The Adaptive Agent Model

Liang Xiao (MSc. BSc.)

Doctor of Philosophy

School of Electronics, Electrical Engineering and Computer Science

Queen's University Belfast

2006

Abstract

Software must change if it is to remain useful. However, with maintenance accounting for up to 90% of the cost of software, change is an expensive necessity. In this thesis it is proposed that the main effort for potential maintenance in the future must as far as possible take place as early as possible in the design phase or earlier if the maintenance effort is to be minimised. Several design paradigms are evaluated for effectiveness in reducing the maintenance cost. Two in particular are considered as most promising, being software agent technology with its inherent adaptivity and annotated business models with their ease of maintenance advantage. Combining these provides an easy means to change the behaviour of software systems without the expensive need to rebuild. The output is an Adaptive Agent Model (AAM), capable of allowing dynamic adaptation of system architecture, functionality, and ontology.

A collection of business knowledge models organised in a hierarchy forms the model architecture of AAM, which drives agent system behaviour. The models originate from the business requirements and are interpreted and executed by agents at runtime. The interpretation and execution of models can be supported by existing object-oriented infrastructure built for the given domain. Agents use continuously maintained models and current object components to fulfil their knowledge and facilitate their function, achieving dynamic effects and up to date behaviour. The existing system components are fully reusable and models easily re-configurable for adaptation purposes. Agents are a higher level of abstraction over objects and an execution engine of business models, bringing adaptivity into reality.

Overall, AAM is built from three-layer business models that interact with a two-layer computing component model. Using this approach an adaptive and easy to maintain system architecture is achievable and deployable in a distributed environment. The configuration of the system is supported by a set of tools. Changes can be made directly by business experts via the tools, and are reflected immediately in the system behaviour. The adaptivity and maintainability of AAM is achieved at its various compositional layers and demonstrated using a railway management system.

Acknowledgements

First and foremost, I would like to thank my supervisors, Dr. Des Greer and Prof. David Bell, for their advice and guidance throughout the course of this research. Special gratitude to Des, who from the beginning strived to arrange my position at the Queen's University of Belfast, helped to arrange funding, and persistently supported and encouraged me over the past three years. His help not only made my research study at Queen's possible but also enjoyable and interesting. Without him the accomplishment of this research was impossible. His extreme kindness and helpfulness has created a pleasant impression of the work I have done under discussion with him.

I must thank my Mum and Dad for their constant support of me during my higher education towards a PhD. They have given me enormous support to allow me to have the chance to study in the UK for the last four years. I can steadfastly rely on their love and encouragement.

I would also thank Dr. Dave Robertson, my former MSc supervisor at the University of Edinburgh. He was my tutor and provided invaluable advice and suggestions during my study of master degree in Edinburgh. He has inspired and directed me to an interesting research area that I could carry on afterwards and explore further now.

Special thanks to I-Ching Yeh, her companionship and timely distractions permitted me to keep a balanced and clear mind. Many local friends must be acknowledged as well: Ruth, Colin, Isobel and her family, among others, who have offered Northern Ireland hospitality and some happy Christmases. They ensured that I enjoyed the past three years and have given me lots of valuable memories and experiences.

Finally I would like to give thanks to the Queen's University of Belfast and the School of Computer Science for supplying a university scholarship, covering travel expenses and so on. Without their provision of financial support I would not have had the opportunity to focus my research study or attend international conferences.

Table of Contents

Table of Figures

Chapter 1: Introduction

1.1 Introduction

Estimates of between 50% and 90% have been made for the proportion of the overall cost of software development spent on maintenance activities [149]. This corresponds to rapidly changing business environments, where business level adaptivity is critical in business efficiency, capitalisation opportunity, cost reduction, and revenue generation [202]. 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, if the expensive burden on business applications is to be relieved. The principle conceptual knowledge that are related with this PhD research topic and its thesis is outlined in the beginning, impinging on four major areas of Software Engineering namely, Software Requirements, Software Design, Software Construction, and Software Maintenance. These are introduced on the basis of a total of ten Knowledge Areas (KA) identified in the 2004 edition of the Guide to the Software Engineering Body of Knowledge (SWEBOK) [151], a baseline on a core body of knowledge on Software Engineering established by the IEEE Computer Society and the ACM.

Software Engineering is defined as the application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software; that is, the application of engineering to software. Software Engineering body of knowledge can be organised into Knowledge Areas. The whole thesis covers combinational areas of Software Engineering.

The *Software Requirements* Knowledge Area is concerned with the elicitation, analysis, specification, and validation of software requirements. Software requirements express the needs and constraints placed on software systems that facilitate people who will use the systems to address real-world problem, for example to automate business processes. Software requirements can be classified as functional requirements, describing functions that the software is to execute (also known as capabilities), and non-functional requirements, being ones that act to constrain the solution (also known as quality requirements). Examples of non-functional requirements include maintainability requirements, a special concern of this research.

A non-functional requirement such as maintainability, or adaptivity, has typically global scope in that their satisfaction cannot be allocated to a discrete component. Therefore, care must be taken on the overall software architecture and the design of collective components to meet such a requirement. Software requirements are typically unambiguously identified with a unique identifier. This makes them subject to software configuration control and provides a means to manage them over the entire software life cycle. It is also useful and important due to the fact that a significant proportion of the requirements will change. Without the understanding of the changing nature of requirements in requirements engineering and the preparation for requirements change, the later software development phases might be confronted with unpredicted difficulty. Requirements could be changed for many reasons. Change is sometimes due to errors in the analysis, but it is frequently an inevitable consequence of change in the environment. Whatever the cause, this often leads to the revision of requirements late in the life cycle, which is very difficult and costly. Rather than manage changes after they occur, which is what people do traditionally, one of the primary beliefs held in this thesis is that the cost will be greatly reduced if adaptivity is embraced in design well before the changes occur. The case study of this thesis presented in Chapter 8 describes functional requirements only. Adaptivity itself is within the scope of non-functional requirements but relates closely to changing functional requirements. The aim in the AAM is to achieve adaptivity through an integrated process, guiding (functional) requirements analysis, design, implementation, and maintenance.

Requirements Elicitation is concerned with where software requirements come from and how the software engineer can collect them. Primary requirements sources include goals, referring to the overall high level objectives of the software; domain knowledge, being knowledge about the application area; and the organisational environment, prescribing the business processes the software is required to support. Goals and processes are useful for requirements reconstruction, and are presented in Section 6.3. *Requirements Analysis* is concerned with the process of analysing requirements. The development of conceptual models of a real-world problem is a key to software requirements analysis. Several kinds of models can be developed, such as data and control flows, object models, event traces, and so on. An important factor that influences the choice of model is the nature of the problem. The modelling process can be better supported and communicated if widely accepted standard

methods and notations are used, such as those of the Unified Modelling Language (UML). Requirements process overlaps with design process at the point of system architecture in the sense that requirements analysis and elaboration demands that the components which will be responsible for satisfying the requirements be identified. Requirements allocation, the assignment of responsibility to components to meet requirements helps to identify components, their responsibilities and relationships, and the overall architecture of components. This method is applicable not only to object-oriented paradigms but also agent-oriented ones as adopted by the proposed approach. Requirements of volatility and stability can be distinguished during the analysis, helping the software engineer establish a design which is more tolerant of change. Following these principles, and with the aim of adaptivity, a new conceptual modelling method is illustrated in the thesis. AAM will adopt the position that requirements are distinguished and assigned to two major kinds of conceptual modelling elements. Volatile requirements are assigned to and managed by dynamic components (agents). Stable requirements are assigned to and managed by statically established components (classes). All these are presented in Chapter 6. *Requirements Specification* refers to the assignment of numerical values or limits to software's design goals. It establishes an agreement on what the software product is to do, and possibly what it is not expected to do. Usually written in natural languages, software requirements specification may be supplemented by formal or semi-formal descriptions. The notation system and its supporting tools for modelling, presented in Chapter 6 and Chapter 7 of the thesis provide a more concise and precise means to document requirements specification. The precise representation of requirements capturing both structural and behavioural semantics enables the later auto-interpretation of requirements, once fully documented in a machine-readable format, by running components.

Software Design is defined as the process of defining the architecture, components, interfaces, and other characteristics of a software system. Precisely, a software design must describe the software architecture in components composition and organisation, and the interfaces between the components. Software requirements are analysed in the design, and the resulting design models enable the software construction. Software design is generally considered a two-step process: architectural design, describing the decomposition and organisation of various components, and detailed design, describing the behaviour of each specific

component. Respectively, higher-level architectural styles design and lower-level design patterns are useful in defining software structure at different levels of abstraction. Software design artefacts can be represented by languages and notations that generally categorised into structural and behavioural descriptions of the software. Structural languages on components and their interconnections include architecture description language (ADL), interface description language (IDL), class/object diagram, and so on. Behavioural languages on dynamic component behaviour include activity diagrams, sequence diagrams, decision tables/diagrams, data flow diagrams, and so on. Software design methods include the classical function-oriented design, where functions are identified, elaborated on and refined in a top-down manner; object-oriented design; and component-based design. Software design principles include abstraction, coupling, cohesion, decomposition, modularisation, encapsulation, information hiding, and separation of interface and implementation. In this thesis, a new design method is envisioned, accompanied by a structural design language and a behavioural design language, both of which are adaptive. The architectural and detailed design utilise two integrated models at different abstraction levels. The result is that, all of the design principles are strived towards in the method, being a means for designing products with a high quality of adaptivity. The details are presented in Chapter 6.

Software Construction refers to the detailed creation of working, meaningful software through a combination of coding, verification, testing, and debugging. Software systems change over time, and the anticipation of change drives many aspects of software construction. Some development models (linear ones), such as the waterfall model, tend to distinctly separate construction, mainly emphasised on coding, from other activities, namely requirements and design, which are preceding in sequence and exceeding in importance over construction. Other models (iterative ones), such as Extreme Programming, tend to treat construction as an activity that occurs concurrently with others. These activities are mixed and often the combination of them is treated as construction. Construction languages include any language that humans can use for specification and for computer comprehension and execution. Examples range in order of increasing complexity include configuration languages (such as ones used in text-based configuration files), toolkit languages (such as scripts), and programming languages (such as Java). Three general kinds of notation used for programming languages are linguistic, formal, and visual. Coding

techniques in using languages include use of classes, variables, control structures, error conditions handling, and so on. Reuse is very important in software construction. During construction, separately constructed classes, components, and subsystems need to be integrated, also possibly with other external systems. *Software Testing* is necessary after and/or concurrent with construction but is omitted here in favour of concentrating on the main topic. In the thesis, two major construction components (agents and objects) are distinguished to fully exploit reuse and accommodate unanticipated changes. Model driven architecture is promoted (with visual and semantic model diagram construction) to avoid reconstruction or recoding whenever changes occur. Instead, reconfiguration of models, sometime textually and more often diagrammatically, is the only effort required to address changes. These are presented in Chapter 7.

Software Maintenance is defined as the totality of activities required to provide cost-effective support to software. Once the software products delivered from the development is in operation, defects are uncovered, operating environments change, and new user requirements surface. Software must change or evolve. Historically software maintenance has received much less attention than other activities in software development. This is changing as software products are expected to keep operating as long as possible.

Maintenance activities, although similar to those of software development, also include unique ones such as: transition (from developer to maintainer), modification request acceptance/rejection, modification request and problem report help desk, impact analysis, software support, service level agreement, and creation of specialised maintenance contracts. An important activity for software maintenance is planning, including business planning, maintenance planning, release/version planning, and individual software change request planning.

Maintenance activities are performed during pre-delivery, such as planning for maintainability, as well as post-delivery, such as software modification. Maintainability, a software quality characteristic, is defined as the ease with which software can be maintained, enhanced, adapted, or corrected to satisfy specified requirements. It is difficult to achieve software maintainability because developers often pay much more attention on development at the moment than the maintenance to be handed over to maintainers in the future. This thesis demonstrates the importance and influential characteristic of pre-delivery maintenance,

planning/designing for maintainability in particular. The result is that post-delivery activities are greatly eased in nature and reduced in overall capacity. Post-delivery maintenance efforts will otherwise involve a process similar to the normal development process, including analysis, design, implementation, test, and delivery for each request of modification (IEEE 1219 [108]). The release of a new version of the software product for each requested change could actually be avoided by simply reconfiguring the existing product, if maintainability has been integrated in design as is proposed in the thesis.

Maintenance consumes a major share of software life cycle financial resources. Studies and surveys indicated that over 80% of the software maintenance effort is used for non-corrective actions. According to Lehman's findings [104], maintenance is evolutionary developments and maintenance decisions are aided by understanding what happens to systems over time. Adaptive maintenance, keeping the software product usable in a changed or changing environment, is therefore a focus of the thesis.

Research indicates that some 40% to 60% of the maintenance effort is devoted to understanding the software to be modified. Software comprehension is of great interest to maintainers. Maintainers will meet challenge when developers of the software under maintenance are not reachable, from whom maintainers can learn knowledge and gain understanding of the software. Comprehension is more difficult in text-oriented representation, for example, in source code. It is shown in the thesis how diagram-oriented maintenance is possible, which greatly boosts effective and efficient software comprehension and ultimately maintenance, even without the presence of the original developers.

Large corporations are outsourcing software maintenance, the required activities becoming a major industry. Some will outsource only if they can find ways of maintaining strategic control, which is often difficult. This may be more possible if businesses can focus on business strategy control on their own and not been distracted by the technical details, software maintenance being a business task rather than a technical task. This is an aim of the AAM approach and details of the maintenance methods are presented in Chapter 9.

In short, the methodology presented in this thesis impinges on four major areas in Software Engineering, the chief aim being to decrease software maintenance cost. In the thesis methods and tools are provided so that: *Software Requirements* are easily

changed and immediately available in the formatted specification; *Software Design* such that it is able to accommodate changes without major rework; *Software Construction* needs no rebuilding following design changes; and Software Maintenance is carried out on a regular basis textually and diagrammatically, reflecting changes on the running software products instantly. Although the approach is illustrated by a specific case study, the integrated process, the guiding knowledge spanning over various phases of the process, and the supporting tools accompanied the approach serve a common framework that is transferable across many application domains.

1.2 The 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 [103]. In the real world, often originating from changing requirements or running environments (83% of overall changes [63]), progressive change and adaptation of the system is inevitable if customer satisfaction is to be maintained. When individual entities or entity classes are adapted in their respective contexts, they evolve and attempt to maintain their fitness to the forever changing environments [112]. However, evidences given in the Law of Continuing Change and Increasing Complexity among other Laws of Software Evolution [104], show that after many adaptations, software systems tend to drift away from their original architecture and design, becoming more and more difficult to adapt and progressively less and less useful. To compound matters, maintainability worsens [63] and the system loses value. Consequently, maintaining existing software turns out to be more challenging than building new software.

Typically operational costs are 20% and non-discretionary maintenance 40% of development costs for the life of the system [148]. Other recent studies [146] suggest that 30% of software projects are cancelled before completion while 50-100% over budget and 6-12 months late. Similar alternative surveys [147] show that only around 40 percent of respondents thought their projects were late or over budget. When the responses with respect to the project duration of more than two years is examined, however, it is believed that they keep to schedule and budget only about 20 percent of the time. Responses for projects lasting beyond two years were, on the whole,

more negative. In any case, this is clear evidence of a major problem facing software development organisations.

One important source of trouble in software projects is that their supporting business systems are constantly changing due to the changing business world. Business requirements and knowledge are doomed to be incomplete and imperfect even before starting to build the system. Therefore, the software engineers and developers have to continuously keep maintaining IT systems to align them with the changing business needs. For example, the inherently volatile system requirements such as organisational policies and procedures as well as the business processes applied to are among key factors to be considered, when one predicts the future system maintenance efforts [63].

Therefore, the maintenance burden would be greatly reduced if the system is designed in such a way that it can accommodate new requirements, such as policies, procedures, and business processes. Finding an adaptive software paradigm where software is capable of adapting itself to changing these business requirements is the eventual goal. The primary outcome that Software Engineering activities should seek at the end of software production, therefore, must shift from being an accurate software system that implements all current requirements to also being an adaptive software system that requires minimum maintenance in response to new business needs over time. This shift corresponds to a three-stage software development evolution, each of which has its own requirements to adaptability [202]: build the right thing (adaptable software allows the product to be easily changed if not an exact match to what the sponsors want); build the thing right (adaptable software facilitates the fixing of defects and maintains a low probability of introducing new defects); and support the next thing (adaptable software supports a high level reusability and user customisation to meet future needs).

In designing an adaptive software system, employed software components must be as dynamic as possible, involving least code change. If a software system is expected to meet new requirements dynamically, a means must be sought to externally capture the new requirements and to make them accessible to the running components. Such knowledge represents business needs. If it is to truly reflect the current requirements in a timely fashion, the knowledge on, for example business processes, procedures, policies and alike must be easily editable by business experts, without intervention of IT experts. This leads to a responsibility shift in that business

people will get more involved in software maintenance and IT people must design adaptive systems that facilitate the maintenance by the business people. Such design must enable components of the running software to interpret dynamically and immediately their required structure, and how they should behave, at runtime, by using the currently captured knowledge. Ultimately, such a software system would need no re-delivery at all and therefore no downtime or lost business opportunity, due to unavailability caused by waiting for the next release. The system is controlled by customers, bringing the newest requirements to the system persistently at low cost. IT people are freed from routinely maintaining code and so human resources saved.

1.3 Software adaptivity (OO vs. AO methodologies)

It is argued in [144] that, "over the life of an information system, it is maintenance, not development, that is the main work. And within maintenance, it is resynchronisation/adaptation, not error correction that is the dominant activity". It is also stated that adaptability to functional change is a new and fundamental objective essential to good system design in addition to the traditional objective of functional accuracy. This claim is based on the accepted fact that "knowledge of a system's requirements is necessarily imperfect because a significant part of those requirements lies in the future and is unknowable at the time the system is designed – or built, for that matter. People must therefore rely on techniques for improving flexibility – the system's ability to change gracefully – rather than for improving their knowledge of requirements. The maintenance burden will be eased only when systems themselves can be modified easily to meet unanticipated requirements."

Unfortunately, software maintenance and evolution remain difficult in many systems because they are not built to be adaptive or flexible and so are resistant to frequent modification. Even if some of them are in any sense adaptive, the code that implements adaptation is often tangled and hard to manage or comprehend [57]. New requirements are difficult to introduce to existing systems due to the complicating extra code around the core system structure.

Included within this investigation are various efforts that have been put forward to alleviate the software maintenance burden by making software systems configurable and/or adaptive. Recent examples are the use of the Strategy Design Pattern [15], Coordination Contracts [16] [58], the Adaptive Object Model (AOM) [17] [18], and

Commercial Off-The-Shelf (COTS) systems [107]. All of these approaches and others attempt to add adaptivity to systems but offer limited adaptation and end up creating complexity, unjustifiable presumptions, or introducing side effects that inevitably impede their use. A review of these example approaches is found in [5]. In fact the object-oriented paradigm facilitates design by use of such principles as modularity and information hiding, but this implies ease of re-design rather than adaptation during operation.

In fact, an object-oriented system is vulnerable to a wide range of potential variability. As an example take the simple case of people working in organisations and consider it with respect to the structure alone. Careful analysis would have to be carried out on variability before an adaptive system could be built. Otherwise the system might have to be painfully rebuilt. For example in a given scenario, individuals could take multiple roles. When an operator is promoted to be a manager, they may at the same time work in their original workplace and manage other workers. Nothing prohibits them from taking two roles with the same identity. Moreover, he could also be a customer of a service provided by the same organisation. Failure to consider all these matters may result in a poor system design that requires rework later. Another typical scenario relates to the fact that relationships change. New (types of) relationships come and old (types of) relationships go. Not all of them are predictable. Constant changes on ownership, department division re-organisation, and alike must be coped with the information system keep running and properly supporting the ongoing (changing) business. Another scenario relates to the fact that cardinalities change. It could be an original requirement that a manager manages a certain number of workers. However, this number is very likely to change when a new organisation is merged into the current one, causing the re-assignment of managers to operators. Furthermore, supervisors could be introduced in a stage that supervise workers and report to managers. Managers never need to directly manage workers. Ignoring any of these potential changes could trigger major re-development. It will be demonstrated in later chapters how the solution presented in this thesis responds to many kinds of variability rising at runtime, without the precondition that they have been predicted before the system was built.

Recognising the inherent difficulty of using object-oriented technology to handle variability in the envisioned scenarios, another means must be sought. Unlike

standard objects, agents are active. Instead of using static methods which are to be invoked and have the same effects all the time, agents are granted the flexibility to choose how to react. Intelligent/autonomous agents have been proved useful for bringing dynamics, flexibility and adaptivity to travel planning [19], coordinated product development and manufacture [20], and manufacturing systems control [21]. Also, it has been demonstrated that maintenance tasks can be carried out remotely by some mobile agents [109] in agent systems, running separately from the target software system under maintenance, for example, ERP systems [110] and hospital information systems [111]. The combination of externalised requirements which is the maintenance target and an agent system which is the maintenance actor is a promising solution in using agents for adaptivity.

1.4 Model Driven Architecture (MDA) & Model Driven Agent Architecture (MDAA)

Traditionally, human knowledge is transferred into software systems in the form of requirements documents, design models and eventually implemented code, the performance of which should precisely reflect the desired behaviour in the required system. The initially captured knowledge is typically documented in UML models. UML Use Cases can help with requirements capture by providing a structured way to elicit and identify the actors and the interactions they have with the system. UML Class Diagrams describe the structure of a system and Sequence Diagrams describe a sequence of required actions, in terms of messages being passed. These requirements and design models provide a high level view of the system and help to reduce variability in the specification. However, the original requirements models, being largely textual descriptions of system functions, are separated from the developed design models, which lack the capability to describe behavioural semantics exactly [9]. This is a major limitation of traditional software system development. The UML artefacts rapidly lose their value if, as is often the case in practice, maintenance changes are done at the code level only.

This situation has been recognised in Extreme Programming (XP) [135] which focuses on coding and testing. XP proponents argue that the building of models is an overhead in the eyes of developers. Counter to this, others argue that without these high-level models, developers, especially new team members will get lost in code,

being unable to understand what the software is doing and, consequently, maintenance becomes more difficult. Disregarding modelling contradicts with the expected intention that business is the master, rather than the servant, of technology and limits business agility and technology freedom by emphasising the *what* over the *how* [28]. A further argument in favour of models is found in the need to get the requirements as correct as possible and as early as possible, thus avoiding more expensive changes at the design and implementation stages.

The value of system models would be increased, if they could be converted to executable software. The idea of emphasising system knowledge, and making them reusable models that can be converted to executable software, is promoted in the up and coming Model Driven Architecture (MDA) [137] [9] [39] [201]. 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). However, one difficulty in this process is that the process of PIM→PSM→code starts from the design products rather than requirements models. Consequently it requires highly creative work [39] to build a PIM from narrative requirements documents. This results in a high cost of 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 [136], a combination of UML and OCL [137] is used in MDA. However, OCL constraints are static and are external 'add-ons' to UML [80] and used in the design stages rather than the requirements stages. 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 (usually based on the commonly used UML) to their various target platforms (e.g. J2EE or .Net) makes such auto-transformation especially hard to achieve [200].

Nevertheless, aiming at the objectives of MDA, a modelling paradigm that starts from integrated requirements and design models, fully re-useable throughout the software life cycle must lead to complementary improvement. In addition, such models, if coupled with the adaptivity that can be achieved using agent technology,

shall produce even greater productivity and flexibility. With such a perspective, a Model Driven Agent Architecture (MDAA) is put forward. The proposed agent-oriented methodology allows mutable specifications and agent behaviour to be driven by modelled specification knowledge. Since requirements are unpredictable and their changes lead to maintenance effort, a set of re-configurable models that directly link requirements to executable agents is needed. This requires that at the early stages of software development an adaptive modelling structure is used, that gives the system a clear and comprehensible division into units [55], and that is able to easily accommodate and facilitate changes. The models outline the agent system behaviour without regard to a particular implementation environment. Thus it is agent platform neutral. A common Agent Model is instead specified for any agent to be built from any platform and becoming a participant of the integral AAM agent system. Since the approach follows the principles of MDA, models are reusable and executable by agents. Therefore, when the requirements are changed by business people, newly arising specifications are re-interpreted and the changes are passed to the software system without re-delivery by developers. Consequently, the business knowledge modelling phase becomes the essential and primary step in software development, and effort spent on it will never be wasted.

One benefit of the MDAA is software adaptivity. Software evolution and maintenance is expensive and accounts for the majority of software lifecycle costs [113]. To make systems more adaptable, one can make them easily changed by engineers. Better still, would be the ability to make systems adaptive, where systems change their behaviour according to their context [114]. In AAM, the business models that drive system behaviour capture the context in which agents behave.

1.5 Overview of AAM, its structural components, its goals and novelty

As mentioned in the previous section, adopting the objectives of MDA and using the adaptivity attributes of agents, the enhanced architecture of MDAA is put forward here, where knowledge on agent behaviour is structured as business models. The amendment of these is carried out directly by business people, reflecting desired business requirements. The execution is performed by agents, so reflecting deployed requirements. Such an approach offers several advantages. Firstly, traditional OO

methodology is the basis of the framework so that no radical development model change is involved. Secondly, knowledge of the system can be more easily maintained, as business models are designed in diagram and text formats and can be controlled at run-time. Thirdly and lastly, such systems, even based on OO components, can enjoy the added benefits from the agent side.

Precisely, it is proposed that two hierarchical structures are combined: i) the hierarchy of business knowledge and ii) the hierarchy of computing components that use and apply the knowledge. Collectively these hierarchies are termed the Adaptive Agent Model (AAM), as illustrated in Figure 1.1. The knowledge captured in the business models has two building blocks: a) a Conceptual Model (CM), used for vocabulary definition and referred to by agents and b) a Fact Model (FM), conforming to the CM, constructed at runtime according to a given agent's current knowledge. Two rule models are in turn used to model agent behaviour. Policy Rules (PR) are global rules that all agents should obey and describe policies that must be enforced. Reaction Rules (RR) are local rules that specific agents should use and describe reactions that must be performed when triggered by external events. Business Process Rules (BPR) are collections of PRs and RRs that realise business processes aimed at corresponding goals. In addition to the business models, the computing components of agents and objects form another hierarchy, where objects support agents' behaviour. The interaction of the two hierarchies achieves the required goals of the business. Such structures comply with a long recognised idea that, for complex systems to work, including living organisms, they must be hierarchically structured with minimal dependencies between intercommunicating components [102].

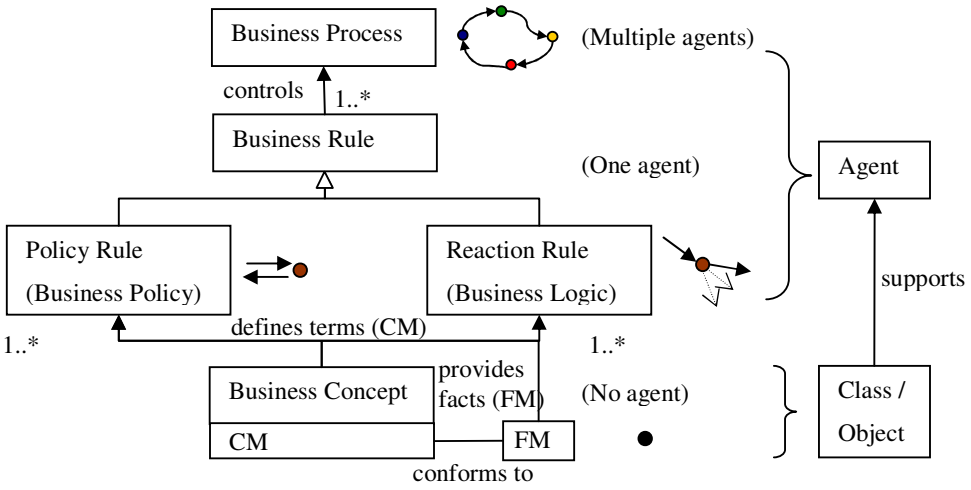

Figure 1.1: AAM components structure: two interactive hierarchies

In order to achieve the top level goals of Model Driven Architecture and of business people directly controlling maintenance, such two hierarchies are defined, which in turn have aims as follows. i) In the hierarchy on the left hand side of Figure 1.1, the overall model structure and each individual layer of the model is adaptive to changing business requirements at various levels. ii) In the hierarchy on the right hand side of Figure 1.1, the use of appropriate objects by agents is decided dynamically to achieve dynamic running component execution effects. iii) Agent behaviours are driven by dynamically maintainable models, and so the application of one hierarchy by the other is adaptive, maximising their existing combinational adaptivity.

Referring to the four knowledge areas discussed at the very beginning of the thesis (Software Requirements, Software Design, Software Construction and Software Maintenance), the AAM business model and component model architectures presented in Figure 1.1, respectively, aim to reduce the effort specifically for:

(i) Requirements evolution,

(ii) Design evolution,

(iii) Constructed software re-configurability and

(iv) Maintenance through enhanced modifiability, as well as straightforward business model evolution.

After a full discussion of the AAM approach, existing approaches that are available in these areas will be revisited at the end of the thesis in Chapter 10, where AAM will be assessed for its satisfaction against these four aims.

Figure 1.2 shows a Goal-Question-Metric (GQM) [211] graph. Using the graph, AAM is described it terms of its main goal of software adaptivity. This top level goal can be decomposed as two sub-goals with respect to object of adaptation and subject of adaptation. These two sub-goals identify *what should be made adaptive* and *who should carry out adaptation and how*. Questions are then identified which can be used to evaluate whether any new approach meets these goals and metrics are identified that will provide the necessary evidence to answer the questions. Example questions are related to whether or not the modification of various business elements is made easier by using the new approach, and if business users can make changes easier with immediate effect. Ideally the metrics to be collected are based on controlled experiments or detailed observation. However, given the novelty of the approach being proposed and the difficulty in setting up realistic software engineering experiments, it is proposed here that using a case study as a demonstration system will suffice in answering the questions and to provide evidence of an improvement. These topics are the focus of the thesis and are discussed throughout with Chapter 9 providing a summary of the result.

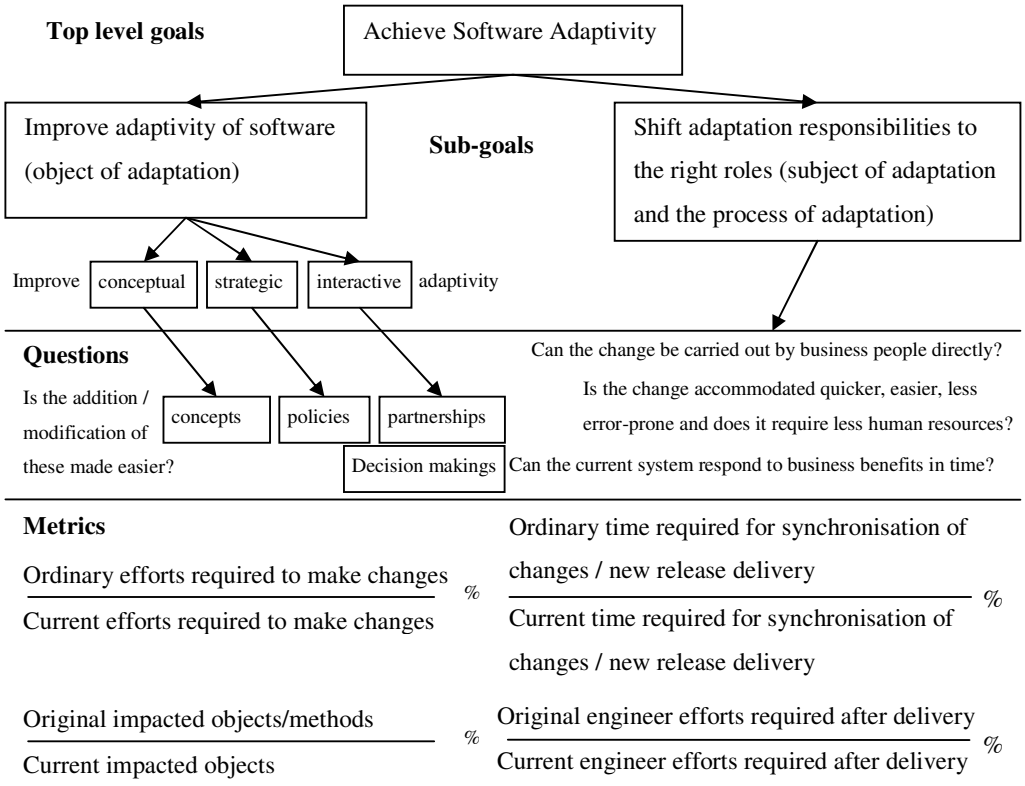

Figure 1.2: AAM method roadmap presented in a GQM graph, starting from its top level goal

The novel aspects of the work at a generic level (technical and detailed novel aspects are found in detail in the Chapters 6 - 10) are as the follows.

