abstraction to agents in the new MDA paradigm. Business experts now only need to configure abstracts such as business processes or rules in the context of agents, assuming components underpinning agents will support specific functions during the execution of processes or rules by agents. The configurability of both business abstractions and available components with pre-assumed capabilities let business experts to define their business in terms of both business logic and business power. The specific implementation of either components or agents is not an issue for business experts at model definition time, but up to technical people, in a separate decision process. Eventually, agents as a component abstraction, accompanied by a model abstraction employed by our approach, together raise the abstraction level of traditional (fixed) computing model. In this way, the AAM provides configurability (via the model abstraction) and adaptivity (via the component abstraction).

Accompanying agents, rules are also introduced to become a core model element. Another relevant approach proposes roles as a basis to allow transforming agent behaviour from models to code [42]. However, the PIM given with agents and roles cannot fully capture the business models of a system. This makes the modelling method hard to generically represent roles and the corresponding code generated from roles, which in turn prohibits a feasible solution for producing a functional system in the real world. In contrast, rule model elements in the agent context capture comprehensive aspects of business semantics. The direct association of agent processable XML code to rules rather than separate OCL/AS statements to various other model elements is an advance. The mapping from rules to agent acts instructs agent to behave both in high level decision making in terms of overall architecture and low level business object usage during their individual function. It is this new element established in an agent context sets the approach apart from other modelling approaches towards MDA.

5.4. Future work

Future work will look further into the AAM business models and evaluate them in terms of their comprehensiveness in capturing business knowledge. This will be justified if they can indeed capture all sorts of knowledge in the proposed knowledge hierarchy and if not, the models will be refined so that they can represent other complementary types of business requirements.

We will also define transformations from the AAM models to a selection of agent platforms such as JADE (PSM) and then code. This is concerned about the mapping from agent acts defined in the Agent Model operating upon business models to specific agent behaviour constructs specified in various agent platforms. Since the Agent Model and Business Models we use can cover business requirements in various levels and highly associated with agent execution model, mapping to specific agent platforms is straightforward. The definition of a mapping from agent act keywords to platform-specific constructs and an engine for rule execution will suffice. The former is an issue on direct mapping from key words to specific functions, while the latter has already been partly implemented, as our developed modelling tools are used and such engines as presented in [6,50]. AAM provides a general business-oriented, Platform-Independent Model, and the need of a complex model transformation is turned into the need of a rule engine, solutions of which already exist. Both the generic PIM model building and the model transformation definition were the difficulties faced by the (object-oriented) MDA [12] and must be taken into consideration if one is about to adopt the paradigm.

Finally, the current XML format of rules could degrade the performance of agents and cause a bottleneck in the system. This is due to the fact that every time when an agent is to behave, it looks for the appropriate rules and interprets from them the current required behaviour at runtime. Although such a mechanism brings about runtime Model-Driven Architecture and associated adaptivity to the AAM it could slow down the whole system. In addition to parallel computing architecture and powerful hardware investment, we will seek solutions to alleviate this, including pre-building object components for agents and cache technology.

These future research areas will further strengthen the AAM and AMDA and provide an important contribution to the combined areas of model driven development, agent oriented software development, business modelling and the larger area of software evolution.

Appendix A. AMDA Model transformation

Here we will illustrate the transformation from CIM through PIM to PSM in AMDA.

A.1. CIM to PIM

The transformation from the CIM starts with the goal-decomposition graph as shown in Fig. 4 along with functional requirements tables (Table 2) and declarative policy descriptions. A goal or sub-goal will be directly mapped to a business process rule (Figs. 16 and 17) that realises it, each consisting of multiple inter-connected reaction rules and policy rules and thus shaping up the PIM. These rules will be mapped, respectively, from the functional requirements tables and declarative policy descriptions (an example of this is underlined in the case study description). The vast majority of functional requirements are transformed to a precise specification of the reaction rules in XML (e.g. Fig. 12), the table sections mapping to XML tags and section contents mapping to XML tag contents. The following transformation rule is applicable.

```
Transformation Definition FunctionalReqTabToReactionRule (FunctionalReqTab, ReactionRule) {
  source f: FunctionalReqTab;
  target r: ReactionRule;
  source condition 1
    f. "Sub-Reqt of" = NULL
  mapping
 r. Event. messageFrom := f. Cause. getActor ();
 r. Event. messageFromContent := f. InformationUsed. getContent ();

 r. processingMethod := f. RequiredEffect. getProcessingMethod ();
 r. Processing := (f. RequiredEffect. productions = r. processingMethod (r. Event. messageFromContent));
 inv: Sum (r. processingMethod (r. Event. messageFromContent))) == f. Outputs;

 switch (f. RequiredEffect. Case (i)) {
    Case (i): {
    r. Condition := f. RequiredEffect. Case (i). getConditionStatement ();
    r. Action. messageTo := f. RequiredEffect. Case (i). getActor ();
    r. Action. messageToContent := f. RequiredEffect. Case (i). getContent (); }
 }

  source condition 2
    f. "Sub-Reqt of" ≠ NULL
  mapping
 Add f to f. superRule (). facilitatingBusinessClasses (f). processingMethods ();
}
```