- AAM puts Model-Driven Architecture paradigm into practice in agent-oriented software development.
- The maintenance of business knowledge models diagrammatically and textually becomes the maintenance of the overall software system. Software adaptivity is thus achieved, with adaptation reflected dynamically and immediately at runtime.
- Business people are enabled to directly control their own business as system maintenance becomes a daily routine of business practice. This shift of responsibility represents a significant change of their role in software development.
- AAM supports a full development process for agent-oriented software systems towards adaptivity.
- A hierarchical structure of business process, business rule, and business concept is organised as business representation across business domains, elicited from business requirements and documented in UML and XML for easy business maintenance and dynamic agent execution.

1.6 Thesis structure

The remainder of the thesis is structured as follows. Chapter 2 introduces recent major approaches towards software adaptivity. Chapter 3 provides a literature review, including agent-oriented software engineering and knowledge modelling, both of which contribute significantly to the solution approach presented in the thesis and serve as a knowledge background. Chapter 4 outlines a problem statement and Chapter 5 an overview and aims of the approach. Chapter 6 discusses the main approach of Adaptive Agent Model, its requirements modelling and design modelling in particular. Chapter 7 discusses the AAM implementation, deployment and tool support. An example implementation on Java Agent DEvelopment Framework (JADE) [134] is given to demonstrate the advantages of using the approach over a traditional agent platform, after the provision of the generic implementation model. A new software development process model is provided in the end of the chapter. Chapter 8 uses a comprehensive railway management system

to demonstrate the overall development process of AAM. Chapter 9 summarises the potential maintenance benefits and adaptivity achieved by AAM. Chapter 10 provides conclusions and ideas of future work.

Chapter 2: Literature Review: Software Adaptivity

2.1 Introduction

Various approaches attempt to bring flexible object behaviour but in general they add extra difficulties such as increased complexity. This chapter will introduce some recent major efforts and investigate their usefulness and deficiencies.

2.2 Strategy Design Pattern

The concept of design patterns has been around for some time now and addresses reoccurring problems by reusing existing proven design solutions, the best known patterns having been documented in [15]. Some patterns have been discovered that describe ways of adapting behaviour, without the need to rewrite code. One such pattern is the *Strategy Pattern*. This pattern is used where a client is served by one of many alternative mechanisms, depending on policy criteria. The pattern allows different algorithm implementations to be interchangeable in a way transparent to the client. Briefly, the required algorithms are predicted and written and then during the operation of the programme these can be selected from as appropriate. As shown in Figure 2.1, one generalised class is created and accompanied by a family of concrete classes which extend it. Depending on which concrete type is required at runtime, the correct object is instantiated and used from that point forward. This is preferable to a solution where a complex if-then-else structure combining all algorithm implementations together is constructed, representing a decision tree for the selection alternatives. In contrast to this, the Strategy Pattern organises algorithms into separate implementations and the dynamic selection of the required one is performed using independent code. Additional algorithms can be accommodated into the pattern in the future. The limitations include the fact that neither adaptivity on structure nor adaptivity on ontology are achievable; that behavioural adaptivity is limited since algorithms cannot reuse the commonality of each other; and that algorithms which cannot be predicted in advance cannot be added at runtime without recoding and rebuilding.

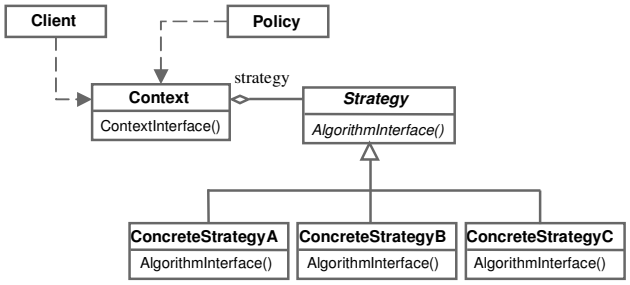

Figure 2.1: Strategy Design Pattern [15]

2.3 Other Patterns

Dynamic selection of algorithms is not the only means of achieving adaptivity. Also important are the capabilities of creation and use of object types with particular behaviour, addition of structuring attributes to those object types, and flexible composition. Each of these provides different facets of adaptivity. Major design patterns that attempt to address these facets have been investigated. The *Abstract Factory Pattern* encapsulates families of related objects and shields the client from the creation process. Different concrete factories are used to create particular product objects with different implementations at runtime, exchanging product families being supported by configuration of product families. The Abstract Factory Pattern is useful for the dynamic selection and use of structures and the Strategy Pattern is useful for the dynamic selection and use of behaviours, meeting requirements coming up at runtime flexibly from two different aspects.

The *Adapter Pattern,* by using an adapter class, allows the reuse of an existing class, the interface of which does not match the one the client expects. The *Composite Pattern* defines a class hierarchy made up of primitive objects and composite objects, the latter being composed recursively by primitive and composite objects into arbitrarily complex structures. New components can be accommodated into the pattern and clients treat composite objects and individual ones uniformly. The *Decorator Pattern* adds responsibilities to individual objects dynamically.

Generally speaking, design patterns help developers to use common solutions to common problems in given contexts. Fundamentally, design patterns can only encapsulate certain changeable constructs such as system structure or behaviour into configurable metadata (for example, a policy for the Strategy Pattern or an adapter class for the Adapter Pattern) to provide adaptive solutions, so reducing the required maintenance effort. Individual patterns each address one aspect or another of the

adaptivity issue by using a specific pattern of class clusters suitable in a certain context. However, when many aspects of adaptivity are required at once, which is the normal situation, the use of all required design patterns at the same time would be almost impossible and doomed to be problematic and impractical. Even combinational use of a few of them demands highly creative and subjective design skills and leads to excessive extra complexity. Three design patterns, namely Observer, Strategy, and Facade have been used together to try to achieve adaptivity. The Observer Pattern allows a set of subscribers to be notified of any state change. The Strategy Pattern describes the implementation of an object that can dynamically change its behaviour. The Facade Pattern provides an interface to encapsulate a set of objects. When a state change occurs, it is notified (Observer Pattern), and a dynamic change of behaviour is imposed under control (Strategy Pattern), the functionality of both being encapsulated and providing a proper and unified interface to use (Facade Pattern) [57]. Though a system designed in this way can be adaptive, the resulting design structure is very complex due to an increased number of classes and a large resultant component communication overhead.

Also in [115], problems of traceability, reusability, and implementation overhead have been identified for the implementation of design patterns using traditional object-oriented languages. New class relation [116], new language support [115], or even new patterns [117] that complement or optimise the use of design patterns have subsequently been suggested in order to mitigate the exposed problems.

More importantly, since business changes arise at the requirements level, while design patterns manage variability at the design level, a wide conceptual gap prevents such solutions from easily adapting to changing environments. The mapping from the business abstraction to the specific patterns is not usually straightforward. It is therefore difficult to identify required changes of the design patterns corresponding to changes in the business. Even worse, design patterns might not be evolvable in response to the required changes at the business level and the potential intrusive changes could eventually cause them to be changed so much that they are no longer adaptable [166].

Actually, a related study on architectural abstraction and language mechanism [192] has examined and compared the Mediator Pattern and the Decorator Pattern with the concepts of “Activity” and “Role”, respectively. The Mediator Pattern allows an object to encapsulate its interactions with a group of other objects so that it

does not need to explicitly refer to them for cooperation. The concept of "Activity" has been characterised as a sequence of interplays between actions of a participant with actions of others. The Mediator Pattern and the "Activity" concept resemble each other in the sense that both attempt to encapsulate collective behaviour. Similarly, both the Decorator Pattern and the Role concept can be used to assign additional responsibilities dynamically to objects.

Language abstractions and patterns are both intended to solve recurrent problems, having similar modelling capabilities. The study argues that the independent development of language abstractions and patterns have mutual influence and inspiration to each other's evolution. An additional language mechanism is usually more expensive but once it is in place the corresponding design patterns become unnecessary. However, an abstraction mechanism is more general and more reusable than design patterns. The former can be applied easily and flexibly across applications but the latter is often constrained in a given context. Moreover, the design patterns must be implemented and maintained manually and individually. Once a language mechanism is developed, its application is automatic with no need of user involvement or management. A conclusion can therefore be drawn that when a new abstraction mechanism becomes necessary, it shall be developed once and for all, rather than having to apply hard-to-reuse design patterns in many places. The concept of Agent as a new language abstraction is proposed in the thesis and believed to be a more appropriate mechanism to achieve adaptivity than the described design patterns that attempt to achieve the same goal.

2.4 The Adaptive Object Model (AOM)

The design patterns discussed above are not designed exclusively to cope with the adaptivity issue but provide common design solutions to common issues. The building of several patterns fully dedicated to adaptivity may address adaptivity better. The *Adaptive Object Model* (AOM) is such an attempt. The AOM is described in [17] as a framework that models business units with metadata, which will be interpreted at runtime. This is achieved using the *TypeObject*, *Property* and *Strategy* patterns [18]. The TypeObject Pattern uses an EntityType-Entity structure to replace the Class-Subclass structure, making unknown subclasses simple instances of a generic class. This avoids the problem where there is an unpredicted number of new classes by allowing their creation at runtime from generic types in the same way as

objects are instantiated from classes. New business entities therefore can be dynamically defined for the system. The Property Pattern in the AOM uses an Entity-Property structure to separate an object from its property which encapsulates a collection of attributes. This allows objects of different types but the same class to have different sets of attributes. The AOM also uses the established Strategy Pattern to allow dynamic behaviour to be configured for classes as described in Section 2.2. Further, the Composite Pattern, as described in Section 2.3, is applied here to evolve strategies, including composed primitive and composite rules. Dynamic association of rules with entity types at runtime can provide the system with the required behaviour. Figure 2.2 shows the AOM architecture with the TypeObject Pattern applied twice with the Property Pattern and then Strategies Pattern added. The power of the AOM lies in its separation of the metadata from the system using a set of patterns that can later be reconfigured. Nonetheless, the AOM has several weaknesses. Firstly, since classes are created dynamically in the AOM, if they are to be persistent, their definition must be stored, say, in a database. The hard-coded original classes, therefore, do not represent business abstractions because most of the information about the business is in the database. Hence, developers may find the architectures required in this approach hard to understand and maintain. Secondly, the AOM in common with using the Strategy Pattern can only customise behaviour within the limits of what has been predicted. Thirdly, there is no easily accessible central model so that analysts and clients alike may find it difficult to track the current state of the system.

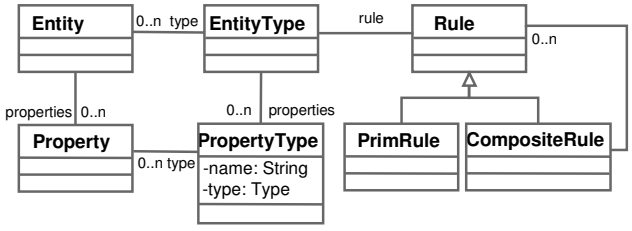

Figure 2.2: Architectural patterns of Adaptive Object Model

2.5 AOM + Adaptability Aspects Pattern

Code for adaptive purposes in AOM systems is typically mixed and scattered throughout the system. This makes AOM systems difficult to maintain since new unpredicted requirements still require code rather than metadata to be changed. When changes inevitably cannot be handled via the metadata, the effort in editing

scattered code is much more difficult. Therefore AOM systems are not 'adaptable', despite being 'adaptive' [119]. To solve the problem, the *Adaptability Aspects Pattern* [118] has been used. The idea is based on the separation of concerns and using an "aspect weaver" for the composition of them. The aim of aspect-oriented software development (AOSD) is the separation of concerns and can be described as the process of breaking a programme into distinct features that overlap in functionality as little as possible. A concern is any item of interest or focus in a programme. An aspect is a part of a programme that cross-cuts its core concerns, therefore violating the so-called separation of concerns. An "aspect weaver" is an engine that glues together and executes aspects and their components. Adaptability Aspects Pattern is an attempt to modularise and better manage those aspects concerned with adaptability. Although the pattern itself provides adaptability and so improves the AOM's overall adaptability, it further increases the complexity by using extra elements, so exacerbating the maintenance problem. In addition, it is impossible to dynamically load the code for new adaptation mechanisms without stopping the running of such systems.

2.6 Coordination Contract

The Coordination Contract [16] [58] [166] approach recognises the importance of rapid software evolution in line with business evolution (possibly in real time) and the weakness of the object-oriented paradigm in its use of tightly coupled components to serve business. It attempts to break such a tradition of inconvenience by a service-oriented paradigm that enables late binding of component services to meet new requirements emerging from changing business domains. Its assumption is that, in a component-based system, each component provides stable functionality. The Coordination Contract approach provides a means to achieve dynamic reconfiguration by externalising the coordination between components as contracts (text-based descriptions). Contracts express business rules and constrain the joint behaviour of system components to which they apply. System evolution consists of adding and removing contracts without modification of the computation implemented in the participating components. Code level change of explicit component function invocation is replaced by change to specialised contracts, externalising component interactions. Thus, flexible combination of components brings more variable behaviour than with the Strategy Pattern.

The Coordination Contract approach can enhance the adaptation of object-oriented systems in the regular and traditional way. Its advantages have been illustrated in an example of bank accounts with customers who can make debit operations on accounts [166]. In the example, both customer and account have been defined as classes in the OO system under development. The class “Customer” has an attribute “account”, instantiated from class “Account”, which itself has a “balance” attribute. The “debit” operation that a customer performs on an account can be expressed as: account.debit (amount). The operation has a precondition that the account balance must be greater than the amount to debit, a requirement/rule of the business domain. When the bank introduces a new “VIP” package for certain customers, the precondition of the “debit” operation that those customers can perform becomes: the account “balance” plus a credit allowance limit” must be greater than the amount to debit. To respond to this change, several regular OO solutions are demonstrated to have various problems. i) Add a new “VIPdebit” operation to class “Account” with the new precondition. This requires a direct coding change to class “Account” and extra input from customers who must choose from debit operations. ii) Define class “VIPAccount” as a subclass of “Account” with a new attribute of “limit” and the new precondition. This creates a class with no counterpart in the real world since it is customers not the accounts that are VIPs. In addition, it requires other classes in the system to be aware of the new specialised accounts. iii) Use a specialised association class “ownership” to capture the interaction of the two classes, including the precondition of the operation. Although the two classes in association need no change and become stable by using the association class as a mediator, this solution is of limited scalability and is problematic if new mediators are to be added to accommodate new business requirements/rules. Similarly, other classes in the system must be aware of the new mediators when they become available and choose the right ones to establish interactions. In case these requirements/rules are interrelated, relationships among mediators must be modelled in addition to those between the original classes and mediators, leading to models that are hard to maintain. The Coordination Contract approach decouples the explicit debit invocation from the execution of service itself. Coordination rules in a form of {when <event> - with <guard> - before <action> - do <action> - after <action>} and contracts in a form of {participants, constants, attributes, operations, invariant, coordination rules} are used to intercept calls and

select the required coordination. In the same example, the two contracts of "Ownership" and "VIPOwnership" are declared, the latter extending the former. The actual code for debit operations including preconditions is extracted from original classes and expressed as condition and action statements in two versions of contracts for different purposes. Using Coordination Contracts, requirements/rules changes are restricted and relevant only to contracts. This allows independent evolution of subject classes and so advances a solution to the adaptation problem.

The example with its challenge upon object-oriented adaptation shows why object-orientation is not evolutionary [166]. The Coordination Contract approach is a successful attempt to alleviate this limitation by providing component dependency adaptivity. However, it is inadequate in other aspects due to its reliance on traditional object-orientation. A major insufficiency is that its components will still have to be re-developed even if their functionalities only need to be slightly changed. Completely new paradigms are necessary to address both adaptivity on inter-component dependency (as being answered by the Coordination Contract) and adaptivity on intra-component functionality.

2.7 Adaptation classes

Adaptation classes [57] are designed for multimedia applications to adapt encoding algorithms to available network bandwidth in respect to a guarantee of performance. The separation of concerns of adaptation classes from basic application features improves maintenance and extensibility. However, adaptation actions must be pre-defined and where adaptation is needed must be exactly known, thus hindering the potential adaptivity of such systems. Moreover, a dedicated adaptation compiler is required, which adds to the complexity of the system. Since objects are passive and traditionally have fixed methods, to make them adaptive either it is inconvenient or impossible [5].

2.8 Conclusions

One fundamental property of all these approaches is that they are OO-based. However, objects are static with fixed methods and so making them adaptive is impractical. For example, if using the strategy design pattern, future behaviour must be fully predictable. In the case of using the Coordination Contract approach, only inter-component collaboration can be adapted, so that it is insufficient. The AOM

approach relies on meta-models and leads to an architecture which is hard to understand and maintain. Table 2.1 summaries approaches discussed in this chapter and assesses them against AAM's aims described in Chapter 1.

Table 2.1: An assessment of existing adaptive approaches against AAM's aims

Aim of Adaptivity / *Approach*	Model (Meta-data) driven architecture	Business people direct control with immediate change effect	Business Concepts	Business Rules	Business Processes
Strategy Design Pattern	Yes, but limited in predicted behaviour. Its use of "policy" drives the dynamic selection of strategies.	No capability to add new behaviour by business people directly. New strategies must be coded by developers in advance to take effect.	No	Yes, but only if all future rules can be predicted and coded into strategies (functional rules) and the policy (policy rules) which controls the selection of strategies.	No
Abstract Factory Pattern	Yes, but limited in predicted structure.	No capability to add new structures by business people directly. New product families must be coded by developers in advance to take effect.	Yes, but only if all future concepts can be predicted and coded into product families.	No	No
AOM (+Adaptability Aspects Pattern)	Yes	No, business abstractions are not represented straightforwardly and its architecture is difficult to understand and maintain (even to IT people) – this difficulty can be alleviated by using Adaptability Aspects Pattern	Yes	Yes, but only if all future rules can be predicted and coded into strategies (functional rules) and the policy (policy rules) which controls the selection of strategies.	No, there is no explicit process modelling. Even where object interaction can be dynamic, processes can be hard to configure and maintain.
Coordination Contract	Yes, but limited in component coordination. Its use of contracts drives dynamic configuration of component composition.	No, configuration for changing effect must be carried out by IT people on writing new or altering existing contracts, in a programming like manner with no support to configure diagrammatically	No	Yes, but only if new rules can be expressed as contracts.	No, there is no explicit process modelling. Even where component coordination can be dynamic, processes can be hard to configure and maintain.
Adaptation classes	No	No, configuration for changing effect must be carried out by programming language experts on writing new adaptation actions that are to be encapsulated in adaptation classes. New adaptation classes must be coded by developers in advance to take effect.	No	Yes, but only if new rules can be predicted and coded into adaptation classes (functional rules), and expressed in the execution context under which the adaptation class compiler switches between behaviour (policy rules)	No

The summary given in the table describes how well these approaches provide software adaptivity. None of these offers both of the top level goals identified by AAM, as shown in column 2 & 3 of the table. Moreover, none of these offers all the three adaptivity targets identified by AAM in business knowledge models used by software systems, as shown in column 4, 5 & 6 of the table. Nevertheless, the idea of using externalisation and metadata to convey runtime variability demonstrates the viability of such a means as reflected in the investigated approaches. For example, Strategy Pattern uses context/policy to dynamically select strategies. Coordination Contract uses contracts to enable dynamic inter-component interaction. AOM uses metadata to avoid being confined to predefined system structures and behaviours.

Recognising the usefulness of externalisation and metadata on one hand, the common weakness of all of the sample approaches and others must be considered and overcome on the other hand. One of the primary motivations of this work is to bring about easy software adaptivity and maintenance through using agents coupled with maintainable knowledge models, directing dynamic agent behaviour. Some discussed OO approaches actually do involve simple business knowledge models for adaptivity, in a limited manner though. For example, Strategy Pattern uses policy rules and Coordination Contract uses coordination rules. Business knowledge models must be extended in other dimensions for greater adaptivity. With respect to computational components however, a dramatic change is necessary. Agents are in contrast with objects, which have static structure and behaviour, being seen as inherent defects in achieving software adaptivity. Nonetheless, what it is proposed is that existing business investments in OO infrastructure need not be lost. Rather, already developed components can be (re-) used to facilitate the operation of agents according to the current needs as captured by the knowledge models. Agents, consequently, become a higher level of abstraction over objects. Improvement and new development on the two aspects discussed above correspond to the aims of AAM for adaptivity. In the next chapter, agent-oriented computing technologies and agent-oriented business knowledge modelling techniques are discussed. They serve the knowledge background and the foundation upon which AAM is built, mapping to AAM's two compositional hierarchies already discussed in Figure 1.1.

Chapter 3: Literature Review: Agent and Business Model

3.1 Introduction

This chapter includes reviews of relevant methodologies and techniques that support the building of the Adaptive Agent Model. The areas of interest largely fall into two categories: agent-oriented software engineering and business knowledge modelling. Major work in these areas has been investigated. The adaptivity aspects of using business modelling will be discussed with more details in Chapter 5. In this chapter the relevant existing knowledge on agents, agent-oriented software engineering methodologies, agent-oriented modelling languages, agent-oriented business knowledge modelling is discussed. Related to this, it should be noted that, since some of the described concepts have no agreed definitions as yet, some notions such as "agent" have been defined so as to be relevant to the aim of adaptivity, but based on existing definitions. The chapter continues with a discussion of business knowledge modelling, including an examination of current thinking on business goals, processes, rules and concepts.

3.2 Agent

The significance of agent technology has been increasingly recognised in the last few years. Although agent technology stems from Artificial Intelligence (AI), the recognition of it as about Computer Science and Software Engineering more than it about AI is imperative. Otherwise it will suffer a comparable fate in the future of the many and varied criticisms of AI, in particular in its perceived failure to delivering its promise [167]. It has been claimed that intelligent agents are ninety-nine percent computer science and one percent AI [171]. Agents must be used in a pragmatic way in software development, supported by engineering tools before their acceptance as a mainstream technology. Whatever, nowadays agents have already been credited as an advance in Software Engineering abstractions, after the appearance of other abstractions such as procedures, data types, and objects [40]. Attempts that cast agent-orientation as the next major Software Engineering paradigm include [168]

[169] [170] [167]. In addition, agents have been found useful on a conceptual level at the requirements elicitation stage in [27], where they are used to guide modelling and design, just like objects have been in object-oriented approaches. A useful survey of agent and agent-oriented Software Engineering can be found in [184].

Although a general consensus on the definition of an agent is hard to reach, the following has been found useful by many [167].

An agent is an encapsulated computer system that is situated in some environment, and that is capable of flexible and autonomous action in that environment in order to meet its design objectives.

The use of a single agent is not sufficient to exploit the full advantage and potential of agent technology. Rather, multiple agents are usually applied in a distributed and decentralised environment, called a Multi-agent System (MAS). Agents are generally attributed with *autonomy*, *reactivity*, *pro-activeness* and *social-ability* [167] [170], properties associated with the agent definition given above. Accordingly, in order to let the software system meet its requirements, it is proposed that a MAS can be specified to include the following components in general [167]: i) the *beliefs* agents have (agents are situated in an environment and have information about the environment and other agents within). ii) The *ongoing interaction* agents have with their environment (agents are reactive systems that continuously interact). iii) The *goals* that agents will try to achieve (agents are pro-active and their behaviour is directed by the designed objectives). iv) The *actions* that agents perform and the effects of these actions (agents have no complete control over their environment but can influence the environment by performing actions).

Agents have intentional behaviour and such behaviour can sometimes be explained by the agent attitudes of belief, desire and intention. The belief-desire-intention (BDI) model [172] [125] has been proposed as a typical conceptual agent framework that satisfies the specification given above. *Belief*, *desire* and *intention* are key data structures that constitute BDI agents. BDI agents have beliefs that represent the information they have about the environment; desires that represent (possibly inconsistent/conflicting) tasks agents want to achieve; and intentions that represent the chosen set of desires agents commit to achieve with respect to their consistency and resource limit. Temporal logics [175] [176] can be used to represent the agent states, the agent reasoning towards goals, and so on for the MAS specification. In the case of BDI, BDI logics [173] [174] have been formalised.

However, it has been pointed out in [167] that it is difficult to transform a formal specification of a system in terms of beliefs, desires and intentions to a concrete computational system. Nevertheless the BDI model and BDI agents are useful for modelling complex systems conceptually.

Others propose agent specification differently, using the Z language [177] [178] [179], with varied agent notion. A related framework of Structured and Modular Agents and Relationship Types (SMART) can then be described. Its use of a four-tiered hierarchy of elements present in a MAS under construction from bottom to top would be: i) *entities* are inanimate objects that have only attributes; ii) *objects* are entities that have capabilities; iii) *agents* are objects that have goals; and iv) *autonomous agents* are agents that have motivations. This specification has built-in inheritance relationships among its layers of elements and so is easily supported by existing programming languages for building functional systems. This is achieved at the cost of its limited support of the reactivity and pro-activeness features of agents [167]. Nevertheless, it demonstrates the potential of transforming from agent-oriented specification to traditional object-oriented implementation, thus gaining the benefits of its wider support.

The appropriateness and advantages of using agents and agent-oriented paradigms have been put forward in [168]. Their advantages become especially obvious with respect to engineering complex systems. This is due to the three mechanisms that agent-orientation uses to manage complexity: *decomposition*, *abstraction* and *organisation*. Firstly, agents can make runtime decisions and their interactions are not bound at design time. Rather, they act and react together dynamically without prescription but towards their objectives. This nature makes agent-oriented decomposition suitable for complex systems, where the overall functionality is emphasised and change is normal. Secondly, agents interact with each other using knowledge level agent communication language and this minimises the potential semantic gap presented in current paradigms, such as in OO systems, between their use of class method invocations and their actually represented conceptual business conversations. The semantics awareness and knowledgeable nature of agents makes agent-oriented abstraction suitable for complex systems. Lastly, agents can interact dynamically to reflect evolving social and organisational associations. Agents and agent groups can also be developed relatively in isolation and added to systems incrementally.

Since complex systems tend to change more frequently in a wider scope and are more difficult to manage, adaptivity is crucial to such systems. In relatively simpler systems, where agents may not even be the most appropriate constructs, adaptivity may not be an urgent necessity either. Therefore, an agent-oriented approach is particularly suited to achieving software adaptivity emerging as a critical business need in large and complex business systems. In fact, it has been argued in [197] that adaptability is not an emergent property but a fundamental characteristic of complex systems, and the decomposition of them into a set of adaptive agents lets the systems react and adapt better in changing environments.

Because agents have dynamic behaviour and are conceptually different from the static constructs applied before, new concepts and meta-models are proposed in relation to agents. For example, a meta-model for agents, roles, and groups has been suggested to expose patterned behaviour of agent societies in open environments. It facilitates dynamic, controlled, task oriented agent group formation and so enhances the predictability, reliability and stability of the overall MAS and its analysability [186]. In this meta-model, each *Agent* instance can be associated with one or more *Agent Classifiers*, sub-classed into *Agent Physical Classifier* and *Agent Role Classifier*. The former defines the primitive properties and capabilities of agents and the latter defines the various roles that agents can play. *Role* is a concept related to agent elsewhere, for example, in Gaia and AUML which will be discussed later on. Agents are usually linked to each other by the roles they play with respect to the requirements they are expected to jointly achieve. In the proposed meta-model, agents can be dynamically assigned to Agent Role Classifiers as required to gain features in addition to their primitive ones that were obtained independently through the assignment of Agent Physical Classifiers. Roles are qualified by and given meaning by the group context. A *Group* is a set of agents that are related via their roles, or say, a composite structure of Agent Role Classifiers. The conceptual model allows a high degree of diversity and dynamics and at the same time predictable patterned behaviour of agents. For example, two distinct agent instances can play the same role only in their respective qualified group context, but not in the other agent's group. That is, the same role can be played in different groups by different agents. Actually, the agent, role, group meta-model enables a three-way association, in which an agent is assigned to playing a role within a group. One particular agent can play different roles in different groups, where what it is expected to perform depends

on the context of the group. One group can be dynamically formed by agents that collectively have the capabilities to accomplish certain emergent tasks. Such a means of organising agents according to the dynamic tasks they are supposed to carry out has its potential in handling complexity and bringing flexibility.

Despite all these arguments in favour of agent and agent technology, and some existing agent-oriented conceptual models and methodologies already being used in real domains, research in this area is still relatively immature and has a long way to go. In particular, a proper agent-oriented methodology towards software adaptivity with full reuse of existing development techniques and knowledge is needed to meet the business challenge of constantly volatile needs, but so far has been paid little attention.

Specifically, when an agent-oriented solution is adopted, perhaps in an attempt to take the advantages described previously, its compatibility with existing techniques must be taken into account. One of the reasons that agent-orientation is considered as the natural evolution of the object-orientation and expected to eventually take its place in mainstream Software Engineering is that agents may reuse legacy software systems [169]. This can be achieved by wrapping agents around legacy code. The wrapper agents present agent interfaces to other agents and embrace functionalities of the wrapped code. They on one hand take and map external requests to calls in legacy code, and on the other hand interpret the results into an agent communication language to inform other agents. The integration of agents and objects compatibly in the same systems can be enabled in this means. Therefore object-orientation and existing object infrastructures need not to be abandoned because of the adoption of agent-orientation. Rather, agents can add adaptivity to objects.

The extension from object to agent also provides a benefit from the more intuitive business representation and the more straightforward execution of the business (requirements) brought about by a software system using an agent abstraction. A key concern of the current software development is the representation, implementation, and execution of business models as system specification in software systems. Translation between business models and software system models is believed to be possible [54] [53]. In Rational Unified Process (RUP) [54] [55] it is asserted that the current object-oriented concepts and frameworks underlying the software development process can serve business process representation and cover the overall business modelling with some extension. Agents are a natural connection point

between the two models. On the one hand they represent at business level responsibility carriers of business activities required by system specification, and on the other hand they are software units that realise the business models associated with them at the system level. An agent-oriented software development methodology in which agents abstract system requirements conceptually and system functionalities realistically is advocated here. In this case, software development starts from agent-oriented requirements modelling, business models being captured and associated with agents, and ends up with executable agent software systems.

A review of commonly recognised agent characteristics is found in [52]. Generally, an agent-oriented methodology is developed from an extension of existing methodologies. The two main such methodologies are object-oriented (OO) focused ones, where agents are considered as active objects, and knowledge engineering (KE) focused ones, where agent knowledge is modelled. Most current approaches only focus on the extension of one of them. This work inherits many usual agent characteristics from its heritage and jointly extends and combines the two methodologies. Specifically, an Agent Model executes business models, requirements of business and knowledge of agents, supported by a lower layer of object components. By using knowledge held in their business models, agents know what, when, and how components are to be used.

3.3 Agent-oriented Software Engineering (AOSE) methodologies

Given the advance of agents over existing constructs, some agent-oriented Software Engineering (AOSE) methodologies such as Gaia [41] [121], MaSE [122], Tropos [123] as well as many others [124] have been put forward.

Probably one of the most well-known one among these, the Gaia methodology provides a set of agent specific concepts through which complex systems can be understood and modelled [170]. Such understanding is captured in system organisation, being a collection of interactive roles. “Role” is a principle concept in the Gaia methodology. Roles give agents well-defined positions in the organisation and the associated behaviour that can be expected. Individual agents take on various roles for different purposes. Four attributes are defined for roles, being responsibilities, permissions, activities and protocols. A role can perform

functionalities (responsibilities), have rights to access resources (permissions), carry on computations (activities), and interact with other roles (protocols). The methodology proposes the production of several models through the agent-oriented design process: i) an agent model, where roles are aggregated into agent types, and the needed agent instances are created from each type; ii) a service model, where agent roles are fulfilled and associated protocols are served; and iii) an acquaintance model, where communication links between agent types are established. Although the conceptual framework is intended to be neutral and so applicable to a wide range of domains, Gaia requires that inter-agent relationships (organisational structures) and agent roles (capable behaviours) are static at runtime [184].

A major challenge in AOSE research is to use the agent-oriented software abstraction and derive practical tools [120]. Once agent models are built under the support of related methods and tools, the eventual goal of agent-oriented abstraction of computational behaviour is being realised in software programs. Methods and techniques are in urgent need that addresses issues over the entire agent-oriented software product life cycle in order to allow agent technology to enter into the mainstream [185]. Current AOSE methodologies usually focus on agent-oriented system analysis and design, from the identification of agent interaction protocols to message routing and communication. The need for complete upfront design and implementation makes it difficult to manage agent conversations flexibly and to reuse agent behaviour [165]. Although some research work provides the means for the specification of agent-oriented software systems, a complete development process is rarely discussed. Similarly, tools supporting the maintenance of deployed MAS are given limited attention. Developers who are familiar with OO systems have not been systematically assisted by a consistent and complete approach which transforms the initial requirements specification to a final executable MAS, making use of their existing engineering knowledge and skills. Further, since new concepts (e.g. goal, role, and organisation) are used in the construction of agent-oriented models, the original object component is often ignored. For example, the BDI (Belief, Desire, and Intention) agent paradigm [125] [172] uses an agent model and an interaction model from an external viewpoint, and a belief model, a goal model, and a plan model from an internal viewpoint. Therefore, later in the implementation of the MAS, existing components cannot contribute to the behaviour of agents, even though their encapsulated functionalities could be reused to support agents'

behaviour. Thus, one cannot easily take full advantage of the agent abstraction over object or adapt the behaviour of MAS through re-configuration of supporting object components.

Some AOSE approaches in the direction of using reusable modules such as OO components have been evaluated. Using Agent Patterns [126] is one way for better code encapsulation and reuse. In support of Agent Patterns, it is argued that much research work emphasises only the design of basic elements like goals, roles and communications, whereas the reuse of patterns, which are observed as recurring agent tasks appearing in similar agent communications, can reduce repetitive code. However, the chance that a pattern can be reused without change is low and reuse of patterns in different contexts is not straightforward. In addition, this approach is not adaptive since any system requirements change means that models need to be changed, patterns need to be re-written and agent classes re-generated.

State machines have also been suggested for agent behaviour modelling [127] and the Extensible Agent Behaviour Specification Language (XABSL) has been specified [128] to replace native programming language and to support behaviour modules design. Intermediate code can be generated from XABSL documents and an agent engine has been developed to execute this code. The language is good at specifying individual agent behaviour, but cannot express behaviour that involves inter-agent collaboration. Moreover, although agent behaviour is modelled in XABSL, it must be compiled before being executed by the agent engine. Thus, changing the XABSL document always requires re-compilation.

Agent behaviour is modelled as workflow processes in [22] and a Behaviour Type Design Tool is described for constructing behaviour. This approach provides a convenient way to compose agent behaviour visually. However, its use of Agent Behaviour Representation Language (ABRL) to describe agent interaction scenarios and “guard expressions” to control the behaviour execution order does not facilitate the modelling of systems as a whole. Further, the approach does not offer an agent system generation solution.

All of the above approaches promote module reuse but do not build an architecture suitable for the reuse of appropriate modules. Usually code change is still required, other complexities introduced, and the abstraction of agent over object not fully exploited. In response to this common weakness an MAS development

process is put forward later, which includes analysis and design and also implementation and deployment.

3.4 Agent-oriented modelling languages

UML is a de facto standard for object-oriented modelling. Designing and designating a completely new modelling language for agent-oriented modelling would be unconstructive and unproductive [181] [129]. Some have promoted the adaptation and extension of the trusted UML notation system for agent-oriented modelling. For example, support for expressing concurrent threads of agent interaction and notion of agent role may need to be accommodated into the new modelling language [170] [130] [129]. Work in this area is wide-ranging, including those of the OMG [137] and the FIPA.

One of the most influential modelling languages, Agent UML (AUML) [81] [129] [130] [131] [180] extends standard UML to cover the needs of agent-oriented analysis and design. In the context of agents and multi-agent systems, the AUML community has proposed two core diagrams: class diagrams and interaction diagrams. The latter consist of a family of diagrams: sequence diagram, interaction overview diagram, communication diagram, and timing diagram. The sequence diagram is the main one, being the most useful among the family and with more work being focused on it. Existing OO based diagrams have been enriched in AUML diagrams with new meanings. For example, class diagrams that used to drive object-oriented design are turned into agent class diagrams for agent-oriented structural modelling [182]. Agents are expected to have the same pattern of structures and behaviours if they are of the same class, being the level of diagram modelling. The concepts of agent, role, organisation/group, service, protocol, message, and so on with their corresponding notations have been accommodated into the AUML diagrams.

In agent-oriented modelling using AUML, one may start from assigning agents to groups and roles. Groups and roles provide agents who belong to them with common services and protocols in a means such as applying templates. An agent identity, along with its assigned groups and roles are contained in the compartments of an agent diagram, called an agent shell that allows agent features to be added. Other elements that the shell can hold include events that agents can perceive and actions that agents can perform to achieve their objectives. Services represent what agents

can do and can be offered from groups, roles, or individual agents. They are attached to the agent diagram either in circles with service names or separate class diagrams with detailed descriptions, the protocols invoking the services, and so on. Protocols support service cooperation and coordination. The message passing in sequences between agents are captured in protocols, which are central to AUML sequence diagrams. They look just like sequence diagrams in object-oriented modelling, and contain lifelines, messages, constraints, termination and so on. A protocol or an Agent Interaction Protocols (AIP) can be defined to describe an agent communication pattern in a pre-agreed message exchange style. They can be viewed as a specific class of design patterns describing problems that occur frequently in MAS and the corresponding reusable solutions [129]. Examples of AIP include Request, Query, Subscribe, Propose, Contract Net, and English Auction Interaction Protocols, all defined and documented by AUML [183]. One of such an example AIP is shown in Figure 7.14. Agents intending to participate using any AIP must adhere to the AUML specification. Levelling is used for refinement of the interaction processes.

The major drawbacks of AUML lie in its lack of semantics associated with its visual diagrams. This deficit can result in inaccurate interpretation of their meanings and also contribute to the absence of dedicated tool support [180]. What has been argued in Section 1.4 against UML also applies here against AUML. Both have the same flaw that their design models lose their value if they are not maintained after the development starts. For example, agents cannot execute interaction protocols simply by finding and understanding the relevant diagrams. Similarly they cannot process or send a message if it is required in its lifeline in the diagram, nor can they perform an action if it is marked through its thread of interaction in the diagram. They also cannot make a decision at a point if a decision node says so in the diagram. The reasons for these shortfalls are that agents do not have the capability to understand AUML notations. Model semantics comprehensible to agents and a mechanism of modelling driven agent behaviour is needed to advance model reusability and software adaptivity.

Moreover, AUML is seen as an extension of UML just like MAS has, rightly or wrongly, been seen as an extension of the OO paradigm. However, in AUML agents replace rather than co-exist with classes. Neither AUML nor UML support the collaboration of agents and objects towards system goals. This is somewhat at odds

with the philosophy of AUML that states: “When it makes sense to reuse portions of UML, then do it; when it doesn't make sense to use UML, use something else or create something new” [131]. Using this same argument, it would be natural to reuse existing classes and add agents associated with them. The integration of agents and classes in an AIP would provide a layer of OO components supporting agents. This philosophy would boost technology reuse and facilitate smooth migration from OO development to MAS development, assisting the wider adoption of MAS. This is especially true in an open environment where agents have to collaborate with existing objects already running in that environment. The need for agents not only to communicate with other agents in their own society but also objects implemented in various languages is becoming more and more urgent in the next generation of systems, where agent technology may be comparably dominant to object technology [132].

The Multi-Agent System Modelling Language (MAS-ML) [133], based on the TAO (Taming Agents and Objects) framework, also provides an incremental extension to UML for diagramming MAS comprehensively. OO concepts and modelling elements are preserved and Agent Oriented (AO) concepts and modelling elements added to complement existing ones. However, it remains at a conceptual and notational level with no contribution either in the form of a method to guide the abstraction of agents over objects or an approach to support the development of MAS using the modelling language.

Agent-Object-Relationship (AOR) [23] models show social interaction processes in organisational information systems in the form of interaction pattern diagrams. These model not only agents, but also ordinary objects, events, actions, claims, commitments, and reaction rules which dictate behaviour. AOR can also be viewed as an extension of UML for agent systems and is capable of capturing the semantics of business domains. Although AOR modelling introduces the additional element of “rule” over the AUML notation system for modelling agent behaviour, the construction and editing of rules are not in its scope. Moreover, how agents, objects and rules work together are not described adequately. However, it provides an appropriate notation system for the agent world and is adapted for use later in the thesis for conceptual modelling of agents, rules, classes, and their interactions.

3.5 Agent-oriented business knowledge modelling

As proposed in MDAA, business knowledge models act as a knowledgebase of agent behaviour and so must be built in an appropriate form and, at the same time, genuinely reflect the original business requirements. This is closely related with Agent-oriented Requirements Engineering. As introduced in brief in the overview of AAM section of Chapter 1, business goals describe what agents are expected to achieve. Business processes organise agent groups that are aimed at shared goals. Business rules constrain agent behaviour in process participation, decision making, or strategy choosing. Business concepts prescribe vocabularies that agents can speak and understand. Agents themselves are the execution engine of the constructed business models. These concepts will be discussed generally in the next several sections, and their usage with respect to adaptivity will be further discussed in Chapter 5.

Briefly, on one hand, business models capture the requirements, which not only constrain but also guide the realisation of the ultimate business goals. On the other hand, cooperative running software agents interact with each other towards common goals [42]. Thus agents and business models complement each other: the former are the actors, the latter guide them and dictate the roles that they play. They work together to achieve business goals. At the Requirements Engineering (RE) level, agents are the container and business models are the knowledge that fills the container. At the running system level, agents put the captured knowledge into actions. Mapping from requirements to implementation for both the agents and the business models brings the traceability to the original requirements.

3.6 Business goal

The concept of agent being used at the Requirements Engineering (RE) level has been introduced in [25]. Agent is advocated as a guiding concept, similar to how object has been central in the OO approach in requirements elicitation and organisation. Many agent-oriented requirements engineering approaches are also goal-oriented. “Goal” is an abstraction that stakeholders are familiar with and are interested in [24]. It is suggested that goal-based reasoning is central to RE because the ultimate criteria for system success is that stakeholders’ goals are achieved and their concerns addressed [185]. The refinement of goals can help to build a

comprehensible requirements structure and, if combined with agent assignment, alternative agent-based system solutions can be explored [29]. Indeed both agent and goal have been recognised as being promising in RE [24]. Some agent-oriented RE frameworks have been put forward.

The *i** [25] [185] methodology models intentional relationships among strategic actors in requirements elicitation and analysis. The agent-oriented modelling framework captures strategic relationships among agents in the world and this allows users and stakeholders to understand and reason about the implications of alternate solutions and jointly make decisions during requirements engineering. The strategic actors in *i** are semi-autonomous units that depend on each other to achieve goals, perform tasks, furnish resources and so on. Alternatives can be explored by reasoning about the dependency relationships, since goals are explicitly represented. The Strategic Dependency (SD) model and the Strategic Rationale (SR) model are the two types of models in *i**. The SD model identifies a network of dependencies, expressing what some actors want and what abilities the other actors have to meet the needs. The model distinguishes four types of dependencies: goal dependency, task dependency, resource dependency, and softgoal dependency, assisting the decision on alternatives through the analysis of dependencies. The SR model is a graph of goals, tasks, resources and softgoals connected by means-ends links, through which goals are associated with alternative ways for achieving them, usually tasks. The model supports the reasoning of actors about their intentional relationships by systematically refining goals to explore the alternatives.

Composite System Design (CSD) [30] uses agents as composite components. In CSD, global goals of the system are decomposed until they can be assigned to individual agents. “Goals” are replaced by “responsibilities” and assigned to the agents during the design and agent behaviour is enabled with capabilities. Similarly, KAOS [31], a goal-directed RE framework, requires designers to refine goals until they are reduced to constraints that can then be assigned to agents. The Albert II approach [32] expresses functional requirements using formal statements and groups them around agents to define the admissible behaviour of agents. In Requirements Engineering Framework (REF) [33] [34] and [35], a recent RE framework based on the notion of Agent, Goal, and Intentional Dependency, high-level requirements are refined using an iterative top-down decomposition process, which is cyclically applied to eventually produce elementary goals, tasks and constraints. Stakeholders’

needs are analysed and various relationship among them revealed: involvement, dependency, sharing, clashing.

A commonality of these paradigms, which is also shared by the AAM, is that agents are used as a way for requirements organisation. Requirements for system features are collectively organised around agents as the basis to direct their behaviour. The goal decomposition methodology, however, could be subjective, and creative talents are required for appropriate and optimal production. More importantly, many of the investigated RE modelling approaches give no concern to the integration of their early phase RE models with the rest of software development. This may make the resulted models isolated, not being made full use of, and eventually losing the advantages originally proposed in their frameworks. In others words, practically useable approaches should be able to transform agents in the conceptual level that act towards goals as specified by the requirements model to agents in the manipulative level that actually execute the assigned model.

3.7 Business process

Business process is a central construct of business modelling and is closely related with the requirements [28], both concerning the realisation of business goals. Analysing and implementing business processes is tightly bound up with designing and implementing the supporting information systems [37]. Describing and structuring information systems using process-driven approaches has become predominant [14]. Computation, interaction, and other activities can all be regarded as processes in system modelling, where process is the new first-class structural object [68]. However, today's software engineers model business activities using constructs such as objects and interfaces, which on their own are not sufficiently rich for the representation of business processes. Therefore business process modelling languages such as ebXML [212], BPML [213], BPEL4WS [214], WS-CDL [215] have emerged. They enable the specification of communication and collaboration of participants. In a typical business environment, participants in a business process include employees, information sources, business units, computer systems, business partners, goods, even business processes themselves [68]. Relationships between process participants change from time to time, possibly caused by changing business goals. New associations may be formed and old associations broken. Business process which is prone to change needs to be in a digital form, being capable of

being changed [68]. The change of process itself encompasses change in all other lower levels: the use of data, individual participant behaviour, and their organisational structure. Business processes can be viewed as processes that are carried out by internal agents and external actors/agents of the information system being modelled, working in collaboration, using business units such as business rules and objects as resources and constraints, implementing requirements, and eventually bringing value to the business. From this perspective, it is expected that mutable business processes are enabled through the interaction of agents, their internal behaviour, and use of other business units. Agent-based business process management mechanisms proposed in [14] [45] [40] and [41] employ agents to implement, execute, manage, and monitor business processes, and these lead to agile system collaboration and changeable business processes.

EKD [26], a goal-driven business process modelling approach uses an "Actor-Role Diagram", where actors represent physical entities and roles represent the responsibilities for performing activities assigned to actors. The fulfilment of the goal in each individual role collectively brings about the achievement of the overall process goal. In EKD, business goals shape business processes. A main purpose of this approach is better business change management, using the process of identifying initial goals for change, refining them, and mapping them to existing processes. However, the refinement of a newly required feature usually demands full understanding of the overall requirements and is, to a large extent, subjective. In addition, in some cases, it is not possible to identify any existing process related to the goal for change, and the new process must be defined from scratch in such cases. In general, each item in a requirements specification must have an objective that adds value for customers, otherwise it is useless. However, goals may not be the best criteria under which to group requirements. This is due to the fact that goals can be expressed at different levels of granularity, so that one requirement may span over several goals and a given goal may relate to multiple requirements. In contrast, it will be much easier to organise requirements around business processes that realise goals and let goals guide the organisation of the document and ensure its completeness. Since the number of the actual business processes in any given business environment is certain, this kind of organisation ensures complete coverage of all requirements and easy comprehension of them collectively, mapping to their actual runtime organisation.

An agent-based distributed business process model is proposed in [10] towards business integration. For business alliance, each participant member enterprise (ME) uses four kinds of agents that reside inside them: ME agents, activity agents, role agents, and resource agents. Business processes distributed over collaborative ME are executed by a global multi-agent system (MAS) consisting of only ME agents, the top level agents in a hierarchy of the enterprise model. The other three kinds of agents are: activity agents, role agents, and resource agents. These form the lower level of the hierarchy. All the four kinds of agents who are from that hierarchy and running within every single ME compose each individual local MAS. It is through them that MEs contribute to the overall distributed business processes. Although the approach achieves interoperability, the integrated business process is not adaptive. Any change to the goal of the integrated business process results not only to the change of interactions among ME in the first MAS, but also the reconstruction of the second MAS, due to insufficient encapsulation. Such a defect, as introduced when an agent and business process combined solution is applied across organisations, is raised here and Chapter 9 contains a discussion on how it can be solved by the AAM.

3.8 Business rule

In every organisation there are business rules, being compact statements that lay down what must or must not be the case in some aspect of a business [28]. According to the Object Management Group [137], business rules are “declarations of policy or conditions that must be satisfied”. They often not only express policies, but also refer to administrative processes which determine how several tasks have to be executed [37]. A typical rule can be as simple as constraining a value among a list of valid values, and as complicated as to control a business process depending on the condition of some attribute(s) or a current policy. Most likely, several business rules may together constrain one aspect of the business, for example, the execution of a business process. They are characterised by their strategic importance to the business and deserve special attention [47].

Business rules capture requirements [46] including the decisions, guidelines and controls which make up the functionality [39]. As long as inconsistency and ambiguity are identified, either in rules or by rules, they are proved to be useful for capturing and resolving conflicts, both at the requirements level [138] and at the design level [139]. However, to date, rather than made explicit as part of the

requirements document, their importance is ignored and often embedded directly into the final software product. This not only leads to the misinterpretation by developers with their own assumptions, but also provides no way to trace the rules in the code back to the requirements. In the end, the lack of explicit capturing of business rules contributes to rework and other inefficiencies [136].

Some suggest that business rules have the form of terms, facts, factor clauses and action clauses, and can be expressed in natural language using a tailored taxonomy [136]. According to this suggestion, if business rules are wrong, or subject to frequent change according to business needs, then there will be a lot of maintenance work. Thus, better still is the case where business rules are present in a structured natural language format and also executable by the system. Finding a way to make these rules executable would contribute significantly to the solution of transforming functional requirements to the final product.

It has not been agreed in the OO community where rules should be put in OO models. It is pointed out in [12] that the current situation where rules are spread out and converted into methods is a weakness of OO approaches. Although rules are used by the Unified Modelling Language (UML) and the Rational Unified Process [54] advocates them in presentation and conversation, sufficient guidance for business rules has never been provided [136]. The only practical representation of rules is by using the constraint element of Object Constraint Language (OCL) in the standard UML. However, this is at the design level as opposed to the requirements level where the interface is with business people. Another stated deficiency of OCL is its lack of rigor, which might lead to ambiguities [37]. Moreover, its elements are static, externally connected to the main models as an “add-on” [80] and in isolation in many cases. It is argued in [11] that what is important to business rules is that they are not presented as isolated elements, but linked to other elements comprising the enterprise model. To properly derive business rule requirements as business assets and let them remain valuable to stakeholders, they should be elicited and validated in collaboration with business customers during requirements analysis. For example, use cases are central in the UML and can have business rules linked to them. Linking business rules to use cases or steps within them provides better structured requirements than traditional use case narratives, since essentially business rules are at work behind use cases. Unless all business rules that underlie the use cases are captured, the software under construction will fail to meet the required business

needs [136]. Moreover, the use of business rules can compensate for the lack of capability in existing UML diagrams for capturing behavioural semantics. Nevertheless, an even better alternative construct of Business Process Rule, rather than use case, has been suggested in Section 6.11 in structuring business rules in the agent context and being used by the AAM as a first-class citizen, an argument being advocated in [136].

3.9 Business concept

Business Concepts in business domains are tightly associated with business objects in the supporting information systems. According to OMG, "a business object captures information about a real world (business) concept, operations on that concept, constraints on those operations, and relationships between that concept and other business concepts. The business concept can then be transformed into a software design and implementation. And a business application can be specified in terms of interactions among a configuration of implemented business objects" [38][62]. Therefore, mapping a business concept in business models to a business object in information systems is a viable and sound solution. The typical current practice is that after the implementation of models, the transformation of business concepts to business objects is finished, and business concepts in business models are never maintained. In that case maintenance is only performed on the implemented business objects, and thus business models lose their value [3]. Changes required to the information systems, however, always originate from changed business needs. MDA proposes to convert from reusable models to executable software systems, and hence the maintenance effort is greatly reduced. To this end, business experts should be able to assemble "plug-and-play" business objects without the help of software engineering experts, so that the flexibility and extensibility of information systems is enhanced [37].

Intra-agent function and inter-agent dependency could be externalised as metadata that serves as a context in which agents use to interpret their behaviour dynamically at runtime. Such metadata consists of structural and behavioural relationship of business concepts. Similarly, ontology is computer-readable description of knowledge about the business concepts present in a business environment. Attributes of different classes of concepts can be related and such fundamental knowledge established in ontology. The ontology forms the heart of any knowledge

representation system. Ontology underlies knowledge and offers a consistent vocabulary for representing knowledge [141]. When ontology is bound to an agent system, all agents agree to use this shared vocabulary and so that knowledge among the agents is shared and reused. Agents can make use of the knowledge contained in ontology in their internal decision making or external collaboration. Ontology modelling provides a means [140] that agents can use to reason about their representative business roles in performing and communicating by using ontological knowledge, since the business knowledge is explicitly modelled and linked to agent responsibility and duty. Both business processes and business rules are composite and complex knowledge built upon ontology, providing a context in which rationale behaviour is supported.

3.10 Conclusions

This chapter has reviewed the current knowledge of agent-oriented software engineering and business knowledge modelling. The combination of both areas can contribute to the building of the AAM methodology in order to achieve software adaptivity. It has been established after investigation that existing approaches in each area alone do not provide a comprehensive solution. In response to this, the next chapter will state the problems that AAM should address according to the evaluation of the weaknesses in the discussed approaches. The following chapters presenting the AAM approach and evaluating of the approach will cover the solution to these problems.

Chapter 4: Problem Statement

The problem of adaptivity in software systems has been described in the previous chapters. Agents have been put forward as a potential solution. However, based on what is known about the state of the art, it is proposed the following problems need to be addressed.

4.1 Need for a new MAS approach for software adaptivity

The major difficulties in building an adaptive object-oriented application include adapting component functionalities and dependencies. Existing component technology and language support provides no direct means of adaptation. Objects have fixed methods and patterns of interaction that pre-determine their runtime behaviour, once they are constructed. The traditional static structural and behavioural nature of objects results in their limited adaptivity. Various object-oriented approaches attempt to bring flexible object behaviour, however in the process they add other difficulties such as increased complexity. Although MAS approaches are promising in using agents to contribute dynamic behaviour, few current approaches, if any, allows the reuse of appropriate modules and reconfiguration of them. Runtime system adaptation is hardly achieved, code change still being required, and other complexities being introduced. A new MAS approach that addresses these problems is needed.

4.2 The approach should not abandon existing technologies

Agents can be considered to be an evolution from objects. In achieving agent-oriented system adaptation, technological changes should be minimised, wherever possible, to reduce the overhead cost that stakeholders may be unwilling to incur. Existing investment on object-oriented infrastructure should not be wasted, but developed components reused wherever possible. This would reduce the migration cost to an agent-oriented paradigm. A practical approach, which keeps existing objects developed by widely accepted languages such as Java, to support the running agent system would be easily adoptable by the well-established OO community.

Use of metadata has been proposed elsewhere to reduce duplicated code and hence alleviate programming tasks [142]. However, a typical technique used is code

generation which requires that generation is done with every significant change. Manual changes to generated files must then be remade or lost at the next code generation.

Using existing technologies includes the ability to reuse existing business knowledge. Business rules represent one way that this knowledge may be represented. Business Rules form the metadata that capture requirements and encapsulate agent knowledge in a re-configurable form. The dynamic re-interpretation of metadata by agents would bring immediate effect to the running system without interruption. Business goals and processes could be used to organise agent groups with shared objective and business rules and concepts guide and constrain individual agent behaviour. Also, these modelling units could be used to specify the existing object components that agents can use dynamically to achieve their objectives. An ideal solution that combines agent-oriented systems and business knowledge models would maximise the reuse of current business assets and minimises the overhead of adding adaptivity to the software system.

4.3 The approach must support a complete MAS development process

Many agent-oriented requirements modelling approaches are only concerned about the early RE phase of the requirements elicitation and analysis. Others, such as existing agent development platforms, provide a means to build agents using traditional object-oriented languages but no explicit support for the requirements engineering specific to the context of agents. Consequently, the disruptive separation of agent modelling in the conceptual level and agent development in the operational level, with no link between them or transformation guidelines, would negate the potential advantages brought about by agents. On the contrary, an integrated process that supports the full agent-oriented software development process can, at the same time, put MDA into reality in the context of agents. This integrated process would facilitate the later maintenance phase of the life cycle of the developed agent system.

4.4 The maintenance cost of software built using the approach must be minimised

The primary motivations for this work are to reduce the cost of maintenance and, related to this, the time required to carry out maintenance. One way to achieve this is to empower users, rather than developers, to take responsibility for evolving software. AAM will aim to achieve this through maintainable models, built so as to enable business people to configure their system. After all, business people are typically the primary stakeholders, who know how the system should change, and should therefore be the ones to take the initiative. Such a philosophy would reduce the delay and cost in introducing change. Knowledge of the system should be maintained via easy to use supporting tools, diagrammatically or textually controlled at run-time. Ultimately (and somewhat idealistically), the cheapest option would be to have the software completely self-adaptive. Relevant to this idea, some characterise flexibility as active or passive [144]. Active flexibility relates to fast modification. Passive flexibility is built into the system, and so its design requiring inherently less modification. Figure 4.1 illustrates how some existing approaches map to a schematic effort scale.

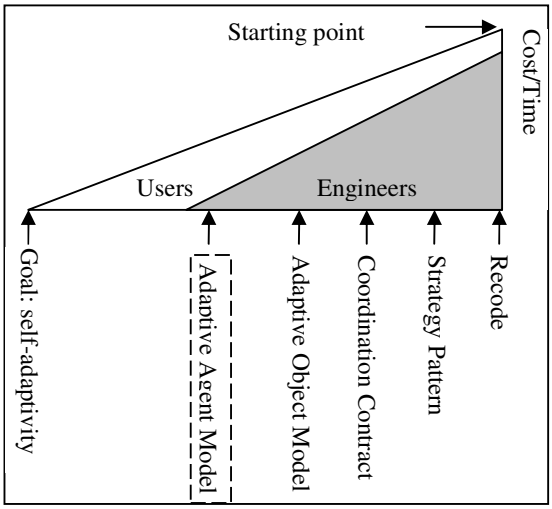

Figure 4.1: Cost/Time and responsibility in sample approaches to adaptivity

Figure 4.1 shows the worst case in terms of cost and time, on the right hand side, being where developers recode manually. The ideal situation where adaptation costs nothing appears on the left side, the system adapting itself without human intervention. In between are selected approaches schematically mapped to illustrate

the relative cost/time for modification. The figure also shows that a reduction in cost/time can be achieved to some extent by a shift in responsibility from (expensive) engineers to the (less expensive) users of the system.

4.5 Objectives of research

To summarise, the objectives of the research are listed below.

1. Provide a practically useful agent-oriented software development methodology that supports full development cycle, covering from requirements transformation to maintenance and deployment.
2. The approach should use business models that are easily configurable by business customers to reflect changing business requirements, and immediately executable by agents to bring the required new effects to the running system.
3. Existing object-oriented infrastructures in business environments should be reusable, supporting agent behaviour and being dynamically selected for function according to business knowledge.
4. The resulting Model Driven Agent Architecture should enable model level maintenance of the system, dramatically reducing the maintenance efforts traditionally required at code level.

4.6 Conclusion

This chapter has reviewed the outstanding problems that must be addressed in MAS development, if adaptivity is to be substantially improved. In particular, the objectives of the research towards that aim have been summarised. The next chapter will provide an overview of the building blocks used to construct the Adaptive Agent Model. The aims of AAM will then be introduced prior to a full solution description.

Chapter 5: Overview and Aims

5.1 Introduction

This chapter provides an overview of the Adaptive Agent Model with respect to its design. The AAM components and their hierarchical structures will be discussed in line with aimed adaptivity and recognised design disciplines.

5.2 Architectural components aimed at adaptivity

Given the aim of adaptivity, it is worth investigating scopes of a typical information system where adaptivity could be achieved. In doing so, two major steps are proposed: i) the separation of executable system components from the knowledge that supports the behaviour of these components; and ii) the hierarchical decomposition of both the components and the knowledge model. A related work published on this specific topic of the AAM can be found in: [162].

Firstly, a system contains structure and process. In fact, structure constrains processes. Structure-oriented change is likely to cause higher costs because such change affects all processes that access the changed structure. Process oriented change, on the other hand has lower associated costs because such change remains local to each process component. Roughly speaking, their distinction is generally recognised as the distinction of the fixed and the variable system parts [145]. Information structures in object-oriented systems are represented by objects, being documented in UML diagrams such as Class Diagrams. Processes are represented by object interactions, being documented in UML diagrams such as Sequence Diagrams. One way to reduce the high cost of structural change and also the process change is to keep the structure part content-free. This means maintaining the actual content, relationships and the interactions encapsulated externally. In doing so, the level of the generality of the composition of the system increases. Also, the knowledge of component interaction and such interaction processes is separated from the components themselves. The separation of structure from process makes systems more flexible. The result is that change to components is facilitated by the independent maintenance of the knowledge about their structure and behaviour. This is a shift from traditional recoding to diagram or text-based edition and boosts

efficient adaptivity of the overall system. When knowledge is changed, the components ideally can automatically retrieve the updated knowledge to behave accordingly.

Secondly, in the context of this research, executable components can be classified as agents and classes. Agents are active components that choose how to behave. Classes are passive components that have fixed behaviour. The combination of both in a system and allowing agents to be the first class citizens that invoke the appropriate classes in various contexts increases reusability and adaptivity.

Finally, the knowledge model consists of guidelines for system behaviour. For example, the knowledge model tells agents how to interact with each other and use classes, what organisational constraints they must obey, or what language all agents shall agree to speak. The knowledge model can be broken down essentially into a hierarchy that captures agent knowledge from different levels. This is an emphasised subject in the remainder of the chapters and is the basis on which AAM achieves adaptivity, as illustrated in Figure 1.1.