To illustrate, the functional requirements table "Handle-Fault" (shown in Table 2) will be transformed to a rule as it does not have a "Sub-Reqt of" section (If it had this it would be a sub-requirement and so would be transformed to a business function for facilitating agent rules). The table has a "Cause" section, triggering the rule function. This is mapped to an "event" coming from the "IMI" agent ("A fault becomes known... from... IMI-..." in the "Cause" section), in this case the event message being a report about a fault, including infrastructure assets, their contracts, and so on ("Information about infrastructure assets and their contracts..." in the "Information Used" section). On receipt of an event, a rule processes it by using the "Information Used" and constructs some output that will be used later, thus providing the rule's "processing" component. In this case, the report message is decoded and the fault and its associated asset created as new business objects ("The fault is recorded" in the "Required Effect" section). The productions will be useful for handling the reported fault ("Fault information to contractors" in the "Outputs" section). A rule will react differently in different conditions to achieve different required effects. In one condition ("Unless the fault has already been cleared" in the "Required Effect" section), the created "fault" ("Outputs") object is encoded into a message, and sent off to the contractor ("the appropriate contractor is identified...for fixing the fault"). As such, pairs of associated "condition" and "action" components are created for the rule. In another condition where the fault has immediate impact, track restrictions are requested to be placed ("If necessary, track restrictions are put in place... there is an impact" in the "Required Effect" section).

Declarative policy descriptions from the requirements description are mapped to a precise specification of policy rules in XML (Fig. 6). The transformation is defined as follows:

```
Transformation Definition PolicyDescriptionToPolicyRule (PolicyDescription, PolicyRule) {
  source p: PolicyDescription;
  target r: PolicyRule;
  mapping
 r. Condition := (p. getConditionClause (). Subject. subjectAttribute = p. getConditionClause (). AttributeValue);
 r. Action := (p. getActionClause (). Subject. subjectAttribute = p. getActionClause (). AttributeValue);
}
```

To illustrate, take the policy rule from the case study description, *"if the fault is located at capital cities, it has impact and needs to be fixed immediately"*. Its condition clause will be mapped to a policy rule condition which describes that the fault subject has certain location attribute values: *fault.location = "London" OR "Edinburgh" OR "Cardiff" OR "Belfast"*. Likewise, its action clause will be mapped to: *fault.immediate_impact = true*.

Together, these transformation rules will turn the requirements model to the Platform-Independent rule-based model. Reaction rules being interconnected and linked with policy rules, they compose business process rules (Figs. 16 and 17) and realise goals (Fig. 4).

A.2. PIM to PSM

To derive the PSM reaction rules and policy rules in XML are mapped to knowledge partitioned by XML tags, populated to agent internal processes along with the XML tags for conditional or sequential constructs, referred to by agents procedurally for execution at runtime. Agent acts are mapped to query, mutation, and helper construct func-

tions to be operated upon the runtime data using the populated knowledge.

The iterative rule selection and application process used by running agents is illustrated in Fig. 13. Procedurally, an agent uses the "⟨event⟩ – ⟨message⟩ – ⟨from⟩" sub-structure for receiving messages from another agent, the ⟨processing⟩ for method invocation, the "⟨condition⟩ – ⟨action⟩" pattern for asserting satisfactory circumstances and performing associated acts, and "⟨action⟩ – ⟨message⟩ — ⟨to⟩" for sending messages. Runtime data will be populated into the structured rules, within individual sections of XML annotations, guiding agents to behave sequentially and conditionally, upon any specifically chosen platform. The structure of platform-dependent agent behaviour will also be in a certain structure mapped from the rule scheme. Being structured deterministically, a rule instructs an agent to proceed along its running process using certain platform-specific constructs when meeting certain XML tags, parameter values of the constructs being the tag contents encoded within.

```
Transformation Definition ReactionRuleToAgentBehaviour (ReactionRule, AgentBehaviour) {
  source r: ReactionRule;
  target b: AgentBehaviour;
  mapping
 b. addToAgent (r. getOwnerAgent ());
 b. Declaration. VariableType := r. getGlobalVariables (). getVarType ();
 b. Declaration. VariableName := r. getGlobalVariables (). getVarName ();
 b. setIfClause (r. getEvent (). getMessage (). getFromAgent (). equals (getMessageSenderAgent ()));
 b. setThenClause (b. DecisionMakingProcess);

 b. Invocation := (r. getProcessing (). getOutput () = r. getProcessing. getInput (r. getEvent (). getMessage ()));
 b. Decision := (
  switch (r. getDecisionTree (). Case (i)) {
    Case (i): {
    b. setIfClause (r. getDecisionTree (). Case (i). getCondition ());
    b. setThenClause (r. getDecisionTree (). Case (i).
       getAction (getMessage (). getToAgent (), b. Declaration. VariableName));}
  }

  )
  b. DecisionMakingProcess := b. Invocation + b. Decision;
 }
```