5.3 Knowledge models

Components modelled in the knowledge model should by nature be capable of handling change, or be configured to do so. AAM assumes three levels of adaptivity, as illustrated in Figure 5.1. Firstly, at the top of the hierarchy is the *business process*, being composed of the actions involved and their sequential and conditional control. For example, an online ordering process may involve several participants, actions, and control flow. This knowledge should be flexible and easily changed. Secondly, inside each action node in a business process, the decision making process should be dynamic. This middle layer of the hierarchy requires that *reactions* (selected depending on the execution context) and also new reactions (arising to cope with new conditions) take effect immediately after their specification. For example, various reactions may be required to deal with an order, depending on the nature of particular order, the customer who places the order, and other emergent conditions. In addition, new *policies* must be applied whenever they become available to the system, and become applicable at the time of execution. For example, different product promotions and special discounts may be applied to different types of customers (ordinary, VIP, etc.) at different times (e.g. off-season). Thirdly, the vocabularies or *concepts* used by the system, such as by its business rules, should be

extendable. This means that new concepts and new rules using them, defined at runtime, should become usable immediately after their addition to the system. For example, a new concept of "credit" may be required as an addition to allow the assignment of a special discount to customers with credit up to a certain limit.

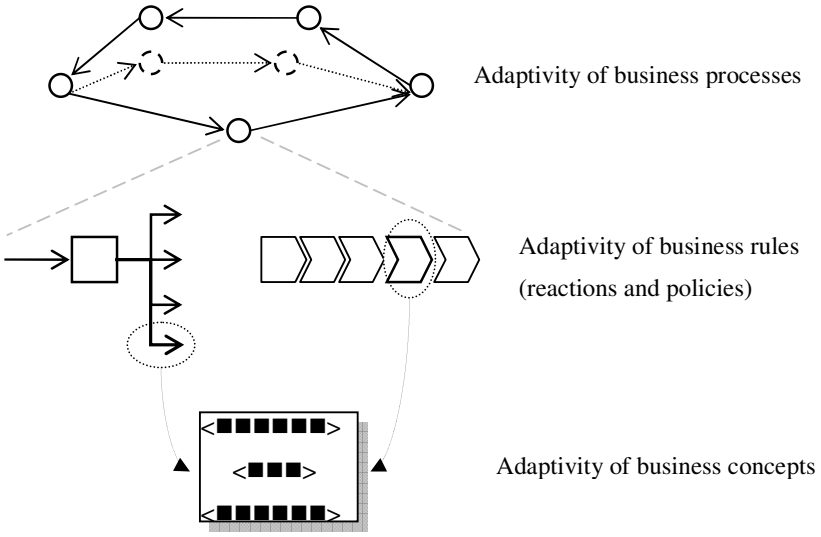

Figure 5.1: Adaptivity perspectives

The compositional components in the presented hierarchy have been introduced in Chapter 3. They are now revisited briefly and analysed with respect to the potential they have in introducing adaptivity to information systems, illustrated by recent and relevant research trends.

5.3.1 Business process

The focus of building information systems shifted from data models, emphasised from the late 1970s to early 1990s, to process models since the middle of the 1990s, resulting in the increased recognition of the importance of process-awareness information systems (PAIS) [150]. This is the natural consequence of the increasingly high demand for visualising and understanding complex individual processes within organisations and integrating distributed processes across organisations. Such demand is enabled and fed by advanced and relevant technologies available today. A PAIS can be defined as a software system that manages and executes operational processes involving people, applications, and/or information sources on the basis of process models. A process is used to organise work and resources to accomplish its aims. People and applications are not

necessarily distinguished. Instead both are considered here as agents that aim at process goals.

The advantage of process-oriented information systems is evident, the chief benefit being that systems become more adaptive to changes and configurations. Since the logic of business processes is explicitly modelled rather than spread across applications and procedures, they are better understood and much more easily reassembled or redesigned. Template business processes, for example, can be reused and adapted textually or diagrammatically, according to the variation of standard processes in order to meet changing business requirements. Every runtime instance of a specific process enacted automatically follows a predefined manner, which is indicated by a chosen template populated by the current runtime parameters. Necessary partnerships, tasks, procedures, and so on are selected and associated in this process dynamically. The capability to adjust process templates and the runtime data population of specific templates brings two-dimension adaptivity to systems. Business procedures, partnerships, and everything else can be easily scalable to suit various business contexts. This paradigm becomes more effective and productive when a model-driven approach is used that aligns the business and the technology. In doing so, a separate component of process management can be dedicated to the business managers and experts who are responsible for the maintenance of the process model towards its current needs. Business systems consisting of the processes, underpinned by the software systems which can automatically interpret the processes, are eventually driven by configurable models rather than fixed code. This makes them easier to be controlled and managed.

Current PAIS approaches and tools support either only the design, only the execution, or both but are restrictive in specific types of processes [150]. Such incompletion hinders an integrated development process for information systems. A new approach covering both design and execution of processes generically is therefore needed.

The core of business processes are usually coordinated by smaller business model components, the business rules. They determine the business with respect to one or another aspect of business processes and are subject to frequent revision.

5.3.2 Business rule

Changes in business rules are a normal trigger of system modifications. Keeping business rules distinct from code has many advantages, including the fact that they can remain highly understandable and accessible to non-programmers. XML-based rules have been used in the IBM San Francisco Framework [143] as templates to specify the contents and structures for code that is to be generated. With this approach, changing of XML rule templates allows mappings to new object structures. Figure 5.2 shows an example, where a generic XML rule has been converted to a specific Java method, getDiscount () in this case.

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

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

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

Figure 5.2: Example of code generation using business rules [143]

Business rules should be easy to change [58]. It has been suggested that they form a bridge between specification and realisation [28], and that business rules at the business level should be linked explicitly with their implementation at the system level [11]. In doing so, any change to the business environment leads to the change of the system through the amendable rules, which could make the system very flexible. The business rules in XML format as shown in Figure 5.2 demonstrate they are a promising candidate for achieving adaptivity in the object world. Because agent behaviour is a means to implement the system requirements, and is also subject to change, the application of business rules encoded in XML to the agent world should offer similar advantages as in the object world. Business Rules Markup Language (BRML) is such a candidate, having been demonstrated as useful for representing contract rules in the E-Commerce domain [196]. However although XML can be considered as a form of database, it is normally used as a form of business logic and knowledge representation rather than a form of storing persistent business information. Hence, database system and SQL language are not the right means to

support the management of requirements captured by the business rules and describe how such data should be processed. This is because their representation of business concepts, rules, and processes is static and not straightforward, business people finding them difficult to understand, communicate, or manage their business and maintenance cost of them being high. The XML encoding of business rules is convenient for agent processing but must have a collective visual representation of their overall structure for human comprehension. UML and supporting tools are therefore needed to complement the XML representation of individual rules.

Actually, OMG's Business Rules Working Group (BRWG) was chartered in 2002, working towards adding business rule capabilities to the UML [154]. This was in response to the lack of mapping from the Computation Independent Model (CIM) to the Platform Independent Model (PIM) in MDA or existing UML techniques. It has been agreed that UML augmented with business modelling capabilities is the best candidate for CIM-PIM-PSM representation and auto-transformation in MDA. Business rules will certainly play a crucial role in bridging business (models) and IT (models), being controlled directly in business organisations for business purposes, and transformed to executable code. Following a Request for Proposals on Business Rules Representation in 2003, the Production Rule Representation (PRR) group has been chartered by the OMG to specify a meta-model for rule representation. This work in progress was presented in 2005 [156], with some current work on rule representation in XML format such as RuleML. The importance of business rules has also been recognised by other major standardisation organisations. More recently, Rule Interchange Format (RIF) Working Group [155] has been chartered by the World Wide Web Consortium (W3C) [216] as part of its Semantic Web activities. The goal of the working group in using business rules is extending or complementing the OWL Web Ontology Language, representing business knowledge in information systems, and enabling the sharing of information in forms suited to machine processing. The group has just published the First Public Working Draft in 2006 [155] and the work is still in progress.

Researchers in this area have proposed some useful approaches. Constraint Description Language (CDL) has been proposed in [74] for providing support for automatic implementation of business rule constraints in component based, distributed systems running on Common Object Request Broker Architecture (CORBA) in particular. A subset of OCL, including invariants, pre-conditions, post-

conditions and guards are modelled in CDL. The aim is to generate improved skeleton code on the basis of the code generated by Interface Definition Language (IDL) compilers and reflecting the constraints specified in the UML design diagrams. Therefore, extra coding for adapting the original skeleton manually can be avoided. Despite saving development time and effort, generating code from business rules makes the maintenance of such systems more difficult, and the traceability of requirements related with business rules degrades. It is argued in [75] that business rules should be explicitly specified rather than lost in complex and fragmented software code, because few people with the business knowledge can check the code and any future change of the business rules would require the tracing of the implementation of the business rules in multiple occurrences in the code. In [75], business rules are used to represent some user requirements, being those considered to be the volatile parts of the software system. A "Link Model" is proposed to relate business rules to software design. Concretely, some operations in UML Class Diagram are linked with business rules that guide their implementation, and additional operations or other elements may be required to be created as the result of business rules. Consequently, the improved traceability from the requirements to the software components makes software evolution easier.

Transforming rather than representing business rules makes systems more vulnerable to change. It is therefore advocated that expressing business rules at an enterprise level model rather than encoded in programs. If rules are bound up in processing logic and tightly coupled with application code, the management and refinement of them would be a difficult process [59], as demonstrated by the high level of maintenance costs experienced in current practices. The establishment of a linkage from requirements elements to design elements by the Link Model improves the traceability and helps the refinement of design through business rules. Even better would be direct modelling of requirements in business rules and then *keeping* (rather than transforming) them as key elements in design diagrams. The models are later interpreted and implemented. Decision makers manage business rules directly and explicitly. When any change is made, they get reflected automatically in systems, rather than developers having to search through code. An example of where business rules are made explicit is proposed in the Enterprise Knowledge Development approach (EKD) [13].

Business rules, when used for business modelling, could be grouped to aid decision making [46], for example, in organisation of business policies. Such grouping eases the management and maintenance of a rule base but not automatic selection and execution. One approach [46] proposes to group together all business rules associated with the same use case and store rules per use case. A process is suggested that follows the rules, in an attempt to take the use case with which all rules are associated through a set of states, eventually achieving the goal of the use case. The authors claim that the efficiency of rule processing is increased because naturally grouped rules are executed together at the same time during user invocation. However, approaches like this require manual operations (e.g. grouping, execution) and tend to miss relevant rules for execution when they become applicable. This is due to the lack of a mechanism for automatic execution of business rules on detection of their relevance on a continuous basis. A rationale for automatic grouping and handling of business rules with an alternative construct (Business Process Rule) as a replacement for use cases has been provided in Section 6.11.

Overall, the eventual aim of taking a business rules approach to information system development is to automate the business processing through the business rules [46] as executable specifications [48]. Later a business rules modelling approach for MDAA is described. Such a solution should bring automation with the use of agents. There has been previous work, for example [22] [23], on modelling business rules in agent systems. However, these are not easy to implement and their rules are not configurable at runtime by business people. A newer approach, the Agent-Rule-Class (ARC) [1] [4] delegates an independent rule element for requirement and design modelling, sitting between agent and class. Agents select and execute appropriate rules at runtime. The externalisation of agent behaviour in business rules for agent-oriented development is detailed in [2] and [6]. It proves to be useful not only for easy migration of the current OO infrastructure to agent-oriented infrastructure, but also brings features otherwise unachievable.

The core of business rules are usually defined by smaller business model components, the business concepts. They determine the business with respect to terms that business rules refer to and are subject to frequent revision.

5.3.3 Business concept

A business object in a business model captures a business concept and expresses an abstract view of the real business world. A business object directly represents a business concept in software and helps to transform business ideas into software realisation [38]. Conversely, business concept modelling provides a means that explicitly expresses the corresponding business object components consist in the system to be developed.

Object technology is the centre of the current mainstream object-oriented development. The Object Management Group (OMG) [137] is established to promote the theory and practice of object technology for the development of distributed computing systems. The OMG, involving a wide range of software vendors, developers, and users, emphasises its effort on producing and maintaining industry standard for business applications. Its modelling specifications of MOF, UML, XMI, and CWM have built a foundation for its flagship MDA specification, recently underway but already well known. According to OMG, its Model Driven Architecture (MDA) "provides an open, vendor-neutral approach to the challenge of business and technology change. Based on OMG's established standards, the MDA separates business and application logic from underlying platform technology. No longer tied to each other, the business and technical aspects of an application or integrated system can each evolve at its own pace - business logic responding to business need, and technology taking advantage of new developments - as the business requires. [137]"

As early as a decade ago, the OMG jointly sponsored the annual Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA) – Business Object Component Design and Implementation Workshop in order to draw research contributions into the drive for building business objects as reusable and pluggable components [152] in a Business Object Component Architecture (BOCA) [160]. BOCA is the OMG's early effort in segregating organisations' core business information from the technological specifics of the systems upon which it resides. In brief, the BOCA architecture specifies a business component dimension and a technology component dimension, the latter connecting the former and supporting it. Specifically, BOCA uses business semantic contracts, which configure and structure business concepts, to describe aspects of the business domain but expose no

implementation detail. The Component Definition Language (CDL) in BOCA provides a common textual representation for defined contracts among the cooperating business objects. Definitions written in CDL on business objects include their interfaces, their structural relationships, and collective behaviour for groups of them. Frameworks provide execution environments for implemented business objects and enforce the semantic contracts across these business objects. The result is that, business requirements are transformed into contracts, which are in turn instantiated as framework-specific component artefacts. The use of contract abstraction, focusing on domain semantics and hence being stable, isolates domain specification from technology implementation, and preserves the integrity and interoperability of business objects in evolving frameworks.

The work published by the workshops includes various approaches towards adaptive software systems using business concepts/objects. The REA accounting model [159] applies an Active Object-Model, which is highly related to the Adaptive Object Model [17] [18], for adaptability. The main contribution and achievement of the Active Object-Model, with respect to adaptability, is its explicit modelling of business object types and the production of a Type Object pattern. The pattern uses two concrete classes. One represents objects and another represents their types. The Type class acts as a builder of the Builder pattern [15], where instances of the Object classes are products of the builder. This schema can better adapt to changing business requirements where new types of business objects need to be created after the deployment of systems. Another approach, the Dynamic Business Object Architecture (DBOA) [157], proposes a business object architecture with four layers. The layers are: business strategy, business process, object-oriented modelling, and database management system (DBMS). This layered architecture is intended to enable software better meet the expectations of business users. The business objects used in the DBOA are driven by business strategy. Business strategies are identified, modelled, and mapped onto supporting software systems built upon DBMS. It is recommended in the DBOA that business objects are reused in business processes and business strategies. The linking of business strategies and software applications supports the integration of business and IT. Others propose assigning goals to business objects, in addition to the usual attributes, behaviours and rules, so that they act as adaptive agents [158]. The goal-driven behaviour of agents can be kept within limits and decided by the changing needs by allowing the conditions to their actions

to be modifiable by users. All of these efforts demonstrate the role that business objects can play in adaptive software systems. In addition, the explicit business modelling involving the business concepts/objects modelling is demonstrated to be of great importance. The ultimate need for standardised business object component architecture is that object systems can be dynamically updated by changing the design and regenerating code, thus re-synchronising the system with the constantly changing business needs. This is, actually, the goal of the MDA.

According to [153], a business component represents the software implementation of an autonomous business concept or business process. It consists of all the software artefacts necessary to express, implement and deploy a given business concept as an autonomous, reusable element of a larger information system. In BOCA, it is demonstrated that the mapping from CDL, at a level of abstraction that expresses semantics among business objects, to framework implementations, allows the specification of business objects in terms of semantics not implementation and a wide variety of implementation choices. When the business concepts co-exist and are linked with the implemented business components, the benefits are obvious. The object types to be defined for the representation of business entities, events, actions, rules, and processes emerging from the business environment could be many and also constantly changing. Moreover, their relationship and interaction could also be very complex and frequently changing. The understanding of their semantics using an explicit model, especially in a distributed and interoperable environment, is therefore crucial to the delivery of business applications. Not only do business objects deployed in systems have higher reusability across systems (because their availability and usage is better known), but also they have enhanced traceability (because their semantics are made explicit). Transformation from model to system, as a requirement of MDA, is better supported in this manner, the business concept modelling element linking to the business object running component.

Although the emergence of object technology allows information systems to be more adaptable than before in the easier adaptation of their comprised business object units in better management, it needs to be noted that after a decade, when systems become more and more complex, the adaptation through objects becomes more and more difficult and even cumbersome. The shift from adaptive object technology to adaptive agent technology may be inevitable. The shift, coupled with

the separation of changeable from unchangeable, marks a much more dynamic adaptation mode, manifested not in code but in models.

5.3.4 AAM business model hierarchy

For the purpose of achieving adaptivity in all three levels and at the same time ensuring generality and technology independency, the AAM is intended to be a reference model that analysts can use to systematically turn requirements specification into a set of maintainable business models. Maintenance of the business models is the responsibility of business experts on a daily basis. The explicit modelling of business elements in various levels has the intention that business experts are enabled to adjust the system behaviour with an immediate effect, with no requirement of IT expert intervention. A supporting Agent Model, independent from business models and underpinned by object-oriented facilities, can be developed once and for all, applying the same business model infrastructure uniformly across agent platforms. The specification of agent acts and procedures is used to guide the implementation of the Agent Model, being able to interpret models on the fly, and interact with Agent Models implemented in other systems which conform to the same specification no matter how they are implemented.

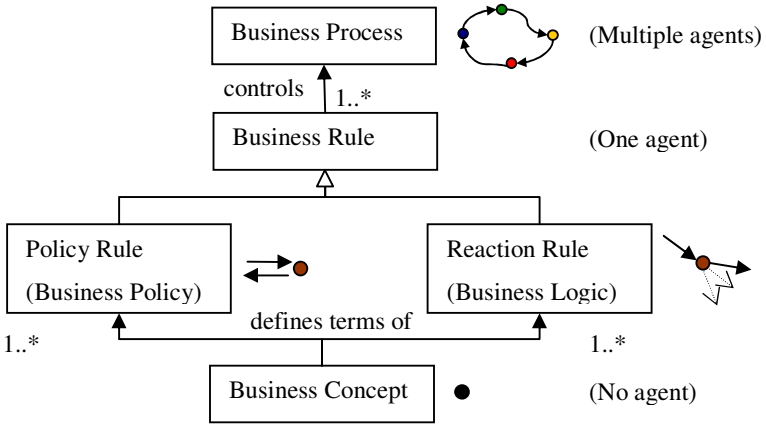

Figure 5.3: AAM business model (simplified version of Figure 1.1)

The business models of AAM are straightforward. Two distinct features should be noted. i) They are directly linked to the requirements. ii) They are intended to be understandable and executable to agents and configurable to business people. These are perceived as a solution towards easier maintenance in response to the weaknesses found in the investigated approaches. To further illustrate, a UML structural model is

given in Figure 5.3, showing that the Business Process is controlled by Business Rules and that there are two subtypes of business rules, namely Policy Rules and Reaction Rules, both of which are defined in terms of Business Concepts. Table 5.1 elaborates the definitions of these elements further.

Table 5.1: AAM components

Component	Description
Agent	• carries out activities required by business processes • structural and behavioural modules reflected in business requirements
Business model	• machine understandable, and so can be translated by agents • built from business requirements
Business process	• principle element of business model • involves other model elements • realises business requirements • can be decomposed into activities specified in business rules and executed by agents
Business rule	• statement, action, and procedure that should be enforced in business environment • business requirements in a form suitable to be assigned to agents • refers to business concepts • distinguished as policy rules and reaction rules
Business concept	• essential and irreplaceable atomic business unit that truly exists in business environment • referred to by business rules • extendable as business requirements change • understandable to agents

5.4 Design aims for adaptivity

The vision for AAM presented above matches with the conclusions drawn by others in the form of design guidelines (e.g. presented in [144]).These are listed below alongside an assessment of the design of AAM's architectural components and how they (could/should) meet the guidelines.

i) "*Design for the long haul*". Making the design flexible in the long run is worthwhile because design for the current requirements will almost certainly prove

to be inadequate some day. The inevitable change of existing elements or accommodation of new elements of business processes, business rules, and business concepts in the future necessitates the provision of an explicit model and a method to modify it well before hand. This design discipline has been considered in designing and will be continued to be considered in building the two hierarchies used by AAM and their interaction for explicit model modification as shown in (i) of Figure 5.4.

ii) "*Use N-tiered architecture*". The separation of three layers: user interface, business logic, and data storage proves to be a flexible way for design in many systems. In N-tiered systems, change to one tier should not require change to others, but the effect is reflected in the overall system. Similarly, the three levels of business models should easily accommodate change to any level without intervene to other levels. For example, alternation or addition of business rules should not affect the core business processes but the effect of change should be revealed automatically when the processes are going on. This design discipline has been considered in designing and will be continued to be considered in building both hierarchies as shown in (ii) of Figure 5.4.

iii) "*Employ and maintain individual data objects*". Logical data entities must be identified and maintained as individual objects. Business rules must be enforced through the objects, ensuring those rules are applied consistently. This discipline, once followed, provides a means that minimises the impact of upcoming entity types and rules about them on the system. In the bottom level of the proposed business models, all types of entities in appearance of the system should be each registered as a different business concept. The concepts and their relationships should be maintained individually. In the middle level, business rules are defined using such established terms. The links of business concepts in business rules should enforce business needs. Consistency should be guaranteed because business rules always point to up-to-date business concepts, both of which are modelled and maintained, explicitly and separately. This design discipline will be considered in building the bottom layer of the knowledge model hierarchy and its interface with the middle layer as shown in (iii) of Figure 5.4.

iv) "*Manage artificial limits (established organisational rules) and regulate them in a business experts controllable way*". The interpretation of artificial limits, or business rules, should be changed as the business changes without effect on information structures or processes. Rules express facts about concepts and

applicable across processes. The potential flexibility in a system is the characteristic of allowing the resynchronisation of the system under development with the real world system at low cost, preferably by business experts. Since business rules are intended to assert business structure or to control or influence business behaviour, a system becomes flexible when business people are given the opportunity to control assertions both on structures and processes. When new requirements arrive, the exemplar reaction of a business rule approach includes adjustment using new types of structures, or new actions in new branches of conditional logic, in processes. An important feature of a flexible system is its use of a content-free programme, which knows where to retrieve information about rules and how to enforce rules, rules being composed of configuration of not only valid values but also combinational logic statements. In AAM the externalised business rules that are controlled by business experts and serve the knowledgebase of agent behaviour should exactly put this discipline into practice, which will be considered in building the middle layer of the knowledge model hierarchy and its interface with the upper and lower layers as shown in (iv) of Figure 5.4.

v) "*Code general-purpose routines. Develop general-purpose business process templates whenever possible, supported by general-purpose processing tool*". This discipline, once followed, provides a means that minimises the impact of upcoming business processes or alternation to existing business processes on the system. If new or changed business processes can be easily defined and processed within the running system, the maintenance overhead of system is significantly reduced. At the top level of the business models, business processes should be specified in whatever way the business analysts need, connecting agents that collaborate towards the goals of the processes. There should not be any constraint in the construction of business processes. This is the case in the generally rather than purposefully built supporting tool discussed later. The adjustment of business processes should not be subject to any artificial restriction and any such change automatically demands the change of agent grouping in the new context of business processes. In design of any organisational domain, a network (many-to-many) organisation, for example, is more flexible than a hierarchical organisation (one-to-many) in specifying entity relationships, since the former structure represents a more general structure with less constraint. When structuring agent relationships, the same is true. This design

discipline will be considered in building the top layer of the knowledge model hierarchy and its interface with the middle layer as shown in (v) of Figure 5.4.

vi) "*Take advantage of reusable logic*". Documenting reusable logic and requiring uniform use of them prevents duplication of work by different developers and hence the wasting of effort. A reusable layer of object infrastructure underpinning the agent layer should support the separation of changeable from unchangeable. The building blocks of objects should have reusable functions available to all agents, knowledge of which, when, and how objects are used being captured in maintainable business models. This design discipline will be considered in building the lower class layer of the running component hierarchy and its interface with the upper agent layer as shown in (vi) of Figure 5.4.

vii) "*Use information-free stable identifiers*". Almost any information about the thing identified is subject to change. It is usually essential that the identifiers (primary keys) used in a system not be subject to change. Information-free stable identifiers contribute to stable information structures and so the reduction of maintenance burden. The proposed solution is that information about identified entities is externalised from the identifiers and maintained independently. In particular, the externalisation of agent behaviour in the business models minimises the potential maintenance to agent programme code otherwise necessary. Instead, the maintenance of business models could bring agent the current business needs directly. The same set of agents could be used in a business process as usual even when requirements on that process are changed. The knowledge of agents, however, could be changed independently, as a result of the maintenance of the relevant business models. This design discipline will be considered in building the upper agent layer of the running component hierarchy and its interface with the lower class layer as shown in (vii) of Figure 5.4.

viii) "*Recognise the unpredictability of required flexibility*". It is simply impossible to imagine and predict all that the system is required to achieve in the future and embed full flexibility into the current design. Therefore, existing processes, rules, concepts must be insufficient. New ones have to be later added via easy to use supporting tool that enables business experts to control the up-to-date system behaviour by manipulating (meta-) data. This design discipline will be considered in building the business tool for business knowledge model configuration as shown in (viii) of Figure 5.4.

With its architectural components designed as meeting all eight design disciplines shown in Figure 5.4, the AAM is designed to be adaptive, in many aspects. The next chapter will discuss the details of the approach of AAM.

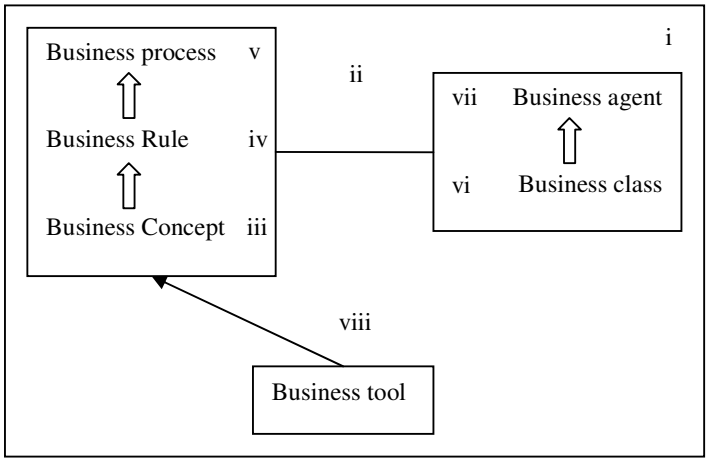

Figure 5.4: A roadmap of achievable adaptivity mapping to design guidelines

Chapter 6: The Adaptive Agent Model

6.1 Introduction

So far, the following have been discussed: the motivation for introducing a new approach for adaptivity (Chapter 1); existing OO approaches towards adaptivity (Chapter 2); a review of the literature that contributes to the AAM approach - specifically agent-oriented software engineering and business knowledge modelling (Chapter 3); the problems to be addressed by the proposed approach (Chapter 4); and the overview and aims of the approach including the hierarchical structures of its model composition and their design disciplines (Chapter 5). The discussion will now centre on how the AAM approach answers the need for adaptivity, and addresses the problems encountered by the existing approaches, with respect to its Requirements and Design Modelling. The hierarchical knowledge models of the AAM will be detailed. To complete the discussion with respect to the full software development process, the implementation and tool support will be described in the next chapter. Major published work covering the overall AAM development process can be found in [1] and [2].

6.2 Requirements meta-model

Figure 6.1 shows a meta-model of the AAM approach, starting from Agent-oriented Requirements Modelling. In the 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. Using this terminology, it can be specified which functional requirements should be organised together, and how their functions interact within business processes.

Individual agents collaborate by playing business roles, which are, in turn, constrained by business rules, which represent the fundamental functional requirements. Agents are the medium that incorporate functional requirements into the business process models, in the form of business rules, connecting input and output ports of agents. Business processes provide a framework that agents and rules can be put together to make the requirements inter-related and understandable from a

high level. Thus, AAM models are agent-oriented, goal-guided and rule-based business process models.

A business rule in such models is developed as much more complex than a traditional simple constraint. It can have a nested structure, where other rules or functions are invoked by this rule to accomplish a specific task, when it is executed by an agent to fulfil a role.

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. This thereby constitutes a hierarchy. In the hierarchy, one piece of functional requirements becomes either a class method or an agent rule. The class-function structure is subordinate to the agent-rule structure. Because the business functions from a wide range of available business classes can be selected by demand according to changing needs to bring dynamic effects in agent behaviour, such separation and distinction of the two major types of requirements representation elements allow the requirements model to be adaptive. The following sections will show the approach based on this meta-model in requirements modelling. A related work published on the specific topic of requirements engineering of the AAM can be found in: [3].

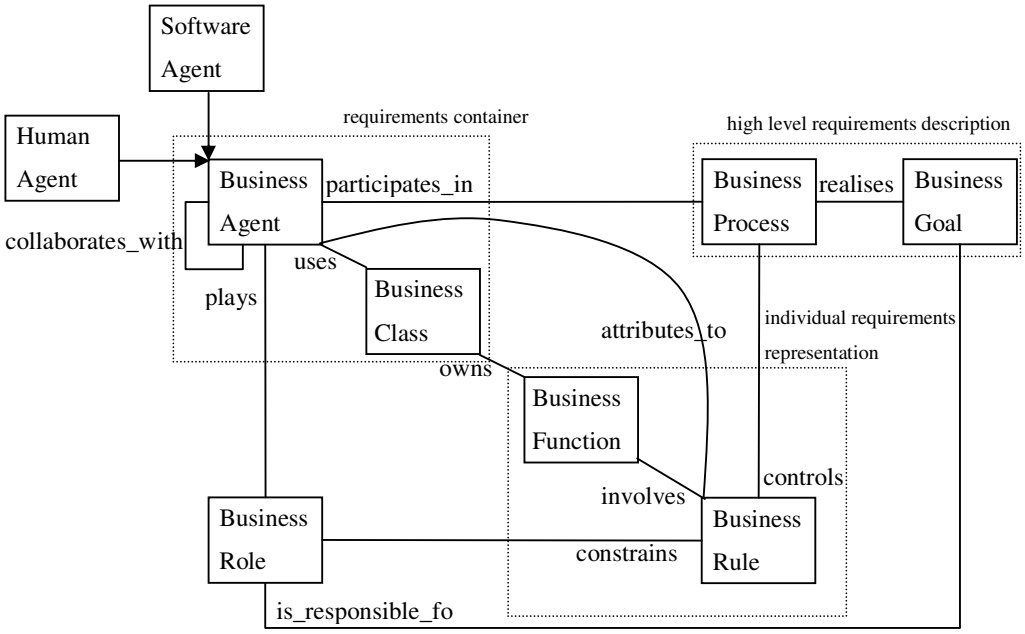

Figure 6.1: Meta-model for requirements modelling

6.3 Requirements Model – goal decomposition

Goal-oriented requirements engineering is concerned with the identification of goals, the objectives to be achieved by the system under consideration; the refinement of identified goals so that they can turn into specifications of operational services and constrains; and the assignment of refined service responsibilities to agents such as humans, devices and software [190]. Most likely, the major goals of the software system under development are explicitly stated in the user requirements document, claiming what its supporting business is for. One way that goals may be obtained from the documentation is by searching intentional keywords such as "objective", "purpose", "intent", "in order to", and so on [190] [191]. A goal-decomposition technique is employed, similar to [26] and [31], 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 realise the goal. For example, the assertion that a goal is realisable by an agent, as in [189], 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 the proposed 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 using a goal decomposition technique down to the lowest level goals, which map to individual functional requirements. This is necessary because of the need to model the business processes in terms of all participating functions in order to later implement those processes. Business goals documented in the user requirements may have been used as the basis to construct multiple functional requirements units. However, the link from individual functional requirements may not have been maintained in the requirements document. Figure 6.2 demonstrates the decomposition process in a generic case.