Carrying out this transformation upon the reaction rule of "HandleFault" will produce a behaviour for the agent IMI, the owner agent section of the XML specification indicating this. In the behaviour, variable types of Asset and Fault will be declared. Instances of these will be manipulated by agents at runtime upon the selected platform. An "If" clause will be used by the agent behaviour for judging if a received message is to be processed by the current rule, its "⟨event⟩ – ⟨message⟩ – ⟨from⟩" sub-structure being retrieved for the purpose. If so, then a decision making process will be carried out as follows. Immediately, the rule's ⟨processing⟩ sub-structure maps to the invocation methods an agent needs to execute, involving the already declared variables. These may be results of interest to other agents which will be known in the coming decision phase. In this case, a concrete fault object will be constructed from a report message and ready to be made known to any interested party. The decision process consists of multiple "If" and "Then" clauses. Their production will be mapped from the rule's "⟨condition⟩ – ⟨action⟩" couplets sub-structure. Whenever a condition in the decision-making tree is discovered to be met, an associated action will be executed. This involves sending a message with variables of interest to a destination agent, being indicated by the "⟨action⟩ – ⟨message⟩ – ⟨to⟩" sub-structure. For example, if a fault is found not been cleared, then it will be sent to a contractor for fixing it. The pseudo-code transformed under the definition can be found in [62]. The transformation of policy rules can be achieved using simple "If" and "Then" clauses and so omitted here for conciseness. Consequently, the application of both sets of transformation rules results in the production of the PIM from the CIM, and further the production of the PSM from the PIM.

References

[1] S. Demeyer, S. Ducasse, O. Nierstrasz, Object-Oriented Reengineering: Patterns and Techniques, in: Proceedings of the 21st IEEE International Conference on Software Maintenance (ICSM'05), 2005, pp. 723–724.

[2] M. Fayad, M. Cline, Aspects of software adaptability, Communications of the ACM 39 (10) (1996) 58–59.

[3] E. Gamma, R. Helm, R. Johnson, J. Vlissides, Design Patterns, Addison-Wesley, 1994.

[4] M. Wermelinger, G. Koutsoukos, R. Avillez, J. Gouveia, L. Andrade, J.L. Fiadeiro, Using coordination contracts for flexible adaptation to changing business rules, in: Proceedings of the Sixth International Workshop on Principles of Software Evolution, IEEE Computer Society Press, 2003, pp. 115–120.

[5] J.W. Yoder, R. Johnson, The adaptive object-model architectural style, in: Proceedings of the Third IEEE/IFIP Conference on Software Architecture: System Design, Development and Maintenance, 2002, pp. 3–27.

[6] L. Xiao, D. Greer, The adaptive agent model: software adaptivity through dynamic agents and XML-based business rules, in: Proceedings of the Seventeenth International Conference on Software Engineering and Knowledge Engineering (SEKE'05), Taipei, Taiwan, Republic of China, 14–16 July 2005, pp. 62–67.

[7] P. Boinot, R. Marlet, J. Noye, G. Muller, C. Consel, A declarative approach for designing and developing adaptive components, in: Proceedings of the Fifteenth IEEE International Conference on Automated Software Engineering, 2000, p. 111.

[8] M. Fowler, UML Distilled: A Brief Guide to the Standard Object Modeling Language, third ed., Addison-Wesley, 2004.
[9] T. Morgan, Business Rules and Information Systems, Addison-Wesley, 2002.
[10] Object Management Group, Inc., 250 First Avenue, Suite 100, Needham, MA 02494, USA.
[11] A. Kleppe, J. Warmer, W. Bast, MDA Explained: The Model Driven Architecture: Practice and Promise, Addison-Wesley, 2003.
[12] T. Meservy, K. Fenstermacher, Transforming software development: an MDA road map, IEEE Computer 38 (9) (2005) 52–58.
[13] W.S. Ambler, L.L. Constantine, The Unified Process Inception Phase: Best Practices in Implementing the UP, R&D, CMP Books, 2000.
[14] J. Warmer, A. Kleppe, The Object Constraint Language: Getting Your Models Ready for MDA, second ed., Addison-Wesley, 2003, 2003.
[15] D. Thomas, MDA: revenge of the modelers or UML Utopia? IEEE Software 21 (3) (2004) 15–17.
[16] S.H. Yim, J.H. Ahn, W.J. Kim, J.S. Park, Agent-based adaptive travel planning system in peak seasons, Expert Systems with Applications 27 (20) (2004) 211–222.
[17] Z.H. Jia, K.S. Ong, J.Y.H. Fuh, F.Y. Zhang, 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] T.W. Goh, 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.
[19] I. Lovrek, G. Jezic, M. Kusek, I. Ljubi, A. Caric, D. Huljenic, S. Desic, O. Labor, Improving software maintenance by using agent-based remote maintenance shell, in: Proceedings of the 19th IEEE International Conference on Software Maintenance (ICSM'03), 2003, p. 440.
[20] O.B. Kwon, J.J. Lee, A multi-agent intelligent system for efficient ERP maintenance, Expert Systems with Applications 21 (4) (2001) 191–202.
[21] S.N. Murphy, U.H. Rabbani, G.O. Barnett, Using software agents to maintain autonomous patient registries for clinical research, in: AMIA Annual Fall Symposium Proceedings, 1997, pp. 71–75.
[22] I. Jacobson, S. Bylund, The Road to the Unified Software Development Process, Cambridge University Press, Cambridge, 2000.
[23] A. Lamsweerde, Requirements engineering in the year 00: a research perspective, in: Proceedings of the 22nd International Conference on Software Engineering (ICSE '00), 2000, pp. 5–19.
[24] L. Xiao, D. Greer, A hierarchical agent-oriented knowledge model for multi-agent systems, in: Proceedings of the Eighteenth International Conference on Software Engineering and Knowledge Engineering (SEKE'06), 2006, pp. 651–656.
[25] B. Bruegge, A.H. Dutoit, Object-oriented Software Engineering: Using UML, Patterns and Java, second ed., Prentice Hall, 2004.
[26] L. Xiao, D. Greer, The Agent–Rule–Class framework for multi-agent systems. International Journal of Multiagent and Grid Systems 2 (4) (2006) 325–351 (Special Issue on Agent-Oriented Software Development Methodology, IOS Press).
[28] V.K. Rai, C. Anantaram, Structuring business rules interactions, Electronic Commerce Research and Applications 3 (1) (2004) 54–73.
[29] L. Xiao, D. Greer, Software adaptivity through XML-based business rules and agents, in: Proceedings of the PREP2005, Lancaster, UK, 30th March–1st April, 2005, pp. 287–288.
[30] S.L. Pfleeger, J.M. Atlee, Software Engineering: Theory and Practice, third ed., Prentice Hall, 2005, 2005.
[32] D.M. Dikel, D. Kane, J.R. Wilson, Software Architecture: Organizational Principles and Patterns, Prentice Hall, 2001.
[33] P. Oreizy, N. Medvidovic, R.N. Taylor, Architecture-based runtime software evolution, in: Proceedings of the 20th International Conference on Software Engineering (ICSE'98), Kyoto, Japan, 1998.
[34] Object Management Group, Inc., CORBA 3.0 – IDL Syntax and Semantics Chapter, OMG Document Formal/02-06-07, 250 First Avenue, Needham, MA 02494, USA, 2002.
[35] A. Korthaus, Using UML for business object based systems modelling, in: M. Schader, A. Korthaus (Eds.), The Unified Modeling Language – Technical Aspects and Applications, Physica-Verlag, Heidelberg, Germany, 1998, pp. 220–237.
[36] J. Hogg, Applying UML 2 to model-driven architecture, in: Proceedings of OMG Workshops: MDA Implementers' Workshop, 2003.
[37] D. Riehle, A. Perry, Framework Design and Implementation with Java and UML, Tutorials at OOPSLA, 2002.
[38] Object Management Group, OMG Unified Modeling Language Specification (Action Semantics), OMG Document ptc/02-01-09, 2002.
[39] G. Sunyé, A.L. Guennec, J. Jézéquel, Using UML action semantics for model execution and transformation, Information Systems 27 (6) (2002) 445–457.
[40] D.C. Schmidt, Model-driven engineering, IEEE Computer 39 (2) (2006) 25–31.
[41] L. Ehrler, S. Cranefield, Executing agent UML diagrams, in: Proceedings of the Third International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS'04), 2004, pp. 906–913.
[42] J. Huamonte, K. Smith, The use of roles to model agent behaviors for model driven architecture, in: Proceedings of the IEEE SoutheastCon, 2005, pp. 594–598.
[43] S. Mellor, M. Balcer, Executable UML: A Foundation for Model Driven Architecture, Addison-Wesley, 2002, 2002.
[44] L. Xiao, D. Greer, Externalisation and Adaptation of Multi-Agent System Behaviour (Chapter IX), in: K. Siau (Ed.), Advanced Topics in Database Research, vol. 5, IGI Publishing, 2006, pp. 148–169.
[45] L. Xiao, D. Greer, Modeling, auto-generation and adaptation of multi-agent systems, in: 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, pp. 605–616 (in conjunction with the Seventeenth Conference on Advanced Information Systems Engineering (CAiSE'05)).
[46] M. Wooldridge, N.R. Jennings, D. Kinny, 1999. A methodology for agent-oriented analysis and design, in: Proceedings of the Thirrd International Conference on Autonomous Agents (Agents-99), 1999, pp. 69–76.
[47] M. Wooldridge, Agent-based software engineering, IEE Proceedings on Software Engineering 144 (1) (1997) 26–37.
[48] M. Wooldridge, P. Ciancarini, Agent-oriented software engineering: the state of the art, in: P. Ciancarini, M. Wooldridge, (Eds.), Agent-Oriented Software Engineering: First International Workshop (AOSE'2000), LNCS 1957, Springer, 2001.
[49] E. Yu, Agent-oriented modelling: software versus the world, in: Proceedings of the Second International Workshop on Agent-Oriented Software Engineering (AOSE'01), LNCS 2222, Springer, 2002, pp. 206–225.
[50] L. Xiao, D. Greer, Environment support for the configuration of adaptive agents, International Journal of Multiagent and Grid Systems, Special Issue on Engineering Environments for Multiagent Systems, in press.
[51] FIPA, Foundation for Intelligent Physical Agents, <http://www.fipa.org/>.
[52] JADE platform, <http://jade.tilab.com/>.
[53] AUML, <http://www.auml.org/>.
[54] B. Kristensen, Architectural abstractions and language mechanisms, in: Proceedings of the Third Asia-Pacific Software Engineering Conference (APSEC'96), IEEE Computer Society, 1996, p. 288.
[55] R. France, S. Ghosh, T. Trong, Model-driven development using UML 2.0: promises and pitfalls, IEEE Computer 39 (2) (2006) 59–66.
[56] L. Xiao, D. Greer, Agent-oriented requirements modelling, in: Proceedings of the First International Workshop on Requirements Engineering for Business Need and IT Alignment (REBNITA'05), Paris, France, 29–30 August, 2005, pp. 28–37 (in conjunction with the Thirteenth IEEE Requirements Engineering Conference (RE'05)).