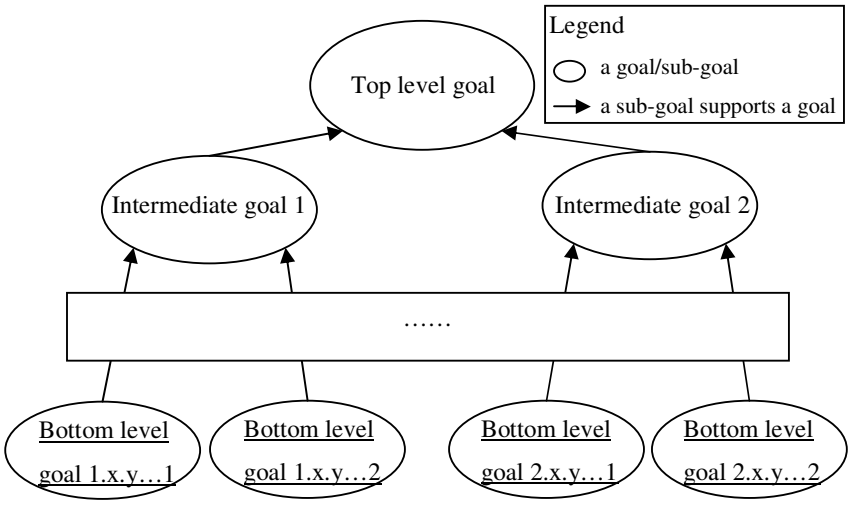

Figure 6.2: Goal decomposition graph

The top level business goal, in the first place, can be iteratively 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, the process terminates and all the leaf nodes, underlined in the figure, are presented as operational functional requirements. This provides a means for incremental elicitation of requirements.

A business process realises a business goal. Therefore, 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 a tiny 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.

The resulting goal-decomposition graphs can help the deduction of intermediate goals, provide a way to organise functional requirements, and check the

completeness and validity of the original user requirements. For example, 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.

6.4 Requirements Model – goal mapping and conceptual modelling

Typically agents represent actors or information systems, identified in the knowledge domain, targeted at accomplishing business goals by participating in business processes. The proposed agent-oriented development approach aims for an integrated development process, starting from requirements modelling, where agents are used to organise the requirements. Following the goal decomposition process described in the previous section, identified goals can then be assigned to agents who participate in corresponding business processes. Agents must be identified before the business process modelling, which will be discussed later on.

Requirements are usually structured by domains, in each of which a collection of requirements defined with the same nature is grouped. Naturally, when one business domain is 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 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. Some agent-oriented methodologies let designers choose to aggregate related roles into a single agent for convenience [40]. In contrast, it is proposed here that this should be done at the specification level, where domain division is the most appropriate criteria that decides the nature of agents and their responsibilities. Domain requirements are assigned to agents in the form of rules or class functions. The case study chapter explains this with illustrative details.

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. 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. Principle modelling elements in this scheme are given below.

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

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

Class: 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.

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.

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

The model structures Agent, Rule and Class in a hierarchy: (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 behaviours; 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 in the previous Chapter. Here, rules represent both Reaction Rules and Policy Rules which compose Business Processes. Classes represent the implementation of Business Concepts. The two viewpoints complement each other and form the blueprint of AAM systems.

The AAM development approach makes use of two principle design models built using its modelling elements. Firstly, a Structural Model, with UML Class Diagram as its counterpart in the OO world, is developed for structural relationship modelling. Secondly, a Behavioural Model, with UML Sequence Diagram as its counterpart, is proposed for behavioural interaction modelling. Both models are developed based on a Reaction Rule Model, which captures the requirements and re-constructs them, distinguishing agents from classes and instructing agents to use available classes at various opportunities. The design models constitute the system blueprint and form the Business Process Rules, the constructs of the highest knowledge model shown in Figure 5.1, mapping to business goals elicited in the previous section. The required design models will be discussed later in the chapter. Prior to that, the supporting Conceptual Model and Fact Model are introduced. They are the foundation of the knowledge hierarchy and establish the vocabularies of the Policy Rules and the Reaction Rules, the Policy Rules being applicable across the Reaction Rules globally in the whole system and the design models being built upon the Reaction Rules.

6.5 Conceptual Model & Fact Model

The use of metadata has been put forward as a means to achieve adaptivity [18]. However, this means that there is a need for management of this information separately from the applications that gave rise to much of that information in the first place [59]. For the purpose of easy maintenance, and to ease interoperability, metadata should be treated as assets, externalised from the applications that use them. To achieve this, conceptual schemas must be made explicit. This is a central idea of the Semantic Web research towards interoperability, where a web of data in

common formats is related to real world objects and interchangeable among applications that use the data. Structural metadata gives form to data and describes relationships among data elements. It is a way for machines to understand varying levels of scope within a data stream, document, or message [59].

The use of concepts and having them related in structures makes information described by them interpretable and understandable by computers, or agents running on computers (as in the AAM). The Conceptual Model of AAM, therefore, serves two major goals. One is for easy extension of system concepts, so that agents in the same system can speak extendable vocabularies. The other is for interoperability, so that agents in different systems can communicate if a mapping between their vocabularies is given. This second goal is further supported by the communication language detailed in Section 7.5.

It is a common practice in the object-oriented community to begin the modelling process with a conceptual model of the domain space relevant for the application being designed [59]. 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. These are collated into a glossary or data dictionary [64]. This technique is plausibly applicable for the identification of business concepts and the building of a taxonomy, the starting point of the AAM. Since business concepts are valued by business people and exist in business models for business reasons, just like objects exist in object-oriented systems, the extraction and explicit modelling of them is important.

As an example, business concepts identified in an e-commerce domain may include “customer”, “name”, “organisation”, “type”, “discount”, and so on. A “customer” has properties indicating their name, organisation they belong to, a classification type for shopping, and discount they can enjoy during shopping, these themselves being further business concepts and properties of “customer”. Some of the concepts must have values in a certain range when instantiated in the running system (then they become facts), e.g. a customer can have a type of either “ordinary” or “premium”. Some others can be more flexible but have to conform to certain constraints to be valid. For example, a customer can belong to an organisation named by a string and their discount can be any floating point number between 0 and 1.

customer {name [string], organisation [string], type [ordinary, premium], discount [float, 0-1]}

Together, all business concepts, their constraints, and concept-property relationships form the ontology of the AAM business models, the Conceptual Model. These must be registered before being referred to by business rules and business processes. Additional relationships between business concepts may be enforced as required by business rules for business needs. One business rule in the e-commerce domain may specify, by implication, a relationship between two of the identified business concepts, e.g. “customer” and “discount”. For example, by default each customer can enjoy a certain discount, but customers from certain organisations may enjoy a different discount. 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 on them using the extended concepts.

At runtime, concrete facts are established as instantiations of abstract concepts documented as shown above by “customer”. For example, when a customer called John starts shopping, one fact may be established states that John, being an ordinary customer, belonging to Queen's University Belfast, according to his registered profile. Properties of this business object are thereafter populated with values.

customer {John, ordinary, Queen's University Belfast …}

Facts like this accumulate and are stored in the Fact Model. One dedicated agent, the Fact Manager Agent (FMA), is responsible for the management of all facts at that moment. It interacts with a Policy Rule Manager Agent (PRMA, detailed in Section 6.6.1) to add new deduced facts after the application of Policy Rules, and acknowledge all agents the facts available to the system. For example, one policy may say that a customer belonging to any UK university can enjoy a higher discount of 20% instead of the default discount of 10%, so that more is now known about the original fact for the same customer. Accordingly, the Fact Model gets updated.

customer {John, ordinary, Queen's University Belfast, 20%}

Facts can be established when event messages bring information to agents and used by agents to reason about their reactions using Reaction Rules (see Section 6.6.2). FMA adds new facts or demolishes invalid facts dynamically at run-time. Business models thus can change dynamically and this is reflected in agents' knowledge.

The Conceptual & Fact Models methodology 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 [1] [4], 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. For example, the customer of John may be mapped to a business object instantiated from class "Customer". By using existing facts, or properties of the business object, after some computation defined by the class methods, additional special arrangements about this customer may be made. However, several methods may be defined in the same class for different purposes, or even several versions of the class are available. 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.

Because the business concepts that comprise the business rules are separated from the classes which are associated with them, it is only at the time when they are used, that the specific matched class methods are found and invoked. Therefore classes to be invoked at runtime are exchangeable and new behaviour can be achieved by the replacement of classes or their methods. Adaptivity is hence enabled. Furthermore, although systems built as such should all share the same message exchange schema (detailed in Section 7.5) and metadata schema in XML for interoperability, internal object-oriented representation may vary and get mapped differently for individual platform processing.

Consequently, purely from the business model aspect, defined business rules and business concepts can be used to deduce business information to support forthcoming behaviour as and when business events occur and business facts are established. Several dedicated agents are responsible for accumulating and imparting knowledge

to other ordinary agents through collaborations in this respect. Also, business concepts that are referred to by business rules correspond to business classes. In the running system, agents are supported by the methods inside the classes. The invocation of these contributes further knowledge, and additional information becomes available to support forthcoming behaviour. Facts are accumulated by both means and maintained in the Fact Model. Later, Section 8.9 uses a concrete scenario to illustrate this.

6.6 Business rules

Ontological vocabularies represented by business concepts, must be constrained in value, logical relationships, and so on, reflecting the nature of the business and the business needs in knowledge comprehension, sharing and use. These are specified in business rules, connecting the ontology building blocks, and describing semantic relationship among them. Such expressions are stated in machine understandable form and allow human comprehension and edition.

Rules can be classified differently, some classifications being found in [46], [28] and [59], based on different needs. Two types of rules are distinguished here: Reaction Rules (RRs) are rules about processes that are in response to some events; Policy Rules (PRs) are rules concerning business policies. Underpinned by the Conceptual Model and the Fact Model, both contain metadata knowledge and they are applied locally by individual agents at the execution points of business processes, which globally involve many agents. As such, business rules exist independently of the processes and concepts. They provide a connection layer that supports the execution of processes using the defined concepts.

A RR is a business rule that constrains system behaviour based on conditions. A PR is a business rule that constrains system component relationship by connecting some business objects/attributes/associations with others. The two types of business rules are similar in nature, role and presentation, and must be obeyed by agents. However, the two types of rules operate within different scopes. A RR describes a process that a particular agent uses business objects for some purposes during its reaction with other agents. A PR describes a relationship between business objects themselves, decided by business people. It may be activated by any agent when a policy must be enforced. Nevertheless, both kinds of business rules are eventually used and applied by agents to meet the system specification.

6.6.1 Policy Rule Model

The justification for using business rules is that they provide a means for implementing business policy [46]. Business policies, naturally, change over time and thus externalisation of them as executable rules is desirable. The use of business policies structured in an IF-THEN format is presented in [5] and [7] as global rules that all agents in the system should obey. It could be conceived that a RR captures a constraint of pre-condition and post-condition (see Section 6.6.2), while a Policy Rule (PR) captures a constraint of invariant type. PR assertions on the logical relationships between entities must always be true to reflect the required policies. These may be carried out automatically in certain settings such as a reaction process taken by an agent, if relevant policies must apply. The compositional *entities* of a PR are: business objects, attributes, associations, operations, and the compositional *operators* of a PR are: IF, THEN, AND, OR, and so on. The IF condition of a PR returns a Boolean expression over relations between entities/values. The THEN action of a PR is an assignment of entity values/concrete values/acts to other entities/actors. If operations, acts, and actors are allowed in a PR, it is localised in its application scope, and could contribute or become a RR (see category 3 and Section 6.6.2). Typical PRs are categorised as follows, illustrated here with an e-commerce scenario.

i) Policies defined for classification of business objects.

The same types of business objects are sometimes required to be classified and treated differently because of their different nature. For example, customers in an e-commerce domain may be distinguished in accordance to their types indicating the associated different potential business opportunity. Rule1 in Figure 6.3 is an example of this, stating that any customer representing an organisation is a premium customer, differentiated from ordinary customers so that different strategies possibly need to be applied to them.

Rule1.
If a customer is an organisation representative
Then he is a “premium” customer

Figure 6.3: Definition of a Policy Rule on business object classification

In the knowledge of Rule1, the Policy Rule Manager Agent (PRMA) could update the Fact Model through its interaction with the Fact Manager Agent (FMA) at runtime. Suppose a “customer” starts to interact with the system, a fact is then

established in the Fact Model with its "type" initially unknown. It is through PRMA's evaluation of this particular property using the PR currently set by the business people that the Fact Model is updated, so that the property "type" of the "customer" is set as "premium". Relevant rules are selected to work in sequences when business events are detected in business processes. Available business objects/attributes can be selected to compose PRs, as long as they are registered. Concrete values, as well as object attributes, can be used as condition or action parameters. For example, "organisation representative" and other specific categories in use must be allowed values of the known and registered properties of the "customer" concept. New PRs can be defined on new concepts immediately after they are defined. Other changes in the Conceptual Model also affect the rules which refer to the changed objects/attributes immediately. For example, invalidated concepts would make relevant rules become invalid, thus guaranteeing a consistent system. The application of a PR through a dedicated PRMA responsible for the deployment of policies to other agents is also found in previous work [5] [7].

ii) Policies defined on the relationship of (classified) business objects and their attributes, triggering the formation of a "PR chain".

The term "*premium*" used by Rule2 (Figure 6.4) as its condition is defined in Rule1 as its consequent action. In other situations, if the result (post-condition) of one PR is the prerequisite (pre-condition) of another PR at the time of its execution, then the execution of the former triggers the execution of the latter. A PR chain is thus formed, Rule2 triggered by Rule1 in this case. Such relationships as concept reference, logical inference, and collaborations among PR are likely to contribute to the establishment of a PR chain. The result of such a PR chain may be used by RRs to make decisions or choose reactions. Other PRs may contain computational formulas that can be used to calculate the concrete values of certain elements at runtime.

Rule2.
If a customer is a premium customer,
Then he enjoys a discount of 5%

Figure 6.4: Definition of a Policy Rule on a business object attribute

iii) Policies defined on the relationships of (classified) business objects/attributes, triggering the system behaviour.

PRs can be defined with respect to the PR performers. Different effects can be expected depending on the nature of the classified business objects and other conditions. Rule3 (Figure 6.5) is such an example and actually used in combination with Rule1 and contribute to a RR. When a customer logs in, his identification is made clear and whether or not he is a premium customer is established as facts, then other policies (showing VIP products & assigning a discount of 5%) may be triggered to accumulate facts or eventually prompt an action, as a result of the combinational application of the PRs. A RR describes taking different actions in different conditions, and a PR can contribute to a RR in this sense.

Rule3.

If a customer is a premium customer

Then the customer service agent residing on the product browser page shows products under the VIP brand

Figure 6.5: Definition of a PR contributing towards system behaviour (RR)

A PR takes the form of an IF-THEN statement. If the IF clause evaluates to be TRUE, then its condition is satisfied, and the THEN clause statement becomes a fact or its described action is activated. Therefore, new facts are increasingly known and other PRs may come into play, leading to additional facts being established. Iteratively, new knowledge is derived from existing knowledge as these "PR chains" are formed. The process proceeds until no more PR conditions are satisfied.

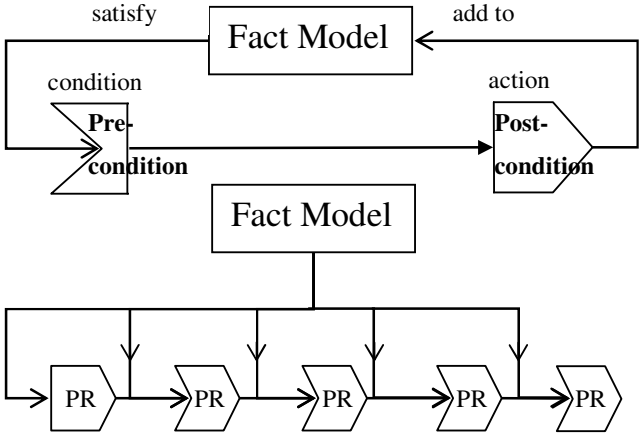

Figure 6.6: Reaction Rules and Policy Rules are unified

Figure 6.6 illustrates that the two types of business rules are actually unified. This simplifies the later implementation. A PR is divided into condition and action, which resemble the pre-condition and the post-condition of a RR. At the business model level, both RRs and PRs can be described as statements about something that must be true or that must be done if some conditions are satisfied. At the information system

level, both RRs and PRs can be represented by {condition, action} pairs, encoded/stored in the same manner (using the same XML schema), and executed by agents in the same manner as well. However, the deployment and effective scope of two types of business rules differs, PR having global effect in the entire system and RR having a local effect in an agent [5] [7]. A PR is a business rule assigned to the PRMA responsible for request from other agents, for the deployment of a pre-defined strategy in the entire system. A RR is a business rule assigned to a particular agent responsible for the deployment of a pre-defined procedure. A BPR is a set of PR and RR involving multiple agents responsible for a common goal.

Using the AAM ensures the information system is tightly aligned with the underlying business models. The approach proposes two kinds of rules representing business logics of policies and reactions for business requirements. They are built from the requirements elicitation, analysis, decomposition, and reconstruction, providing models with traceability from their origin, so increasing maintainability. Such systems can be implemented independently from any particular technology. The AAM supports a process from capturing requirements in models to building running agent systems that interpret the models as agent behaviour.

6.6.2 Reaction Rule Model

The overall architecture of a software system is important to the efficiency and effectiveness of maintaining it [44]. AAM has an event-driven architecture, its business rules being reactive to events. 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 [43]. In an approach proposed to improve the flexibility of object-oriented system architecture, it is suggested that events being raised to the same level of importance as objects [199]. The result of such event-driven systems is that objects interact through events and so are loosely coupled, leading to a more flexible architecture.

In many agent-oriented systems, agents monitor the occurrence of interesting events and respond to them [42]. The agent response might generate new events to be sent to other agents. Most contemporary systems view event information as a string without any semantic meaning [46]. In contrast, the AAM considers events as providing the context for agents to react and through sequences of successive events,

business processes can be formed and executed logically and meaningfully. Dynamic handling of events is a means to bring adaptivity. Since not all events are predictable, new types of external events to a system may change the way it works and entail evolution in its agent reactivity and business process interactivity. On the whole, a business process may be viewed as a community of service-providing agents, each of which represents a distinct role and is capable of providing one or more services [42] [40]. A service is simply a single, coherent block of activity that an agent will engage in, and may have properties of input, output, pre-conditions and post-conditions [40]. Agreements are bound between agents for their interactions. Business rules are used in the AAM to capture such agreements, and decide and constrain what and how agents provide services. They specify sub-processes that agents should perform in a reactive and proactive manner, and realise in the specification level the expected cross-domain interactions and policies enforced, wherever applicable. Business rules serve as contracts between interactive agents, and provide contexts for individual agents to behave. Various business rules can be defined for a single agent in playing various roles. They are triggered by events and result in actions. Events correspond to pre-conditions, without which agents won't execute the business rules. Actions correspond to post-conditions, which must be satisfied after the execution of the business rules. Events and actions have messages in which business objects are encoded. Agents process event messages and produce action messages. Agents communicate with each other through passing messages. It is through events and actions that agent interactions become possible. The details about the communication infrastructure of AAM are found in Section 7.5. The principle published work on Reaction Rule and the associated two Design Models built upon it for a united design framework can be found in [4] and [6].

Event Condition Action (ECA) rules have a structure of "on *event* if *condition* do *action*". They have been used for automatically triggering actions in active databases [51] [50] and Resource Description Framework (RDF) repositories [49] on SQL events such as "INSERT" or "DELETE" operations. ECA rules allow the definition of systems' reactive functionality and claim to enhance modularity and maintainability [49], for example in the e-commerce domain [194]. However, the actions they support in response to events are usually limited to database operations. More importantly, they are weak in terms of extensibility and cannot capture the full decision making processes, where multiple actions should be considered in multiple

conditions on detection of the same event. Nevertheless, the use of ECA rules has offered users the flexibility to re-configure workflow systems according to evolving needs [56]. From this it can be inferred that if rules are extended in structure, so that each individual rule can cope with many conditions, this increased flexibility would make the rules more powerful.

To this end, RRs are allowed to follow decision making procedures and allow agents to reason about the proper actions to take in various conditions. Different judgements may be made at different times, resulting in different actions, for different situations. Suppose Figure 6.7 represents a schematic decision making tree, as shown. Each RR's decision making part can be decomposed into multiple {condition, action} couplets. An action can follow each condition. Those actions that correspond to the conditions on the tree branch will be selected while decision making is performed. For example, a branch of condition2 → condition2.2 may be selected initially. If some additional conditions need to be considered following requirements extension, the extended couplets could be, for example, {condition2.2.1, action2.2.1}, {condition2.2.2, action2.2.2}, and so on. The selected tree branch would then be: condition2 → condition2.2 → condition2.2.1 → condition2.2.1.1. The tree structure of a RR is evaluated by an agent and a branch is selected to be performed if that RR is appropriate for the agent in the context of a business process. Condition sets may be evaluated with the assistance of the FMA using the Fact Model.

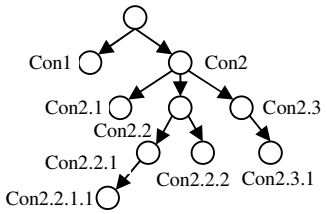

Figure 6.7: Schematic decision making tree

Generally, driven by events, RRs make business decisions using {condition, action} pairs before actions are performed, to fulfil the function of RRs, and in turn produce events to be sent to other agents. Informative event messages have business objects encoded in them, conforming pre-defined structures but with runtime values resulted from computation. The specific business objects are decoded and used by the recipient to make a corresponding decision and respond accordingly to the request the message senders make at that particular moment. Depending on the

specific event contents, different decisions are made by individual agents in different situations even when the same RR is used, playing different roles and yielding different results. While different actions are chosen by agents after the decision making, different collaborative agents are chosen, and business processes are also formed differently. Consequently, adaptive agent reactivity and business process interactivity become possible. This is due to the fact that both agent-agent interfaces and agents' chronological performance in processes are subject to configurable rules.

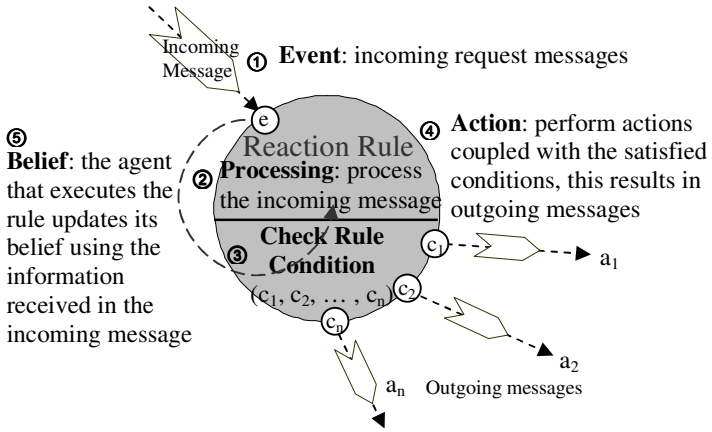

Figure 6.8: Reaction Rule Model

As RRs guide agents to interact with other agents in a similar manner, they share a common structure of: {event, processing, {condition, action}$_n$}. The graphical representation of this is given in Figure 6.8. Essentially a RR determines, on receipt of an event message(s), after the processing of the message, if certain conditions are satisfied, the actions that agent should perform. It is possible to specify in a RR that a combination of several independent event messages or any one of them triggers its function. Event types are specified in the RRs. At runtime, when an actual event is received, this instantiation is checked against of its type and only processed if its type matches with that of the RR.

Figure 6.8 shows that an agent processes a rule using the following steps.

1. Check event – find out if the rule is applicable to deal with the perceived event.

2. Do processing – decode the incoming message, construct business objects to be used in later phases.

3. Check condition – find out if the (condition ci) is satisfied.

4. Take an action – if ci is satisfied, then do the corresponding (action ai) that is related with (condition i) as defined by the rule. Then send a result message to

another agent (possibly the triggering one). If ci is not satisfied, then go back to Step 3 and check the condition ci+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 and the configurability of them brings adaptivity. Mutable requirements on components collaboration can be externalised in rules, and the agent knowledge is made updatable for the collaboration partners, events processing, and response messages production. Different actions can be set in rules as reactions to different conditions, in an order of user preferences/priorities. Details on rule configuration are discussed later on.

6.7 Design Model - structural modelling

An AAM Structural Model diagram has the Class Diagram, the backbone of UML, as its counterpart in the OO models. Agents are regarded as superior to classes in this model, just like classes are regarded as superior to attributes in OO models. Referring to Figure 6.9, 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.

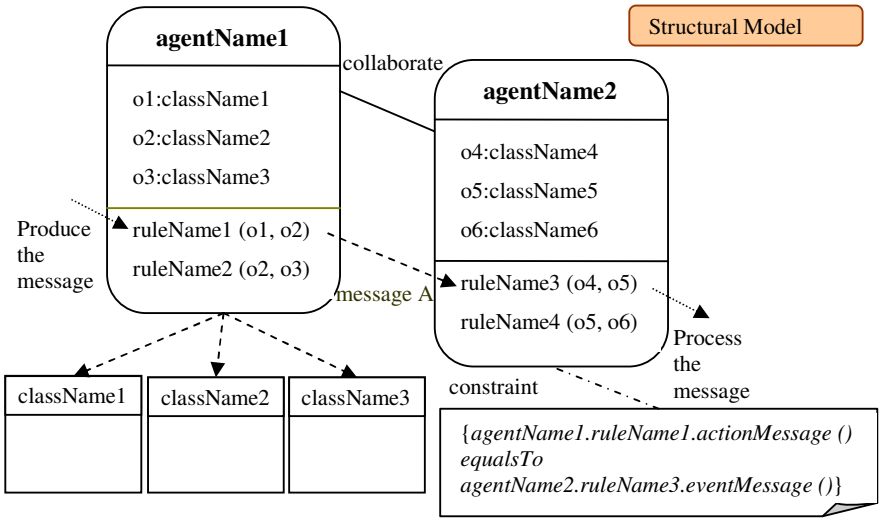

Figure 6.9: Structural Model

In such a model, as Figure 6.9 shows, two agents are connected by a "collaborate" line, if they interact with each other, by passing messages. Standard methods invocation between classes is replaced by rule collaboration between agents, where one rule produces a message and the other processes it. In the above scenario, ruleName1 tells agentName1 that objects instantiated from className1 and className2 will be used to produce message A, sending to agentName2. As the definition of ruleName3 indicates, it is to process message A. agentName2 selects it to collaborate with agentName1, using objects instantiated from className4 and className5. What and how components are to be invoked can be configured with rules and such configuration information is obtained by agents at run-time to ensure the deployment of the most up-to-date requirements. Eventually, the requirements for component interactions are replaced by agent interactions and such knowledge is represented in business rules. Another layer of abstraction is hence achieved.

6.8 Design Model - behavioural modelling

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

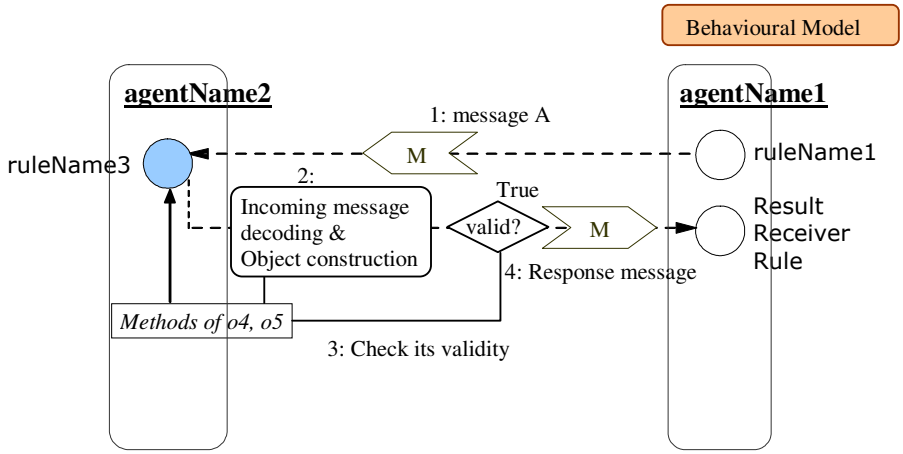

Figure 6.10: Behavioural Model

Structural Model diagrams represent system structure, and the static relationships of the compositional elements of the system. In the OO world, UML Class Diagrams are complemented by Sequence Diagrams for behavioural modelling, and likewise, ARC Behavioural Model is designed to capture the behavioural scenarios. Inspired by the Agent-Object-Relationship model [23], associated agents/rules/classes are grouped and show their behaviours and interaction processes for achieving certain goals. Rules are event-driven processing units for agents, working by invoking classes. A Behavioural Model diagram corresponding to Figure 6.9, assuming that the response message is sent back to the initial agent, is shown in Figure 6.10. It visualises the actual system function in a sequence of actions, with only a satisfied {condition, action} couplet shown in this scenario.

In traditional OO systems, objects are 'aware' of which other objects they will pass messages to, but are unaware of which objects will pass messages to them. Conversely, the applied mechanism achieves full architecture independence, or the two-way encapsulation, in that the details of where entities (agents) will send messages are also hidden [62]. This achievement results from managing agent collaboration with rules, which are not hard-coded but configurable. For each agent, not only its collaboration partner, but also the way it collaborates with other agents can be amended, according to the changing requirements.

6.9 Reaction Rule Language – Rationale

The RR is a core component of AAM, making use of PRs and forming BPRs. Its design and implementation contributes to the novelty of AAM. In particular, it brings

advantages to the integral modelling language over UML/AUML, towards a model driven architecture. Considering its importance, the rationale of this design artefact is demonstrated in this section.

6.9.1 Rule dependency: hierarchical encapsulation

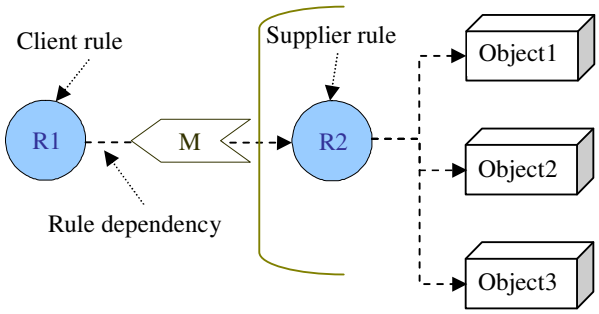

Figure 6.11: Rule dependency relationship

There is a direct dependency between two rules where there is a collaboration relationship between the agents they belong to, because the event of one rule is dependent on the action of the other. Referring to Figure 6.11, assume R1 has sent a request message to R2, asking for a service. Assume also that R2 has done some computation and now is sending a reply message back to R1, as shown in Figure 6.11. Rule R2, which provides the service is termed the *supplier rule*, while rule R1, which receives the service, is termed the *client rule*. This client rule will become the supplier rule if/when it sends a message to another rule and provides its own service. In the above case, there is no direct dependency from the client rule R1 to the three business data objects that supplier rule R2 has used to fulfil the service. If these business objects change, corresponding change may be necessary for R2 when it invokes the objects, for example while it checks rule condition or constructs the action message, etc. In most cases, the change will be in the computational logic, internal to the objects, not the produced message format, so that the change stops at R2 and does not affect R1. If the business objects maintain the same interfaces while their implementation is changed, then the change stops at the object level and there is no effect on either of the rules. At the lower level, by using a good OO design methodology, this can be achieved without much difficulty. In any case, there is no ripple effect on R1, the client.

Consequently, the relationships between agents resemble the relationships between objects in the OO world, while internal business classes of agents are

encapsulated and known only by their owner agents, analogous with private methods of objects, which are encapsulated and only known internally by their owner objects [39]. Changing private methods of an object has no direct effect on its associated objects, and likewise, changing business classes of an agent has no direct effect on its collaborated agents. These patterns of design model in agent/rule and class/method hierarchy promise better control of dependencies. Because agent is another level of abstraction over class, an advanced "hierarchical encapsulation" is achieved.

6.9.2 Rule-Rule relationship: a message serves as a rule connector

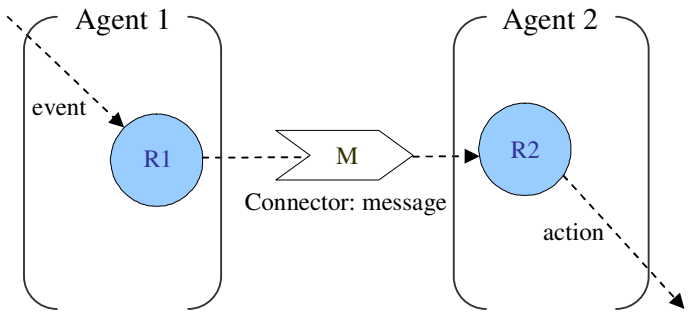

Figure 6.12: Rule-Rule relationship

The communication between two agents, through two designated rules, is by message passing. This message can be viewed as a connector [187]. It connects two rules in the sense that the message is produced as an action of one rule and processed as an event of the other. In Figure 6.12, such a connector between Agent1.R1 and Agent2.R2 is shown.

This implies that the action message of R1 is equal to the event message of R2. Despite the straightforward and obvious relationship, such a constraint is not trivial. Implicit requirements are captured in the rules and the consistency of the original specification can be checked. Assume that two domains are associated at the two functional requirements, Function1 and Function2. Each of them belongs to one of the domains. If Function1 causes Function2 to be affected, then the output and effect of the first function is the input and direct cause of the second function. In other words, there is a relationship between Agent1.R1.action and Agent2.R2.event, provided Function1 maps to R1 and Function2 maps to R2. In this scenario, a message must pass from Agent1 according to its Rule1, to Agent2. Likewise, Agent2 expects to receive a message from Agent1, according to its Rule2, the two messages being identical in fact. Unless the corresponding domain functions match, there

would be conflicts in the original requirements document. The risk of failing to change a requirement to reflect a change in a related cross-domain requirement is thus avoided. Since any natural language based specification can rarely be guaranteed free of inconsistency, a supporting tool to be described later on can be used to check the formatted rules for consistency.

6.9.3 Agent-Agent relationship: a rule serves as an agent interface

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

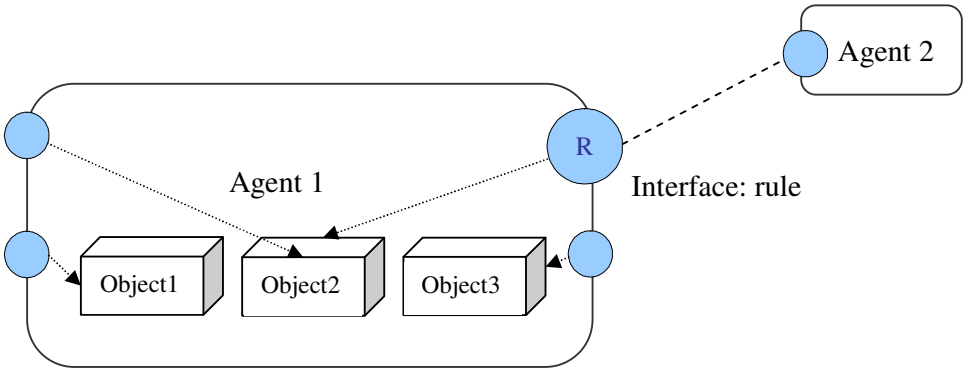

Figure 6.13: Agent-Agent relationship

Figure 6.13 illustrates the interaction between rules (represented as circles), and agents (represented as rounded rectangles), and also their use of objects. Agent2 collaborates with Agent1 at rule R. Rule R of Agent1 makes uses of Object2 for providing a service for or requesting a service from Agent2. Whatever the case, such activities as event message decoding, condition checking and action message encoding are involved while rule R functions. There is a message shared between Agent1 and Agent2, the outcome of which is that this message is decoded as an event message or encoded as an action message, depending on the direction of message passing or their dependency relationship described in a previous section.

Again, a constraint captures implicit requirements and enforces consistency in and with the original specification. A rule is associated with an agent either because its specified event instantiation is generated by that agent as an action to its owner agent, or its specified action instantiation is processed by that agent as an event. A rule is associated with a business class for analysing the event message, checking if its condition is satisfied, or constructing an object for the action message. Thus, a rule in the model has two distinctive interfaces: an external interface between the rule and its associated agent and an internal interface between the rule and its associated class.

6.9.4 Reaction Rule language – why not UML?

The syntax and semantics of a Reaction Rule have been introduced in the previous sections, the specification of which becomes a special language. The rule language plays a similar role to other interface languages like the Object Management Group's Interface Definition Language (IDL) [188]. 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 and mechanism of the server objects. An interface definition specifies operations to be performed, inputs and outputs, allowing clients and servers to encode and decode values for their travel over the network, regardless of their platform, operating system, programming language, and so on [188]. 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.

The disadvantages of UML in traditional software development have been demonstrated in previous sections. An enhanced use of UML 2 [137], however, is found its explicit process modelling, targeted towards direct machine execution. Two major diagrams (with extension) have potential in this respect, in presenting, (i) actions and control flow, (ii) object flow, (iii) organisational structure, and (iv) interactions in processes [150], though only conceptually. (i) Activity diagrams can be used to visually represent coordinated sequencing of *actions* including *control*

flow (with *decision node*, *fork node*, *guard*, etc.). (ii) When the information and data passing for processes must be clarified and explained in terms of types, properties and states, *object flow* can be combined with control flow by extending activity diagrams with class and object diagrams in integrated process models. In this extension, *object nodes* are used to model the occurrence of objects at a particular point in the process. Alternatively, they can appear as *input pins* and *output pins* that are directly attached to an action node. The result of adding these constructs is that, the direction of objects passing through different actions of an activity and the assignment of objects to the input and output parameters of the actions can be modelled. Actions have their output and input pins connected by object flow edges. (iii) In addition to this extension, organisational structures involved in an activity can also be modelled by the class and object diagrams, connecting to the process model. *Activity partitions* and *swim lanes* are used to divide an activity into sections and depict which actions included in an activity will be assigned to and executed by which specific organisational entities that are responsible for the respective actions. An introduction to the UML 2 activity and action models covering the basic of these features can be found in [193]. (iv) In contrast to the activity diagrams, which focus on various dependencies between actions in a process, sequence diagrams provide a complementary view of process in modelling *interactions* among business partners and hide internal actions of the participants. In the diagrams, each participant progresses through a *lifeline*, between which *messages* are passed as indicated by arrows and their sequence along the lifeline represents the order of message exchange.

Adopting a process-aware view of systems, the various diagrams above can be expressed with an aim of process enactment, where business processes can be enacted with the help of computer systems [150]. Specifically, this can be achieved by relating existing software components available in the IT system with the actions of the process model. Two means are suggested. One approach is to feed process models into a central process engine that accesses all software components and uses their services according to process description. A second approach is to use process-driven coordination of existing component services to realise the desired services (an approach taken by service-oriented architecture). The later approach requires specifying components in terms of their provided and required services, coordinating their invocation and control/object flow, and integrating them into the process model.

Either approach boosts explicit business process modelling and direct execution of such models and so guides model reuse. Furthermore, both the activity diagrams with the extension of object flow and organisational structure modelling, and the sequence diagrams with message passing between business partners modelling are useful in exposing the actual business entities seen in authentic business environments. However, their separation means the difficulty of building one single coherent model that captures the business aspects in both models. Such issues must be considered. Different views must be consistent (actions in different diagrams are in the same order) and the combinational use of them for an integral process enactment may bring additional complexity and difficulty. Moreover, semantics of the models are not directly associated with them for immediate machine understanding and processing (object processing in the control flow, message encoding and decoding, and so on.). External means must be sought to relate models with computing services and manual effort is required to achieve this.

The AAM diagrams are especially designed to answer such deficiencies in an agent-oriented context. All business aspects are captured consistently and coherently in unified models. A Structural Design Model captures the organisational structure of agents and classes/objects. A Behavioural Design Model expresses the interaction among agents externally, their decision making internally, the message passing sequence resulting from the decision, and the role that business objects (integral part of the computing services in support of agents) play (decoding/encoding message, being the content of message, facilitating decision making, etc.) in the interaction involved in the process. The comprehensive design models are integrated due to a common RR core component as the foundation of both.

In addition, models have direct links to the computing components of agents, which also have direct business representations. In particular, RRs add semantics to the AAM diagrams since, on finding the appropriate RR, agents know how to proceed immediately because the composite pattern of the rules is designed for agents to function in steps captured by rules. The RR implementation, to be discussed later on, further supports this mechanism since definitions in XML format are attached to the rule elements in AAM design diagrams, providing a straightforward means for rule execution by software agents. At the same time the diagrams are easily editable by humans and can be synchronised with the rule definitions in XML. A Model Driven Agent Architecture (MDAA) linking the model

development to the agent systems executing the models is hence achieved. Details of this will be described in the chapters following this one.

6.10 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, sequentially and conditionally following event message flow, forms business processes, combining inter-related Behavioural Models built upon Reaction Rule Models, and thus forming the blueprint of systems. The overlap of RRs in their event and action components allows them to be interconnected in business processes, as shown in Figure 6.14. The interconnected RRs in a business process collectively constrain the whole business process and form a higher level rule for a system goal, termed a Business Process Rule (BPR). Thus, one RR is about how one 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 a collection of RR. A BPR thus delivers, a goal shared by many agents.

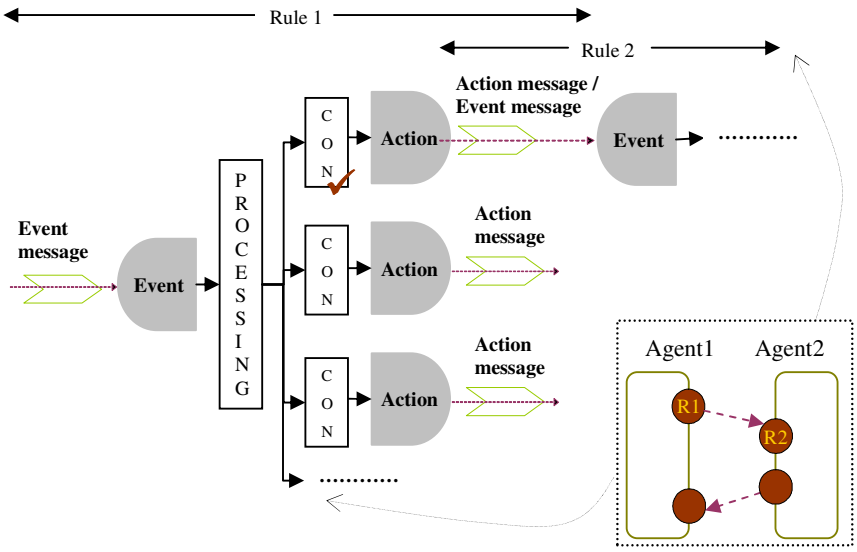

Figure 6.14: Rule1 sends a message to Rule2, rules collected at messages in BPR, reflecting matched event and action

The formation of BPR is by matching action and event pairs of interconnected RRs. Globally to each BPR, Initial Agents (IA) are referred to by those that initiate the BPR and Final Agents (FA) 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 BPRs. Tools support the composition of BPRs to ensure the completeness of business processes and their goals. This means for every input from the IAs, results 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 (see also BPR vs. use case in Section 6.11). 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.

A business process gives rise to a number of instances of message passing among agents. The message schemas are pre-defined by involving RRs and PRs which are used at the time of execution, the concrete content of messages being populated at runtime. The Rule hierarchy of Business Process Rule – Reaction Rule – Policy Rule is therefore 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.

BPRs can be viewed from a software architecture perspective. Firstly, a conceptual architecture has been built where agents in each BPR are identified and their responsibilities are assigned. Secondly, a logical architecture has been built where RRs in each BPR are modelled to serve agent interface specifications and connect these components. Finally, a realisation architecture will be built in the following chapters, where the resulting BPRs are formatted and become executable by agents associated with the BPRs.

In sum, the BPR is the organisational architecture of the agent system to be built. Although BPR composition is described here as the last step in the process of AAM requirements and design modelling, actually it could be the starting point, co-considered with goal-decomposition to obtain an initial blueprint of the system. They have to be elaborated on with details, however, when all the other composite components of business concepts, business policies, and reaction rules have been clarified. The overlap of requirements elicitation and overall architectural design over identifying and assigning requirements responsibilities to components of the architecture has been introduced in the Requirements Elicitation part of Chapter 1. The connection between goal-oriented modelling and architecture design has also

been mentioned elsewhere in [198]. Moreover, the importance of an explicit architecture model for runtime adaptability has been suggested in [203]. In that proposal, an architecture model describing user and system context in a changing mobile computing environment has been used by an adaptation middleware, which applies pluggable and exchangeable components to solve adaptation concerns.

6.11 BPR vs. use case in standard UML

UML use cases can be used to organise actors and the roles they play. It is proposed in [8] to use goals for structuring use cases. Responsibilities are assigned to each actor, and known to other actors. On a request related to one of its responsibilities, an actor forms a corresponding internal goal and tries to achieve it by using its own functionalities and responsibilities of other actors. A use case thus has a goal and achieves it by means of the collaboration of actors.

Although use cases are useful for requirements elicitation from the viewpoints of actors of the system, they are not as effective for eliciting constraints or high-level business requirements from indirect viewpoints of the system or for discovering domain requirements [63]. Use of BPRs shares some common features and advantages with use cases, but avoids some of its disadvantages. Both BPRs and use cases can be structured according to goals during requirements elicitation. They are both useful in capturing system functions, identifying roles of actors, clarifying interactions, and organising message flow. A use case will fire business rules to constrain its process, if they are relevant in that use case context. BPRs act in the same way as an integration unit of RRs. In addition, use cases can be embedded in structure so as to reduce redundancies, one use case including another with refined information. Similarly, one BPR can be encapsulated into a RR and participate in a higher level BPR (as described in Section 9.4). Apart from the organisational structure and capability, presentation is another commonality of BPRs and use cases. It has been stated that text-based use cases are more important and useful than use case diagrams [9]. Similarly, BPR diagrams are developed after the establishment of a collection of RRs which make up the BPR, the text of which is described in a constrained natural language form, reconstructed from the original specification (Figure 8.7). Complete collection of RR descriptions for a BPR as such plays a similar role in agent-oriented requirements modelling as use case narratives in object-oriented requirements modelling.

An approach is suggested in [76] to obtain use cases from business processes formed in the style of UML activity diagrams. However, in AAM business processes are more valuable than use cases. Once a business rule is changed, all use cases that use the rule have to be changed, manually and individually. In contrast, BPRs are composed by RRs, dynamically at runtime, and so such changes are accommodated automatically. Related to this, another important reason is that they can be used to automate the later development phases and remain re-usable, rather than lose their value, like use cases typically do, after modelling is completed. The insufficient semantics of use cases is even a barrier in its support for model refinement [200]. Moreover, BPRs are also an enhanced form of specification compared with use cases. It is claimed in [37] that requirements have to be hierarchically organised, in order to be able to understand them, reason about them, refine them and use them to validate the developed products. However, difficulties have been suggested [8] in discovering and specifying useful use cases, and finding consensus about how to organise and manage them. More importantly, the hierarchical structure of use cases is not usually obvious. In contrast, the hierarchy of Business Process Rule and Business Rule is apparent and necessary, where the latter level refines the previous level.

BPRs integrate functions provided by all participant agents representing interrelated requirements specifications that span over different business domains. Thus interactions of internal requirements which map to representative agents or other modelling elements are clarified and made explicit. Therefore, requirements captured in BPRs are more fine-grained, concrete, and operational, due to its tangible RR components and agent performers. The completeness and consistency of compositional requirements in BPR can be validated collectively, whereas simple investigation of individual requirements in the absence of BPR may cause the omission of goal implications or combined participant responsibilities. The hierarchical decomposition of BPR-RR-PR eases the later development in contrast with use cases, where objects and their relationships are difficult to directly map to [37].

Furthermore, the infrastructure of the (agent, rule, and communicative act) message as structured in BPR are more appropriate for adaptivity than the (object, method, and method invocation) message structure in OO. Agent interaction and message composition and passing can both be configured according to requirements

changes reflected in rules as opposed to the situation where object interaction and method invocation message are fixed. The automatic interpretation of AAM models by agents reflects the change automatically with immediate effect as collaborative agents adapt their behaviour. Presentation of use cases in BPR style helps the migration of OO systems to AAM systems. Sometimes use cases are extended to include details about event flow, interactions among system entities, rules regarding condition of use, and even decision making processes [64]. In such cases use cases look much more like BPR, except they have no concept of agent. However, such use cases and BPR are both useful as they can be validated by users of the completeness and correctness of requirements.

6.12 A UML profile for AAM

A UML profile has been defined for AAM in response to the lack of the agent concept and knowledge elements associated with the agent in traditional OO modelling. Stereotypes, the UML's mechanism for introducing new types of modelling elements [60], are used. This profile or part of its package can be reused by others to extend the UML language and apply its customised modelling elements and gain the advantages brought by AAM such as adaptivity. A compatible means is thus provided through the profile to widen the use and adoption of AAM to ordinary OO-based development.

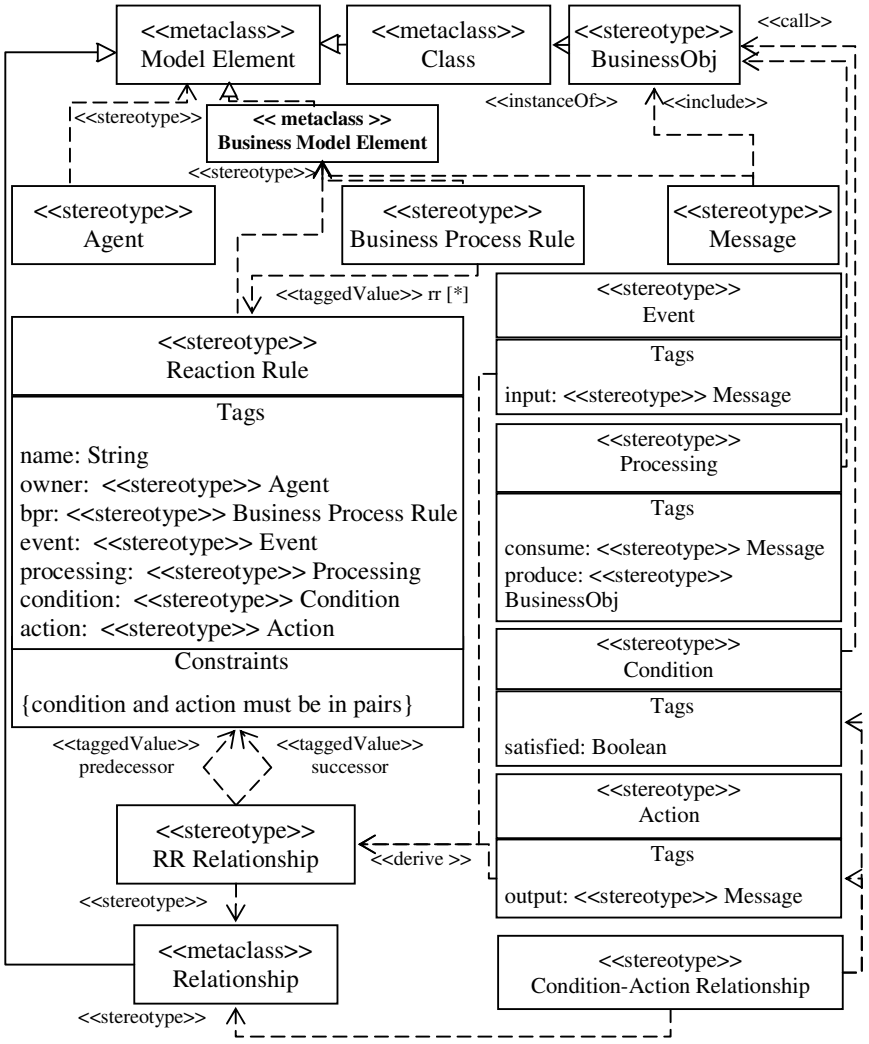

Figure 6.15: Stereotype declarations of AAM business model shown as around Reaction Rule

Some UML pre-defined [61] standard keywords or stereotypes are reused in this profile. Their semantics have been adapted as in the context of AAM, as shown in Table 6.1. Casting the AAM meta-model as a UML profile allows the reuse of standard UML notations, which implies that AAM model elements can be easily understood under widely agreed standards, and existing UML CASE tools can be used to support the specification of AAM models. Another agent-oriented UML profile, based on the Agent-Object-Relationship Modelling Language (AORML), is described in [161]. The necessity and advantage of the UML extension towards an agent aspect can be illustrated by a simple case that, customers or business administrators involved in a financial system, usually having to be modelled as objects in the same way as bank accounts or administrative policies, can now be distinguished as (human) agents. Although both agent and object are stereotypes of

the generic model element, their conceptual semantics are differentiated and can be used in various situations respectively. Business modelling elements have also been emphasised and these notions are made explicit for modelling. They are elaborated with their semantic relationships expressed in the profile. These can be further extended and adapted in other applications according to different business needs.

Table 6.1. Stereotypes adapted in semantics

Stereotype	Origin	AAM
«call»	A source operation in the consumer class invokes a target operation in the supplier class.	An operation defined in business objects is invoked in «Processing» or «Condition» stereotypes. Example: order. isOrderAttractive() == true
«instantiate»	An operation in the consumer model element creates an instance of the supplier model element.	A new business object is instantiated in «Processing» stereotype. Example: order = new Order (businessInfo) An «instantiate» is a stereotype of «call»
«destroy»	An operation destroys an instance of the classifier.	An instantiated business object is destroyed in «Action» stereotype after its usage. An «destroy» is a stereotype of «call»
«derive»	The source element may be computed from the target element.	«Event» and «Action» stereotypes decide the relationships between RR. «Event» decides the predecessor RR of the RR it belongs to. «Action» decides the successor RR of the RR it belongs to.
«instanceOf»	The client is an instance of the supplier.	A business object is an instance of class.
«include»	A use case includes another.	A business object is encoded into a message.

Adopting default semantics of some keywords presented in the UML documentation and adapting others, the profile illustrated in Figure 6.15 describes the semantic relationships between the modelling components of RR and other elements abstractly. It explains the four major components of Reaction Rule Model as shown in Figure 6.8 and the two following design models semantically.

- Event: *receive* Message, *decide* RR relationship (predecessor)
- Processing: *use* Message, *create/instantiate* BusinessObj
- Condition: *call* BusinessObj
- Action: *use* BusinessObj, *destroy* (useless) BusinessObj, *create & send* Message, *decide* RR relationship (successor)

Due to the externalisation of various rules, two distinct contributions to adaptivity can be identified here.

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 [62] 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.

The notion of Agent-Rule-Class (ARC) framework [1] [4] is about the assignment of rules to agents that enables the selection of agent partners and class assistants at runtime for the function of particular agents. The adaptivity aspect of this conveyed with the application mechanism is elaborated in Chapter 9.

Chapter 7: Implementation, Deployment & Tool Support

7.1 Introduction

The Model Driven Agent Architecture in AAM facilitates the interpretation of knowledge models by agents to inform their behaviour. The platform independent knowledge models can be annotated in XML, operated by a generic Agent Model. The specific agent platform of JADE is selected to demonstrate the change of the development paradigm and adaptivity achieved by using the approach, supported by tools at various levels. The principle published work with a focus on this topic of implementation and tool support can be found in [5] and [7].

7.2 Knowledge Model

Platform-neutral implementation of the knowledge model is critical in the adoption of the AAM approach across various agent development platforms available today. At the same time the aim is for easy maintenance of the knowledge model. For these reasons, it is proposed to externalise such changeable information into easier to maintain XML-based rules repository, and let agents interpret it and decide the current applicable rules on the fly. A template Policy Rule made up of an "IF" condition, a "THEN" action and a priority components alongside its XML format is illustrated in Figure 7.1, where op1 and op2 correspond to "less than", "equal to" or "greater than".

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

Figure 7.1: Policy Rule template

RRs are also represented and stored in XML just like PRs so as to enable machine understanding, human configuration, and runtime adaptation. As it is shown in the

Reaction Rule schema specification in Figure 7.2, one rule is composed by five elements. These are encoded using respective tags in XML. They include details about the triggering event, the processing of the event, a series of {condition, action} couplets and the priority. Here Policy Rules as previously specified have been extended. They are enriched in format and also include complete content, not only rules processing but also their ownerships as well as details of incoming and outgoing parties under various conditions. Now rules can be used to adapt both individual agent behaviour and inter-agent collaboration. Specifically, the shared schema specification declares the behavioural pattern of agents. This pattern defines the triggering event received, the event processing, a series of actions to be performed if corresponding conditions are satisfied, and the priority sorting rule orders. Concrete rules in this format are demonstrated in the case study in the next chapter for an actual business application. According to the schema, multiple {condition, action} pairs are allowed. Event and action sections detail message passing. Message contents can include any encoded object required by business needs.

```
<?xml version="1.0"?>
<xs:schema targetNamespace="http://www.cs.qub.ac.uk/~L.Xiao/"
xmlns:xs="http://www.w3.org/2001/XMLSchema"
xmlns="http://www.cs.qub.ac.uk/~L.Xiao/">
<xs:complexType name="msgcontent">
   <xs:sequence>
      <xs:any minOccurs="0"/>
   </xs:sequence>
</xs:complexType>
<xs:complexType name="msg">
   <xs:sequence>
      <xs:element name="from" type="xs:string"/>
      <xs:element name="to" type="xs:string"/>
      <xs:element name="title" type="xs:string"/>
      <xs:element name="content" type="msgcontent"/>
   </xs:sequence>
</xs:complexType>
<xs:complexType name="eve">
   <xs:sequence>
      <xs:element name="type" type="xs:string"/>
      <xs:element name="message" type="msg"/>
   </xs:sequence>
</xs:complexType>
<xs:complexType name="act">
   <xs:sequence>
      <xs:element name="type" type="xs:string"/>
      <xs:element name="message" type="msg"/>
   </xs:sequence>
</xs:complexType>
<xs:complexType name="con-acts">
```

```
   <xs:sequence>
      <xs:element name="con-act" maxOccurs="unbounded" minOccurs="0">
         <xs:complexType>
            <xs:sequence>
               <xs:element name="condition" type="xs:string"/>
               <xs:element name="action" type="act"/>
            </xs:sequence>
         </xs:complexType>
      </xs:element>
   </xs:sequence>
</xs:complexType>
<xs:complexType name="vars">
   <xs:sequence>
      <xs:element name="var" maxOccurs="unbounded" minOccurs="0">
         <xs:complexType>
            <xs:sequence>
               <xs:element name="name" type="xs:string"/>
               <xs:element name="type" type="xs:string"/>
            </xs:sequence>
         </xs:complexType>
      </xs:element>
   </xs:sequence>
</xs:complexType>
<xs:element name="rule">
   <xs:complexType>
      <xs:sequence>
         <xs:element name="name" type="xs:string"/>
         <xs:element name="business-process" type="xs:string"/>
         <xs:element name="owner-agent" type="xs:string"/>
         <xs:element name="global-variable" type="vars"/>
         <xs:element name="event" type="eve"/>
         <xs:element name="processing" type="xs:string"/>
         <xs:element name="condition-action" type="con-acts"/>
         <xs:element name="priority" type="xs:string"/>
      </xs:sequence>
   </xs:complexType>
</xs:element>
</xs:schema>
```

Figure 7.2: Reaction Rule schema

XML-based rules add precise definitions of agent behaviours to the Design Models, when the model elements are mapped to these definitions. This is something UML diagrams lack [9]. In general, each agent reacts to the receipt of a message by executing a rule using the following process shown in Figure 7.3.

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.

Figure 7.3: Reaction Rule in XML processing procedures

RRs can play the role of a connector [78] [79] or a contract [16], as they have the capability of evolving software architecture through 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, while the <processing> part serves internal computation. This separation of concern makes software evolution easier. For example, one rule can be updated only in its <event> - <message> - <from> or <action> - <message> - <to> part, achieving the effect of changing agent collaboration patterns. The main <processing> part can be maintained as a core and stable component while the result is notified to additional/alternative partners through the rule interface definition in the <action> part.

Pre-conditions, post-conditions and guards are inherently modelled in the RRs. The <event> section models the pre-condition, which indicates that an event message must be received before the rule is invoked; the <action> section models the post-condition, which promises an action message will be sent after the rule is executed; the <condition> section models the guard, which controls the rule execution path. At runtime agents check these mutable constraint requirements dynamically. This paradigm is much more flexible than the usual practice of using fixed generated

code. Thus each individual modelling component of business rules contributes adaptivity to the system as a whole.

7.3 Agent Model

Multiple agent platforms should be able to run the AAM knowledge model. Therefore, here only minimum requirements are defined, where agent capabilities must be supported by the chosen platform in order to, on one hand put knowledge model into their behavioural practice and on the other hand places minimum constraint on the implementation and thus allow a platform independent model.

Three categories of class method have been suggested for method design: query, mutation, and helper [36]. In this context a classification of agent roles/acts for runtime interpretation and execution of business models using runtime data as required by AAM is shown in Table 7.1. 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 of AAM. At the time when events occur, environmental information is accommodated into business models and agents communicate with business models to handle business needs. In AAM, agents act on business models, agents are subject and business models are object. 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 7.1: 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 business rules are dynamically retrieved as statements about the use of these acts performed on objects or concepts. The interaction of agents with previously unknown parties is thus enabled through business rules. There being no specific requirements of agent behaviour beyond these primitives makes AAM technology independent.

7.4 A different agent notion – less autonomy, more reliability

The notion of agent in AAM is somewhat different from that in an autonomic software unit. It is demonstrated in [168] that a general pitfall of agent-oriented software development directly attributable to the autonomous characteristics of agent is that agents are inherently unpredictable with uncertain nature and outcome of interactions. This leads to considerable unexpected behaviour and, more importantly, significant unpredictability about which objectives agents will pursue and which they will not.

In contrast, the responsibilities and interactions of AAM agents are not totally free, but instead they are restricted in business models as specified and optionally re-configured by business people. However, the lack of full autonomy actually brings advantages. Since traditionally a software agent accepts any message from anyone, it makes no commitment to the kind of response or any further message. Fully autonomous agents discover the responsible ones and collaborate in an unpredictable fashion and this does not guarantee the fulfilment of system services, a fact especially acute if real-time processing is required. It has thus been advocated that academic software agent systems be discarded and instead systems built on top of web services to produce real business services [69]. These advocates propose service agents evolving from web services. These service agents coordinate distributed processes with no fixed process model but with predictable behaviour that fulfils system services. Recognising the rationality of this argument, the AAM enables predictable and reliable service-based agent behaviour, the discovery and use of services defined in business models, but keeps and inherits the fundamental notion of agent. A decentralised and distributed architecture maintained, AAM has its agents individually maintain their own threads of control, choose behaviour from a selection

of configurable actions, but all of these behaviour only work to contribute to the production of useful results to the system. In this respect, a semi-autonomous characteristic of agent has emerged in AAM. This special AAM agent feature has been reported in [1].