[57] V. Kavakli, P. Loucopoulos, Goal-driven business process analysis – application in electricity deregulation, Information Systems 24 (3) (1999) 187–207.
[58] A. Dardenne, A. Lamsweerde, S. Fickas, Goal-directed requirements acquisition, Science of Computer Programming 20 (1–2) (1993) 3–50.
[59] E. Letier, A. Lamsweerde, Agent-based tactics for goal-oriented requirements elaboration, in: Proceedings of the 24th International Conference on Software Engineering (ICSE'02), ACM Press, 2002, pp. 83–93.
[60] E. Yu, J. Mylopoulos, Why goal-oriented requirements engineering, in: Proceedings of the Fourth International Workshop on Requirements Engineering: Foundations of Software Quality, 1998, pp. 15–22.
[61] D. Kinny, M. Georgeff, A. Rao, A methodology and modelling technique for systems of BDI agents, agents breaking away, in: Velde, W., Perram, J. (Eds.), Proceedings of the Seventh European Workshop on Modelling Autonomous Agents in a Multi-Agent World, MAAMAW'96, LNAI 1038, Springer, 1996.
[62] L. Xiao, D. Greer, Towards agent-oriented model-driven architecture, European Journal of Information Systems 16 (4) (2007) 390–406 (Special Issue on Model-Driven Systems Development, Palgrave Macmillan).