7.5 Communication infrastructure

If every application is capable of mapping its own data from/to a common communication language, when data is exchanged between different systems, each system receives data by disassembling the common data structure from its source and reassembling the data in a form that it uses internally, and sends data in a converse procedure. What one system knows about or wants to know can therefore be understood and shared in a larger scope through communication with other systems, which either add it to their own knowledge or in return provide their own knowledge. Interoperability and cross-system collaboration becomes possible.

The aim in AAM is open interoperable system architectures and makes no assumptions about the use of the underlying agent platform. Instead, the generic Agent Model is advocated in favour of building distributed but interconnected collaborative systems, each of which is based on the architecture of AAM, but a specific agent platform preferred by an individual organisation can be selected freely. AAM is neutral with respect to both the agent implementation and programming language used for components that support the function of agents. As long as the communication among Agent Models conforms to FIPA Agent Communication Language (ACL) [65] [81] message format, such systems interoperate without concern for agent/object platforms. AAM makes use of the communication infrastructure of the existing agent platforms but its agents override the pre-defined behavioural model. Despite the fixed Interaction Protocol of FIPA, and the actual practice of existing agent platforms, where fixed class methods represent agent behaviour, agents in AAM use business models to behave and communicate dynamically. Agents know what incoming messages are expected to be and the appropriate outgoing messages as described in <event> and <action> of the relevant rules dynamically selected in that context at execution time and these rules are configurable. This contrasts with the traditional agents behaviour in which the messages to be passed or produced are fixed.

AAM agents running on different platforms exchange messages by encoding/decoding local objects. Traditionally objects should be written in the same language for the communication parties to comprehend each other. For example, in JADE [73], a particular agent platform, the following code is used to pass a List object, whose structure must be understood both by the sender and the receiver to enable their communication.

```
public void passList (String l) {
    ……
    // Fill the message content with a List l, containing the object
    fillContent(requestMsg, l);
```

By contrast, in AAM XML-based information is used as objects to be filled in or extracted from message contents. Hence, there is no requirement for the use of the same underlying object language for agents in different systems to understand each other. The universal use of XML in business models and elements representation makes interoperability wider in scope and easier to implement. Data binding techniques such as Java & XML data binding [66] can be applied to convert between object instances of agents and XML data. In Java & XML data binding, an XML document is represented by a business Java class, and the data is mapped from the document into the member variables of the Java class. These Java classes are tailored to the business need and generally match up with the element and attribute naming in the related XML documents. XML is used as the interchange medium for data. Various object languages and XML data binding can be used, and XML provides a data format for exchanging the contents of objects written in different languages. The information interchange process between two parties involves a marshalling process of converting an object into an XML document representation, and an unmarshalling process of converting an XML document to an object.

Following on the achieved interoperability, it would be also possible to encapsulate business processes using the AAM and publish them as web services to further integrate the proposed framework with existing technological infrastructures. Additional wrappers may be necessary to enable interoperation and let AAM services be invoked by clients and return results in an expected manner. XML messages are still used for information exchange, ACL and SOAP (Simple Object Access Protocol) message conversion is required. According to W3C, “Systems interact with the web service in a manner prescribed by its description using SOAP messages,

typically conveyed using HTTP with an XML serialisation in conjunction with other web related standards" [67]. An obvious advantage of using AAM services over classic web services is, messaging between AAM services and their clients won't require agreed interfaces. This means clients can access services previously unknown by assembling messages in specified styles without re-coding. Services can be dynamically added or changed without server side re-build or restart and clients are served continuously with immediate new effects if required. Section 9.5 discusses a topic related to this.

7.6 Implementation using JADE

As AAM is an agent-oriented software development approach, JADE is selected as a sample agent platform to demonstrate the advantages that AAM brings to the agent system development paradigm and the convenience of adapting such systems. A simple e-business example is used to demonstrate the specific JADE coding practice and compared with the AAM approach. Supporting tools developed for AAM approach are introduced in this section. A paper with a focus on this specific topic can be found in [210].

7.6.1 Java Agent Development Framework

Java Agent DEvelopment Framework (JADE) [73], fully coded in Java itself, is a Foundation for Intelligent Physical Agents (FIPA, IEEE standards organisation) [81] compliant framework, popular for developing Java agents. JADE provides a distributed agent platform, which includes the Agent Management System (AMS), maintaining a directory of agent identifiers (AID), the Directory Facilitator (DF) offering a 'look-up' service, and the Agent Communication Channel (ACC) controlling message exchange within the platform. Each JADE agent is an instance of a user defined Java class. An agent can have multiple tasks executing concurrently, each implemented as one ore more behaviours. JADE starts up an agent by giving it an AID, registering it with the AMS, and executing a *setup()* method, in which customised behaviours can be added using the *addBehaviour()* method. Multiple behaviours can be added, scheduled, and executed starting with the first one in the queue after setup. Behaviours can also be removed whenever necessary by using the *removeBehaviour()* method. A *takedown()* method can be used to implement any cleanup operation before the agent thread is destroyed, while

doDelete() stops agent execution immediately. Agents communicate via Agent Communication Language (ACL) messages. Attributes can be set for ACL messages referring to different performatives (purposeful actions performed during conversations between the agents) such as REQUEST, AGREE, INFORM, and so on. Java objects can be encoded into messages as 'payloads' and reconstructed later. In the messages, the AID of the receiver agents can be specified as a message destination, the ACL expression mapping to and from Java objects as message contents. An agreed-upon common language and ontology provides the syntax and semantics for the conversations. The methods *setContentObject()* and *getContentObject()* can be used to pass Java objects through agent communications. A *send()* method allows an agent to send a message. Messages received by an agent are put into the agent's private queue, typically accessible by using the *receive()* method. JADE, as a specific agent development platform, is selected to develop prototypes that demonstrate the approach.

7.6.2 Web-based tools for Policy Rules & Concepts

JADE agents implemented as Java objects are not adaptive since agent behaviour is decided at design time and written in fixed code. Therefore, a new behavioural pattern is obviously required. This section demonstrates the proposed externalisation of context dependent metadata with respect to policies and concepts and the use of agents to interpret their behaviour dynamically at runtime from it.

The next chapter provides a comprehensive case study with AAM. To explain implementation details and aid understanding a simplified e-commerce example will be used here.

E-business example 1: Suppose a customer agent is communicating with a seller agent. The customer has obtained from the seller a list of currently available goods and now selects items and requests their corresponding prices. Upon any enquiry, the seller agent retrieves its goods data with their default prices. Discount policies subject to amendment exist, specifying in various conditions the discounts that customers can enjoy. Thus, appropriate discounts applicable to the current customer, according to the customer profile, are applied before the actual prices are presented to the customer. Figure 7.4 illustrates customer and seller agents as implemented using JADE.

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

Figure 7.4: Code segment for customer agent

Figure 7.4 shows how when the customer has chosen items from a user interface, the customer agent responsible for the shopping task of this customer contacts the seller by constructing a new message, setting its destination and action to perform, and finally sending it off. “Item” is a concept representing any sort of goods for sale with a type slot and a price slot. “Enquiry” is an agent action representing an action that can be performed by some actor agents. Both the concept and the action must be implemented as Java classes and declared in a user-defined ontology that will be shared among communicating agents before being put into use. Here “Enquiry” as an action encapsulating the requesting customer and the requested item can be sent as an action by the customer to the seller and understood by both agents to perform a price enquiry.

```
/* SellerAgent.java */

public class SellerAgent extends Agent {
   private rulesParser = new qub.aam.common.RulesParser();
   private ContentManager manager  = (ContentManager)getContentManager();
   protected void setup() {
      // check new customer request every second
      addBehaviour(new TickerBehaviour(this, 1000) {
         protected void onTick() {
            ACLMessage msg = receive(enquiryTemplate);
            try {
               if(msg != null) {
                  ContentElement ce = manager.extractContent(msg);
                  if(ce instanceof Action) {
                     Enquiry enquiry = (Enquiry) ((Action)ce).getAction();
                     AID buyerId = enquiry.getBuyer();
                      Item item = enquiry.getItem();
```

```
                    /* find and apply the rule that is relevant to customer discount and also applicable
                       to this particular customer */
                    String discount = executeRule("customer.discount",   buyerId);
                    double price = item.getPrice() * (1 - Double.parseDouble (discount));
                    ACLMessage inform = msg.createReply();
                    manager.fillContent(inform, price);
                    send(inform);
                  }
                }
              } catch (Exception e) {}
          }
      } );
……
   }
}
```

Figure 7.5: Code segment for seller agent

When a message that fits in the context of this enquiry conversation is received, the content of the message is extracted and the action object restored, in this case an "Enquiry". The Java code for the seller agent is shown in Figure 7.5. To simplify, suppose at one time only one item is selected by the customer for an enquiry, then the seller needs to find out the discount applicable to a potential customer for a specific item at that particular time. Because business policies must change continuously to achieve the best business opportunities, using fixed code such as complex logical structures greatly increases the maintenance burden. Policy Rules formatted in Section 7.2 are used. Figure 7.6 gives three concrete and self-explanatory examples of actual rules in the demonstrator system.

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

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

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

Figure 7.6: Sample rules

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

```
/* executeRule is used by SellerAgent to apply the applicable rule */
private String executeRule(String actionTarget, AID buyerId) {
   Vector globalRules = rulesParser.getRules();
```

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

Figure 7.7: Code segment for rule execution

The rules are executed by agents by using a facilitating RulesParser module, the code for which is shown in Figure 7.7. The RulesParser module interprets the operational Java objects from the XML code. Rules relevant in the context of customer discount are retrieved, evaluated in the order of ranked priority and the rule with the highest priority evaluated as being satisfied would be eventually executed, returning a discount value. The method testCustomerCon() retrieves the customer profile and uses this information to evaluate the rule condition. The code of the method is omitted here in the interest of conciseness. However, another interesting

process is involved. The seller agent does not hold all customer information, since this is managed by the customer agent. Thus, the seller agent must request customer information from the customer agent in order to calculate the actual price with regard to the discount. The process is adaptive in two aspects. Firstly, the rules set in the business environment are dynamically defined, so that the rules being enquired upon are changing over time. Secondly, at any given time enquiries on rules are performed dynamically because customer satisfaction against the same set of rules varies. Discounts to be applied are decided by these two factors. For example, three rules in Figure 7.6 will be tested sequentially. Thus a researcher named "Liang Xiao" can enjoy a discount of 15% as a result of the first rule while another researcher named "Des Greer" does not satisfy the first rule but the second one. Details of these two aspects of adaptivity follow.

The Policy Rules Manager Agent (PRMA), having a general purpose of managing the rules repository, is dedicated to evaluate rules in order of priority, instead of specific agents individually doing the same routine of rule evaluation and execution. Ordinary agents need only contact the PRMA and get the computed result. The PRMA retrieves and ranks rules each time when a request is received, thus guaranteeing that the updated list of rules is available. Each time where the top ranked rule is found unsatisfactory, the PRMA switches to the next one, and so on, until the satisfactory rule with the highest priority is found. The agent responsible for the customer "Des Greer" and the PRMA communicate as shown in Figure 7.8. This extended UML diagram shows agents, rules, and their interactions via message passing. R1 is rejected when it is found that the name of the customer does not match with what is required by the rule R1. The PRMA then switches to R2, which is satisfied because the customer is found to be a researcher when the information is obtained from the customer. In this case R3 is never used. This process is dynamic. For example, the PRMA will apply R1 to a customer named "Liang Xiao" and the process will finish, while R3 would be applied to a premium customer with no research background. Rules can be externally changed as required and will take effect automatically without amendment of the seller agent code (Figure 7.5) or rule execution process code (Figure 7.7)

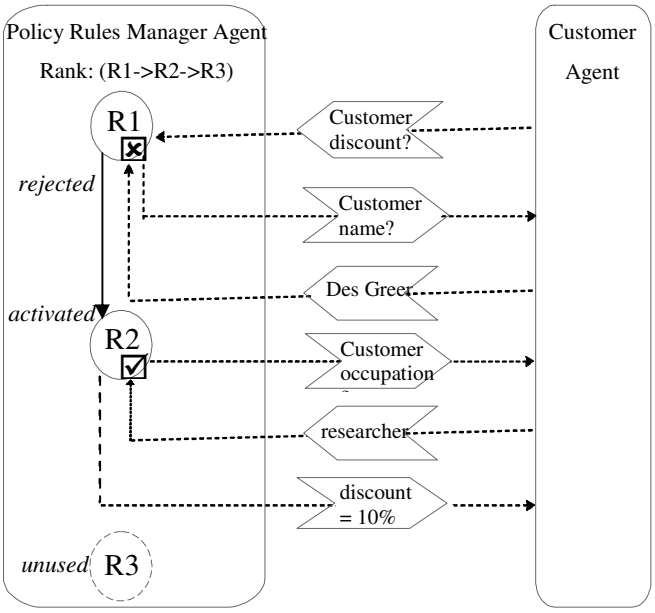

Figure 7.8: PRMA evaluates PRs for ordinary agents in message passing sequence diagram

All rules in the repository are formatted in XML conforming to the template shown in Figure 7.1. Therefore, generically, the steps the PRMA takes to find and execute rules on behalf of other agents are:

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

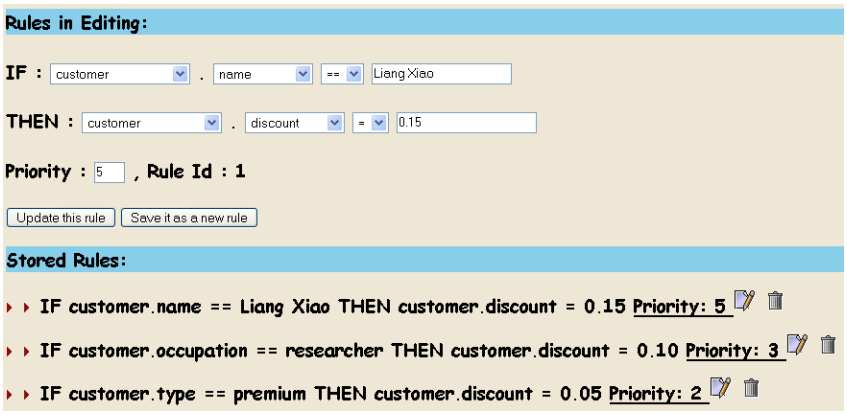

Figure 7.9: Policy Rules Editor interface

The provision of tools for business analysts and strategy makers adds to the dynamic capability of the approach by allowing on-the-fly configuration of policies. Figure 7.9 shows a web-based editor that has been developed. The business entities which are used to compose these rules are abstracted and listed on the interface for simple selection, so that people who have no programming experience can specify their relationships as rules in a straightforward way. The current available rules list is retrieved from the XML repository and shown on the interface for viewing and editing. Whenever rules are updated from the interface, the repository is updated accordingly. This easily allows the adjustment of existing business strategies and the addition of new ones. For example, a new policy with a higher priority than all existing policies may be defined that grants all staff from a particular university a discount of 30%. Then customer "Des Greer", being staff from that university can benefit from this new policy since it overrides all the others automatically.

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

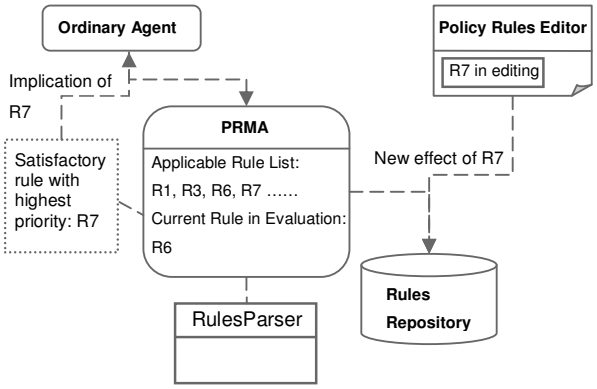

Figure 7.10: AAM policy adaptation tool support

Figure 7.10 shows the interactions between Ordinary Agents and the PRMA which involve the dynamic message-passing process from Figure 7.8, and the role of the Policy Rules Editor shown in Figure 7.9 which updates the rules repository. By replacing previously fixed agent behaviour by dynamic rule execution which includes an iterative interaction process evaluating externalised rules, agents can now behave adaptively not only to the current conditions of that particular agent but also the policies set at that moment. Recall that in using the Strategy Pattern all strategies must be predicted, in a hard-coded, non-configurable means. This limitation has been overcome here by the separation of changeable policies from the main agent programme. This subsection demonstrates the adaptivity achieved through externalised policies. Additional support for adaptive use of Business Concepts further supports the adaptive application of rules.

Business Concepts refer to the terms allowed in business rules. In the AAM approach, a Business Concepts Editor allows new objects/concepts to be declared via a web interface, along with new attributes/properties. Once they have been registered, they are immediately available and can be accessed via the rules editor. Therefore, new business behaviour involving these new concepts can be created, reflecting new requirements, as agents execute the new rules defined on them. For example, Figure 7.11 shows the tool web interface being used to add a new property of “credit” to the existing concept of “customer”.

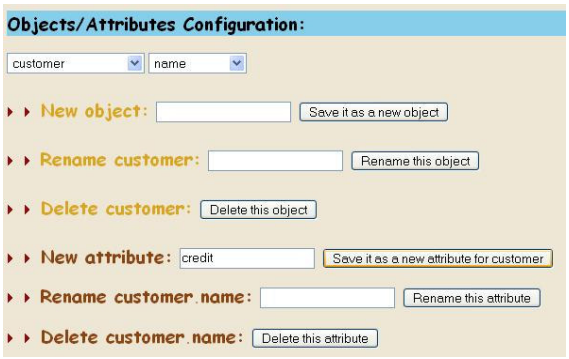

Figure 7.11: Adding "credit" as a new property to the existing "customer" concept through Business Concept Editor

The updated XML document is as shown in Figure 7.12 with an additional entry registered with the "customer".

```
- <concept>
    <name>customer</name>
  - <properties>
      <property>name</property>
      <property>type</property>
      <!-- … more properties … -->
      <property>credit</property>
    </properties>
    <behaviour/>
  </concept>
```

Figure 7.12: Updated Business Concept in XML

This makes it possible to define the following new rule via the Policy Rules Editor as shown in Figure 7.13.

IF customer. credit > 5000 THEN customer. discount := 0.20 Priority: 1

Figure 7.13: New rule defined using the new concept property "credit"

Without changes to the rules already defined and without restarting the agent system, the new attribute is registered and a new rule using it has been added to the business rules repository, with immediate effect. Customer "Des Greer" and "Liang Xiao" will both enjoy a discount of 20% instead of 10% and 15% respectively, if their credit values are higher than 5000, because this rule is applicable and has the highest priority, provided all rules defined in the previous section are not changed.

7.6.3 Application tools for Business Process Rules & Reaction Rules

Considering the interactions shown in Figure 7.8, it is obvious that dynamic processes like this are normal but cannot be foreseen in advance. Support for adaptivity of agents in choosing at runtime other agents with which to form dynamic interaction patterns is difficult using traditional technologies or platforms. Objects normally have to interact in a fixed mode. This restriction on component dependencies must be removed from agents to allow free communication even in unpredicted patterns arising from new needs or goals. This idea of agent society has led to the term multi-agent systems (MAS) where agents can coordinate through cooperation or competition in different conditions for different purpose, resembling human society. Such systems are highly dynamic and must be adaptive in terms of internal communication.

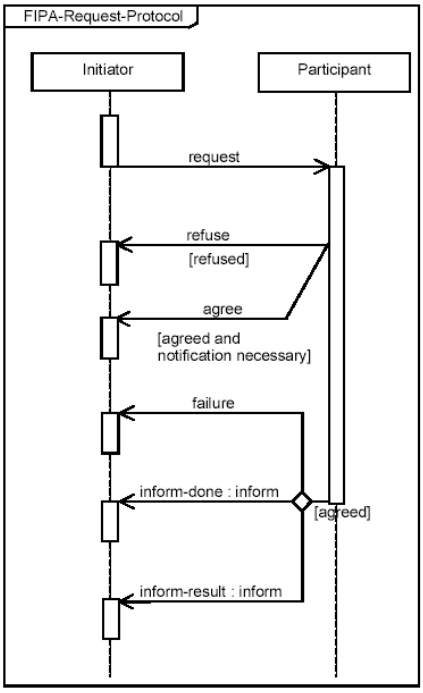

Figure 7.14: FIPA-Request protocol [81]

FIPA specifies a set of standard Interaction Protocols: FIPA-Request, FIPA-Request-When, FIPA-Query, FIPA-Subscribe, and others. These can be used as agent conversation templates. JADE distinguishes the "Initiator" role and "Responder" role, being agents starting conversation and joining conversation after contact. "AchieveREInitiator" and "AchieveREResponder" implement the roles required by most protocols. For example, FIPA-Request protocol as shown in Figure

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

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

```
    protected ACLMessage prepareResultNotification(ACLMessage request, ACLMessage response) {
      // if the catalogue is prepared, send it to the customer
      if (checkCatalogue()) {
        ACLMessage informCatalogue = request.createReply();
        informCatalogue.setPerformative(ACLMessage.INFORM);
        informCatalogue.setContentObject(productCatalogue);
        return informCatalogue;
      } // else send a FAILURE message
    }
} );
```

Figure 7.15: Code segment of catalogue request conforming to the FIPA-Request interaction protocol

The code in Figure 7.15 is extracted from the CustomerAgent and SellerAgent programmes and is as required by JADE. It is used when the customer requests the available goods list from the seller before he/she chooses items and the seller evaluates the discount. The customer agent must initially construct and send a request message, the seller agent must respond to the request with an AGREE/REFUSE message, followed by an INFORM-RESULT/FAILURE message, and finally the customer handles the result. The JADE programming model must be followed exactly to implement the protocol defined in Figure 7.14.

The fixed dependencies as required by the FIPA/JADE interaction protocols impose a restriction on adaptive collaborative behaviour in agents. This restriction can be overcome to achieve adaptive inter-component dependencies by using XML-based rules to externalise dependencies as metadata. Together with the intra-component adaptivity for individual component functions presented in the previous section they cover both inter- and intra- aspects.

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

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

The agent interactions for this scenario are shown in Figure 7.16. In this example, an order is placed by a customer, processed by a retailer who, in turn makes a call for proposal to a company. The company replies with a proposal, if the order is attractive and the retailer accepts it, if the proposal is satisfactory. The process completes with an acknowledgement from the company to the retailer who then acknowledges the customer.

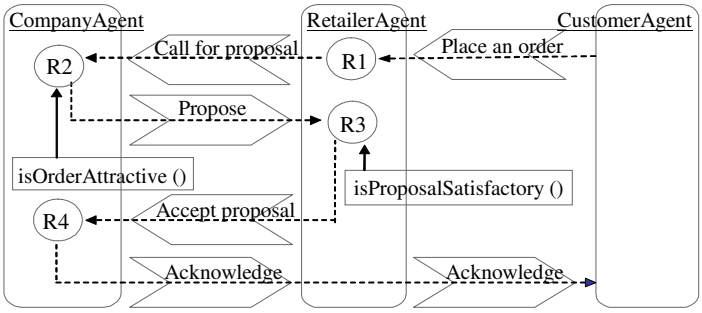

Figure 7.16: Message passing sequence diagram for changed inter-agent relationships

Using most existing approaches it would be expected that the system would need a complete reconstruction after such a dramatic change, even if using agents and so requiring the introduction of a new agent and new dependencies of the existing agents. In the AAM, adaptive agents, however, can be exempted from such inevitable code surgery if previously required JADE behaviour is not coded directly, but instead, they interpret at runtime their behaviour from rules as shown in Figure 7.16, the definitions of which are externalised in an XML repository.

Figure 7.17 shows the processing components of the rule R2, according to the steps described in Section 7.2. It is defined among all emergent rules to meet the new requirements. Such addition enables the CompanyAgent (originally SellerAgent) to understand its newly required interaction with customers through a new dedicated agent called RetailerAgent as their mediator. Similarly rules on the customer site shall be changed accordingly and new rules added to the new RetailerAgent.

Event: receive a “Call for proposal” message from “RetailerAgent”;

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

Condition: check this order of its attractiveness;

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

Figure 7.17: Rule R2 as shown in the diagram of Figure 7.16 performs according to the Reaction Rule schema

During the rule identification and definition process, concepts like “order”, “proposal”, and “attractiveness” can be expressed explicitly. It is up to the designers that designate classes and methods for these, later on. For example, a criterion that order is evaluated as attractive can be designated as the checking of a method on a business object: order. isOrderAttractive () == TRUE.

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

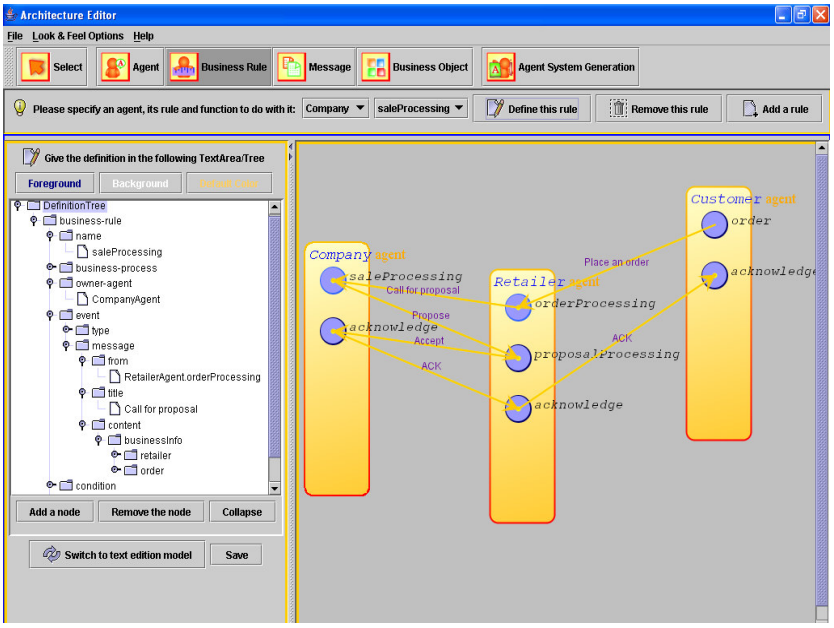

Figure 7.18: Tool support for inter-agent collaboration specification

The same RulesParser that has been used as a JavaBeans component serving the Java ServerPage based Policy Rules Editor shown in Figure 7.9, a facilitating module

by the customer agent to retrieve and execute applicable rules, is used a third time here by a Java Swing based diagrammatic editor. Figure 7.18 shows the construction of agent communication diagrams in its main panel and the definition of rules via a tree structure. The tree structure realises the XML schema defined in Figure 7.2 and XML based rules are generated from it and stored in the repository. These rules decide the agent communication pattern shown diagrammatically. The edition of diagrams in the main panel passes the specified agent relationships to affected rules in the left panel and vice versa so that diagrams and trees are consistent. The RulesParser interprets XML rules and shows them on the tool for business customisation and it also enables agents to execute rules in a similar pattern like the customer agent does as shown in Figure 7.7. When the agent system is running, both this tool and the web-based ones can be used by business people continuously to maintain business knowledge. Agents always use the up-to-date rules and bring business requirements into reality as soon as they become available through the configuration using the tools. Figure 7.19 summarise the process and demonstrates the architecture of the system extended from Figure 7.10, agents collaborating freely and forming any required interaction pattern by specifying condition/action pairs.

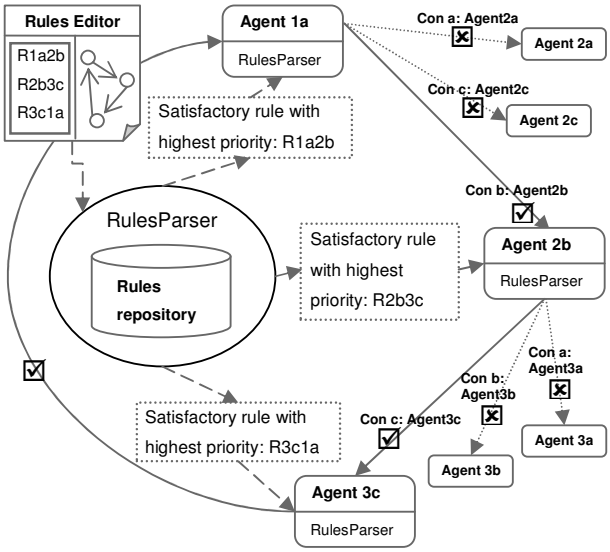

Figure 7.19: AAM adaptive dependency pattern

Additional support of adaptive use, invocation, and replacement of external classes in support of rule executing, for example, the evaluation of order attractiveness method has been described in [2]. It would be useful when an alternative version of a class can be found and used to replace an old version by

agents on the fly to achieve a changed effect. This will be discussed in details in the following chapters.

7.7 Deployment

The deployment of an implemented system using the proposed approach is shown in Figure 7.20. The actual agent system (❸ in Figure 7.20), running on the JADE platform in a distributed network, is initialised according to the required interactive agent models provided by the supporting tool (❶ in Figure 7.20). A central XML-based business rule repository (❷ in Figure 7.20) is deployed in the network, containing the rule definitions and the registered business concepts that are used by the rules. The XML parsing package is implemented as a JavaBeans component, responsible for parsing the XML format of business rules and presenting the parsed business knowledge in the tool. The tool is continuously used by business people to maintain requirements (❶ → ❷). The edition through the tool for the requirements change is saved in the XML repository using the same JavaBeans.

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

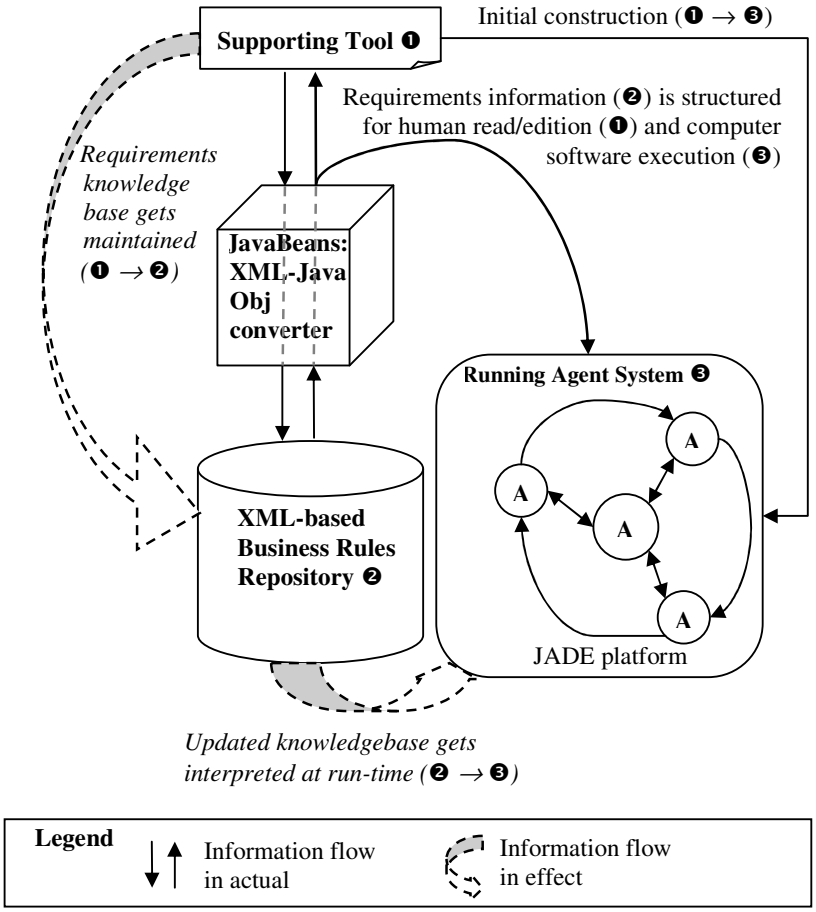

Figure 7.20: Deployment of the system

Figure 7.21 shows the pseudo code that "CompanyAgent" shown in Figure 7.16 will interpret from the "saleProcessing" (R2) rule to execute as one of its behaviour. The XML definition of this rule is found in [2]. Generally, an AAM system runs on the JADE platform and can be in a distributed network. All agents access the central XML-based rules repository via the XML parsing component. By using this package, agents can do the comparison to check the applicability of rules (① in Figure 7.21) and run pre-defined statements embedded in the XML tags of the rules (②,③,④ in Figure 7.21). These are interpreted from the XML specification of the rules. While the system is running, the rules specification can be continually changed through the supporting tool. This allows dynamic adjustment of agent communication structure and therefore the software architecture of the system. The shared module for the XML parsing and Java converting component is called "Rule" in here, being able to access the XML definition of rules and assemble corresponding objects, is used by

all agents. The methods getPriority(), getEvent(), and getAction() are provided by "Rule".

```
thisAgent. addBehaviour (Rule thisRule) {
    thisBehaviour. setPriority (thisRule. getPriority ());
    Order order;
    Proposal proposal;
    Message m = thisAgent. receiveMessage ();
    while (m != null)
    {
       Agent fromAgent = m. getSenderAgent ();
       if (fromAgent. equals
          (thisRule. getEvent (). getMessage (). getFromAgent ()))     // ①
       {
          /* the rule is applicable to the received message */
          BusinessInfo businessInfo = (BusinessInfo) m. getContentObject ();
          order = new Order (businessInfo);   // ②
          if (order. isOrderAttractive ())  // ③
          {
             /* the condition of the rule is satisfied */
             proposal = order. createProposal ();   // ④
             Message m2 = new Message ();
             m2. setContentObject (proposal);
             Agent toAgent =
                  thisRule. getAction (). getMessage (). getToAgent ();
             m2. addReceiverAgent (toAgent);
             thisAgent. send (m2);
             /* update this agent's beliefs */
             thisAgent. addBelief (System. getCurrentTime (), fromAgent, m);
          }
       }
      m = thisAgent. receiveMessage ();
    }
}
```

Figure 7.21 Pseudo code for behaviour of "CompanyAgent", mapping to its "saleProcessing" rule (R2)

7.8 A new development process model

One of the challenges facing the traditional software development process for MAS is the need for a new efficient process model, suited to the abstraction of agent-based computing [120]. The AAM framework introduces a parallel development and maintenance process for MAS so that different roles can be played simultaneously in system development and thus it becomes more efficient. This advantage is attributed to its hierarchical structure of Agent-Rule-Class, as shown in Figure 7.22. In three separate layers a system is developed and maintained by actors playing three types of role, each specialising in one of the layers. i) The AO expert role is concerned with strategic selection of an agent running platform appropriate for their specific domain, such as JADE [73]. This means that systems built using the AAM framework are not limited to a particular platform, since all platforms comply with the FIPA [81]

specification and are exchangeable with no effect to the systems running on any specific platform. The currently used platform may be discarded in favour of another with additional features and better performance. ii) The business analyst role concerns those familiar with the domain and relates to the development and maintenance of business models, including organisational structures, policies, and ontologies. Since the upper layer agents and lower level objects run independently from the business rules but reflect the changed requirements captured in rules in system behaviours. This can be done without intervention by developers, rather by business customers directly, requiring no system re-development or re-deployment. This also advances a model driven architecture approach. iii) Once OO developers have implemented a layer of supporting components, the need for maintaining the individual atomic functions is comparatively low, as they capture the most fundamental operations and it is up to the business rules that dynamically use them. Alternatively, these OO components could be COTS bought from third parties and the updating of them is up to the component providers.

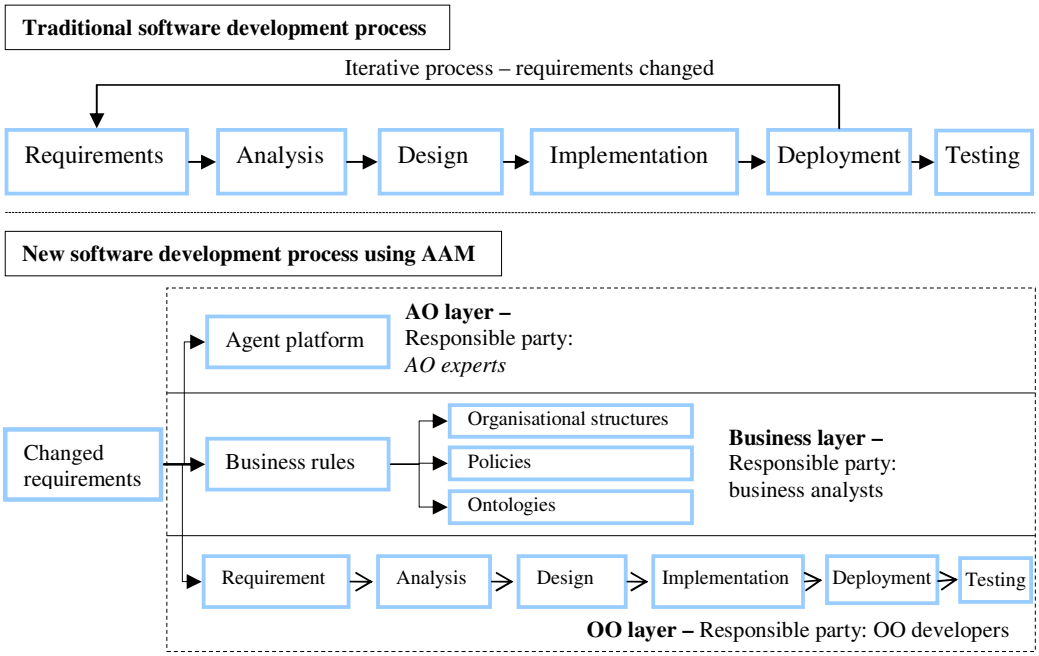

Figure 7.22: Parallel development and maintenance of MAS based on AAM boosts separation of concerns and efficiency

Chapter 8: Case Study

8.1 Introduction

In the previous chapters the AAM approach was described and its advantages over traditional agent-oriented software development in implementation illustrated. In this chapter, the complete software development process, starting from requirements analysis and design modelling through to the executable system development is discussed using a railway management case study. The knowledge models are built step by step, coupled with the agents which will use them. The overall system architecture described in the last section of this chapter will illustrate how all these components work together.

8.2 Case study

An actual national railway 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 sub-divided into *Business*, *Incident*, and *Execution* domains. These areas or domains are closely linked. For example, an infrastructure fault (Infrastructure Management) may block the access to track and cause rescheduling of train services (Train Running).

Briefly, Train Running Business domain supports the principal service to customers, including delivery of planned train paths and response to requests for further train paths. Relating to the domain, Train Operators run train journeys on the network. Train Operators are normally freight or passenger train operating companies. Each train journey is first supplied in the form of a plan, either as part of the working timetable (planning), or as a result of a request from a customer (re-planning). Railway asset faults/incident will cause train service re-planning as part of the case study. Although the running of train services itself is not within the scope of the case study of this thesis, it has been studied thoroughly in [1].

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.

Table 8.1 summarises roles of major business domains and actors in the fault management scenario of the case study.

Table 8.1: Roles of actors and domains

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

Functions in several business domains coordinate to manage faults/incidents and make rectification wherever necessary. Train services that will use the faulty assets may have to be rescheduled, and once assets recover from faults, train services come

back to normal. The system must have the capability to respond to unexpected faults and consider the business implications of decision making for rescheduling train services.

A large number of specific functional requirements tables have been documented for each domain. One of them, *IMI-HandleFault*, is given in Table 8.2, 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.

Table 8.2: 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*)

Particularly, the requirements document for the case study is organised by business domain. This is similar to the IEEE recommended template, in which specification is organised by features [163]. A feature is an externally desired service by the system that may require a sequence of inputs to affect the desired result. A domain usually consists of multiple related features and domains interact regularly for shared goals. In Table 8.2 “HandleFault” belongs to the IMI business domain, one of the domains that have been documented in the original “Production Function” specification. Also “AcceptFaultReport” and “NoticeFault” are in the same domain. Thus, falling into the same knowledge domain, these requirements should be managed by the same agent, here named IMI agent. This agent is responsible for the IMI business domain, detecting and handling of asset faults corresponding to the requirements of the domain. The services provided by each domain have co-related features, so that it is appropriate to organise them into the same domain.

The particular manner of documentation of requirements by domains and tables provides convenience for agent identification in this case. However, this manner is arguably a generic means, applicable to many large applications. The IEEE standard 830-1998 for Software Requirements Specification (SRS) [163] provides a template for documenting a software requirements specification. Organised by domains, the rail track specification largely follows this but has some extra fields, the IEEE standard leaving the organisation style to the analysts. In the IEEE standard every specific functional requirement is required to include input (stimulus), output (response), and a function as performed by the system in response to the input or in support of the output. Given that most of these components should be present, whatever the format, only a modest effort is required to transform any alternative format specification to this format.

An analysis method has been proposed for building a similarly structured semi-formal representation [164]. The method identifies the current requirements via a reformulation in which the input and output references are made explicitly visible. For requirements formulated in such a structured manner four classifications are made: i) requirements on input only (independent of output); ii) requirements on output only (independent of input); iii) function/behaviour requirements (output is dependent on input); and iv) environmental requirements or assumptions (input is dependent on output). These perfectly map to the individual function components here of “Cause & Information Used”, “Outputs”, “Required Effect”, and

"Assumption", respectively, the last two containing the pre-condition and post-condition of functions in the chosen standardised format. Functional requirements template with a form of "Identifier", "Description", "Precondition", "Ordinary sequence", "Postcondition", "Exceptions" based on user case description has been suggested in [204]. Another similarly structured specification in a standard form of "Inputs & Source", "Outputs & Destination", "Action & Requires", and "Pre-condition & Post-condition" is presented in [63].

8.3 UML representation and AAM

A UML Activity Diagram, showing the activities and flow of control for the chosen case study scenario is given in Figure 8.1, as a means to enable better understanding of the system.

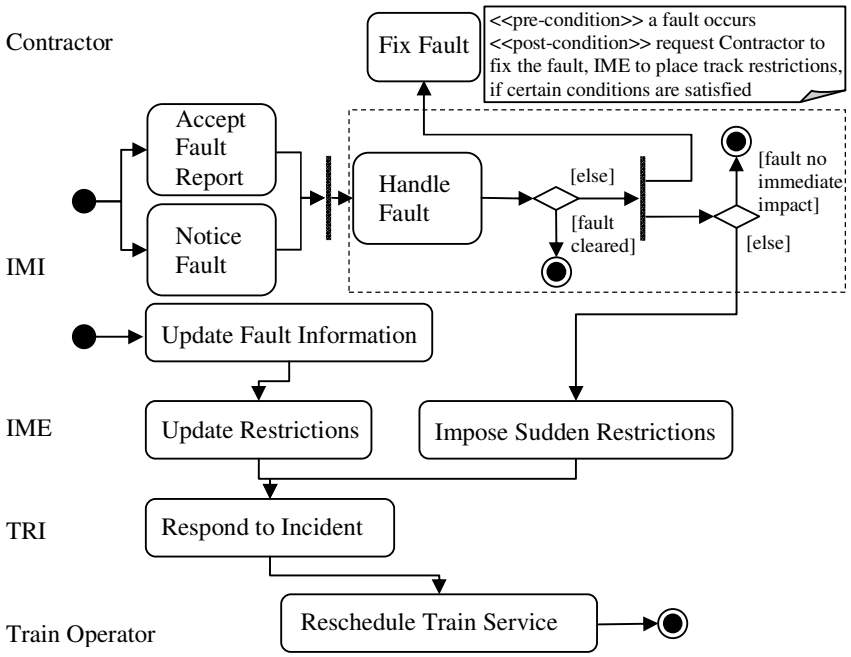

Figure 8.1: A UML Activity Diagram for the case study

Figure 8.1 demonstrates the handling of faults, the subsequent imposing of restrictions, and rescheduling of train services which together compose the process of fault management. Sequence and control flow of activities in the system has been shown to illustrate the relationships among activities. One participant acts after another if determined to be requested and this contributes to the formation of control flow. Decisions are made in the control flow and the system completes a sequence of actions ultimately. Implicitly, events flow out from selected branches in the control

flow as its outcomes and become the source that triggers the target activities. On reaching the target, an event accomplishes a transition between activities. Activity Diagrams provide a task-centric view of the behaviour of objects. When object-oriented systems are being implemented activities are statically assigned to objects and the message passing patterns between objects in the systems are fixed. UML activity diagrams are not inherently object oriented and, due to their flowchart heritage, mapping to object-oriented concepts is difficult [37].

Figure 8.1 also demonstrates the pre-condition and post-condition of *IMI-HandleFault* as given in the form of Object Constraint Language to assert business rules on top of UML models [59]. OCL can be used to express pre-conditions and post-conditions of operations and invariants of operations or classes. Applied to an operation, a post-condition specifies what has to be the result of the operation without saying how to achieve this result, thus ensuring separation of interface and implementation. Pre-conditions describe the system state as expected by the operation before execution, thus defining the responsibility of the caller. An invariant, applied to a class, must always be true at any moment an operation is called [37]. These three kinds of assertions should be expressed in OCL if implementation aspects are to be abstracted from in order to allow for business components substitution. However, OCL elements are attached to other modelling elements, separated from the main control flow rather than integral components. This is due to the lack of support for specifying constraints in activity diagrams or use cases. MDA recognises this deficiency and uses integral OCL for automation.

The initial step in the AAM development process is agent identification and requirements assignment to agents. Note in Figure 8.1, IMI, IME, and TRI represent internal actors that are responsible for the function of their respective business domains. As described in Chapter 6, these domains should be delegated to respective agents, who have the knowledge concerned with those domains. It is through the agent collaboration that domain functions represented in tables such as shown in Table 8.2 are fulfilled. Principle modelling elements of this scheme are given in Table 8.3 alongside their counterparts found in UML Activity Diagrams, playing roughly the same roles. After the replacement of the modelling elements and semantics, Figure 8.1 is still valid and useful but is read differently. Now agent behaviour is constrained by Reaction Rules, according to which agents pass messages. Messages are structured for communicative acts among agents. One

incoming message is an event that triggers an agent to react. After some decision making, the agent generates messages as its action, which becomes events to other agents. Agents cooperate in business processes for the achievement of common business goals through their collaborative behaviour [42]. Rules eventually become agent behaviour that will tell responsible agents how to realise requirements in the implemented multi-agent system.

Table 8.3: UML and AAM elements

UML element	AAM element
Activity Diagram	Business Process
Actor	Agent
Activity	Reaction Rule
Transition (sequential)	Message
Transition (conditional)	Decision Tree (branch)
Transition (incoming)	Event
Transition (outgoing)	Action

8.4 Goal-mapping for case study

Careful investigation of the full user requirements of the rail track system reveals that the safe delivery of train service is the first and foremost goal desired by its stakeholders. This top level knowledge consists of the business goal of "Fault Management" found in the excerpted case study description. Whereas this goal may have been used as the basis to construct multiple functional requirements tables as shown in Table 8.2, the link from individual functional requirements may not have been maintained in the requirements document. Figure 8.2 demonstrates the decomposition process for the top level goal of the overall system and its sub-goal "Fault Management", as the top level goal of the particular case study presented here. Business Process Rules are later derived from the graph, as described in Section 8.8.

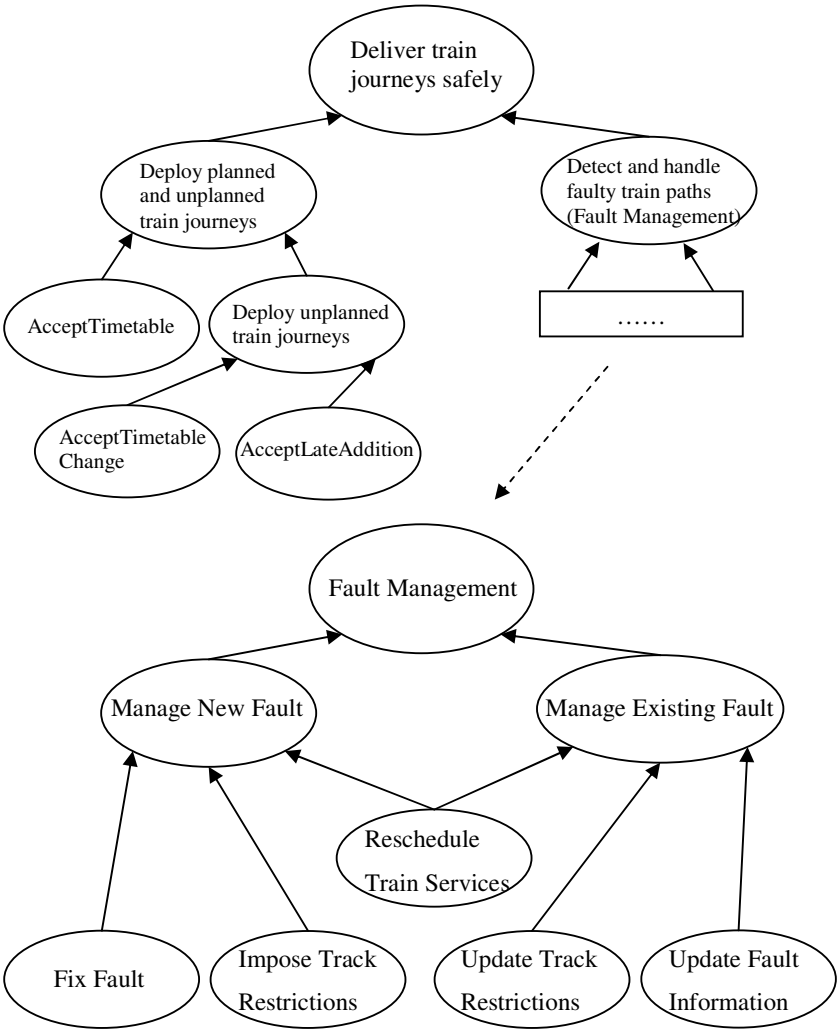

Figure 8.2: Goal-mapping graph for the overall system and excerpted case study

The top business goal, 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 of the case 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. The 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. In the process each sub-goal is decomposed into smaller sub-goals and considered just like an ordinary goal in the subsequent decomposition process. Finally, when the business goal is decomposed into the smallest granularity, the process terminates and

all the leaf nodes are presented as functional requirements tables. Among these is "*IMI-HandleFault*" as documented in Table 8.2. One table may contribute to the accomplishment of a sub-goal in a leaf node or, alternatively, a combination of such tables to a sub-goal in an intermediate node. Therefore such graphs can help the deduction of intermediate goals, provide a way to organise functional requirements, and check the completeness and validity of the original user requirements. For example, only with all the leaf nodes existing as requirements tables in the document and with the top nodes being fully supported by the bottom nodes can the business goals guaranteed to be represented in the requirements. Note that a business process can be delegated to a sub-goal at the graph end, such as "AcceptLateAddition" discussed in [1], and also to an intermediate goal, as a relatively complex case, such as "Manage New Fault" to be discussed later on, if its sub-goals are inter-related.

Before proceeding to the discussion of the agent-specific Reaction Rules and their formation of Business Process Rules towards business goals, the general cross-domain Business Concepts and Business Policies are described in relation to the case study, layering a foundation of the more complex knowledge structures.

8.5 Business Concepts for case study

In the case terminology section of the case study, Business Concepts and their semantic meanings are described. However, they are not always made explicit. Sometimes they have to be identified in case descriptions and functional requirements tables as shown in Table 8.2. Since they are the basic blocks for constructing PRs and RRs, and later mapping to Business Classes that facilitate the function of agents, it is important to discover all of them well before the modelling of other business elements. Otherwise, if some of them are missing, it may cause trouble in later phases. As it has been described in the implementation chapter of AAM, Business Concepts of the case study can be structured and stored in XML.

For example, identifying business concepts in the fault management domain using the grammatical analysis technique described previously, "fault", "incident", and "restriction" can be discovered. A "fault" has properties indicating its location, impact, and priority, these themselves being business concepts and properties of the business concept "fault".

i.e. *fault {type, location, impact, priority, description, cleared}*

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 that 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 {rail broken, London …}*

Represented in XML, "fault" and its related properties are stored as shown in Figure 8.3. It is this concept that being referred to by the PR shown in Figure 8.4 and RR shown in Figure 8.5. A corresponding Business Class "Fault" has later been implemented for agent IMI to operate on. Business Objects can be instantiated from the class. Once they are encoded in agent messages, agents can communicate, announcing among them the occurrence of faults and requesting the fixing of them. At that time, facts and business objects come into play, conforming to the schema of Conceptual Model and Business Classes.

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

      <!-- … more properties … -->

    <properties>
    <operations/>
  </concept>
```

Figure 8.3: Business concept "fault" representation for the case study

8.6 Policy Rules for case study

Unlike Reaction Rules, which are usually documented in functional requirements tables as shown in Table 8.2, Policy Rules are normally implicit in requirements specification, typically embedded in case descriptions. The underlined sentence in the case description read: *if the fault is located at capital cities, it has impact and needs to be fixed immediately*. Faults in the fault management system must be handled in accordance to the nature of the emergency. This rule falls into the first category of the PR classification discussed in Chapter 6, i.e. policies defined for classification of business objects. Figure 8.4 is 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.

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.impact = true
    </action>
    <priority>5</priority>
  </policy>
```

Figure 8.4: Policy Rule representation on fault impact for the case study

Falling into the second category of the PR classification, Rule2 (Figure 8.5) uses the term "*immediate impact*" as its condition and the same term is defined in Rule1 as its consequent action. It is therefore easy to infer that a fault located at capital cities has a high priority as a PR chain is formed.

Rule2.

If fault has "*immediate impact*"

Then it has a high priority

Figure 8.5: Policy Rule of the second category from the case study

Rule3 and Rule4 (Figure 8.6) fall into the third category of the PR classification, and are actually used in combination with and to contribute to a RR. The RR *IMI-HandleFault* in the scenario describes two means for railway asset fault handling in two different conditions, the details of which follow this section. Facts can be derived from them: Each incident has associated with it a fault which has "immediate impact".

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

Figure 8.6: Policy Rule of the third category from the case study

Unless all PRs are made explicit and represented properly, requirements elicitation is not completed. Tools [5] [7] with web interfaces as discussed in the previous chapter have been developed to facilitate viewing, addition and edition of

PRs. The combinational use of this PR and RR, working towards a common goal is described in Section 8.9.

8.7 Reaction Rules for case study

Reaction Rules are major requirements elements, being documented in functional requirements tables shown in Table 8.2 and contribute to business goals as shown in Figure 8.2. As it is discussed in the AAM modelling chapter and previous sections, domains are designated to respective agents responsible for realising domain functions. In the case study, a set of functions are required in the specification with regard to how the system works to manage faults. Three functions of special concern are reconstructed in Figure 8.7. They constrain business domains, 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, seen as an activity in the Activity Diagram (Figure 8.1), and documented in a functional requirements table in Table 8.2, describes constrained system behaviour. The keyword "is informed by" followed by the name of another function indicates the source of an event, and "inform" or "use" followed by the name of another function indicates the target of an action. In this sense, a Reaction Rule specifies the pre-conditions and post-conditions of agent behaviour. However, it is the intention here to make RRs separate modelling elements rather than attachments to other elements as in the UML. In addition, RRs should also be able to specify how events are processed and actions produced, so that a function and its constraints are fully integrated into one comprehensive modelling element.

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**.

Figure 8.7: Reconstructed specification for the case study

Figure 8.8 is the decision making tree that *IMI-HandleFault* uses to handle faults, derived from Figure 8.7.

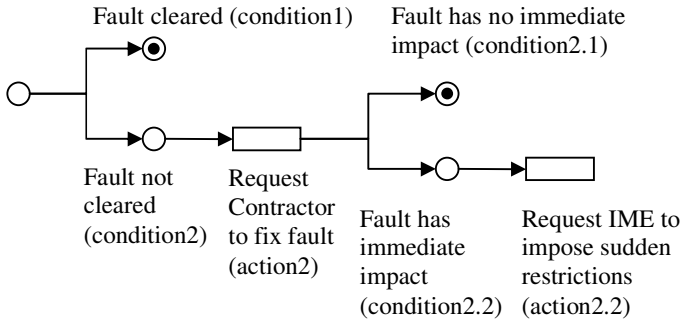

Figure 8.8: Decision making tree of *IMI-HandleFault*

This corresponds to the tree shown in Figure 6.7, in which a branch might be selected as condition2 → condition2.2 to request the fixing of fault and place track restrictions. If some additional conditions need to be considered following the request of IME for restrictions, the extended couplets could be {condition2.2.1, action2.2.1}, {condition2.2.2, action2.2.2}, and so on.

Agent IMI processes its RR *IMI-HandleFault* in the manner shown in Figure 8.9.

- 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.

Figure 8.9: Brief steps of RR *IMI-HandleFault* processing by agent IMI

A RR acts like a contract between agents. For example, *IMI-HandleFault*, as a fault handling RR in this fault management domain, may only respond if an event message with a pre-defined information structure representing an asset “fault” being received. In addition, it promises pre-defined information structures sent to the pre-agreed partners as defined by the RR. The following principles provide a straightforward guideline for transforming a requirement table to a RR in XML. Comprehensive formal definitions on this issue are detailed in [1].

- The “Cause” section is used to make the rule “event” – in this case a message reporting about a fault from IMI;
- The sections of “Information Used” and “Required Effect” are used to make the rule “processing” – in this case constructing relevant “asset” and “fault” business objects by using the reported information;
- The sections of “Required Effect” and “Outputs” are used to make the rule “condition” and “action” – in this case evaluating the clearance and impact of the fault and requesting Contractor and IME to fix fault or impose restrictions respectively, in separate messages.

Its XML specification conforming to the Reaction Rule schema of Figure 7.2 is shown in Figure 8.10.

```
- <reaction>
    <name>HandleFault</name>
    <business-process>Fault Management</business-process>
    <owner-agent>IMI</owner-agent>
  - <global-variable>
    - <var>
        <name>asset</name>
        <type>Asset</type>
      </var>
    - <var>
        <name>fault</name>
        <type>Fault</type>
      </var>
    </global-variable>
  - <event>
    - <message>
        <from>IMI.AcceptFaultReport</from>
      - <content>
          - <report>
              - <reporter>Henry</reporter>
```

```
            – <fault>
                <type>rail_broken</type>
                <location>London</location>
              – <asset>
                  <id>10015</id>
                  <type>rail</type>
                  <contractor>Contractor_A</contractor>
                    …
                </asset>
              </fault>
            </report>
          </content>
        </message>
    </event>
    <processing>
       asset = new Asset (reportMsg)
       fault = new Fault (reportMsg)
    </processing>

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

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

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

Figure 8.10: Reaction Rule *IMI-HandleFault* specification for the case study

The definition of rule *IMI-HandleFault* in XML is based on its Reaction Rule Model transformed from the functional requirements table, and becomes executable to agents. A detailed ten-step process for a generic RR in XML has been described in Figure 7.3 and illustrated specifically in Figure 8.9. To explain its execution, suppose all previous rules managed by the agent IMI are not applicable and now it checks the applicability of this rule. The agent parses the message just received, and finds out that the message has come from another RR called "AcceptFaultReport"/"NoticeFault" of the same agent. This agent knows its rule

HandleFault is defined to deal with such a message, because according to the rule definition, the content of XML element <event>/<message>/<from> matches with that RR in the name. Also the message content has a fault structure, the same structure as specified in the rule: In the rule definition shown in Figure 8.10, the <message>/<content> contains a fault structure, where the rule specifies it will only accept events with that kind of content structure (Step1 in Figure 8.9). Then, by invoking business classes managed by the agent, a business object of "Fault" can be constructed, the name of which is declared and shared as a global variable and its clearance and impact can be evaluated. The "Fault" object can be built from the content of the <event>/<message>/<content> structure of the received message. The asset where the fault is located has been constructed as a business object during the rule processing as well (Step2 in Figure 8.9). Its clearance can be checked using the "cleared()" method of the business class. If it is evaluated as FALSE then the condition2 in Figure 8.8 for executing the rule is satisfied. A corresponding message will then be structured using the created "Fault" object, the name of which refers to the global variable, and sent to the agent Contractor, again as it is specified in <action>/<message>/<to> of action2, to request its fixing of the fault (Step3&4-couplet1 in Figure 8.9). Its impact can be checked using the "immeImpact()" method of the same business class. If it is evaluated as TRUE then the condition2.2 in Figure 8.8 for executing the rule is satisfied. A corresponding message will then be structured using the created "Asset" object, the name of which refers to the global variable, and sent to the agent IME, again as it is specified in <action>/<message>/<to> of action2.2, to request its imposing restrictions on rail track (Step3&4-couplet2 in Figure 8.9). Finally, the IMI agent adds the knowledge that a fault has occurred at this time with a potential associated incident to its own beliefs (Step5 in Figure 8.9). When enough of such information is collected, business reports can be built for analysis for later use. In the whole rule execution process, if the event does not match or the condition is not satisfied, the next candidate rule with the highest priority will be tested for applicability and executed in the same way.

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. An example among others is that the "ValidateTrainPlan" requirement is a sub-requirement of the "AcceptLateAddition" requirement [1]. 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 AAM modelling chapter.

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 below, the basis for building Business Process Rules described in the next section. Figure 8.11 and Figure 8.12 are developed from the two Design Model templates of Figure 6.9 and Figure 6.10, applying the scenario described around the RR *IMI-HandleFault* in Figure 8.7 and its processing steps in Figure 8.8 and Figure 8.9. The XML definition of the RR in Figure 8.10 is associated with the modelling element in Figure 6.9 and Figure 6.10. This provides the agent a means to carry out computation using the models, and at the same time offers people visual presentation and a method to modify models.

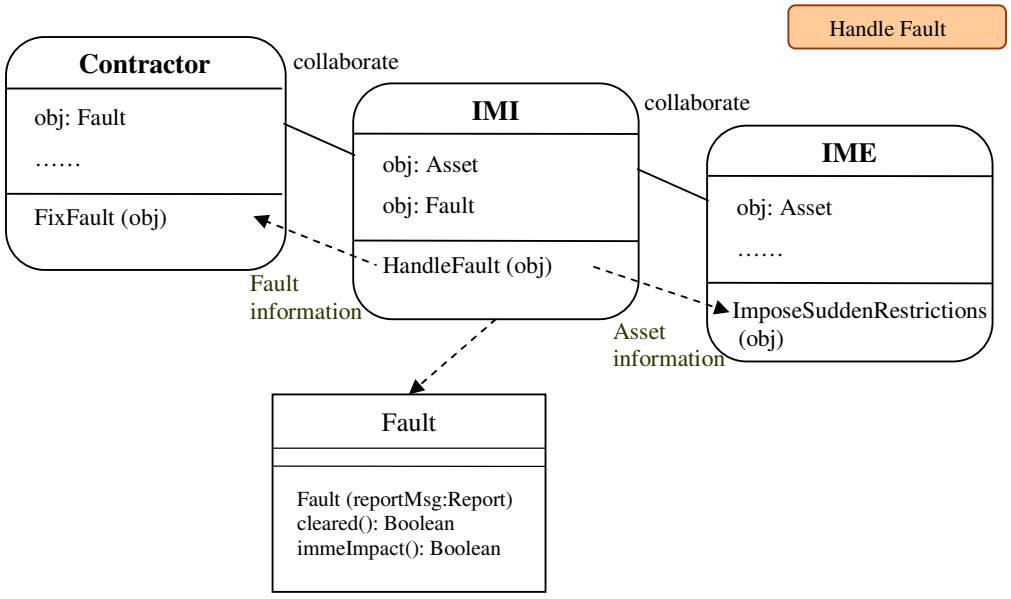

Figure 8.11: Structural Model for the case study centred on *IMI-HandleFault*

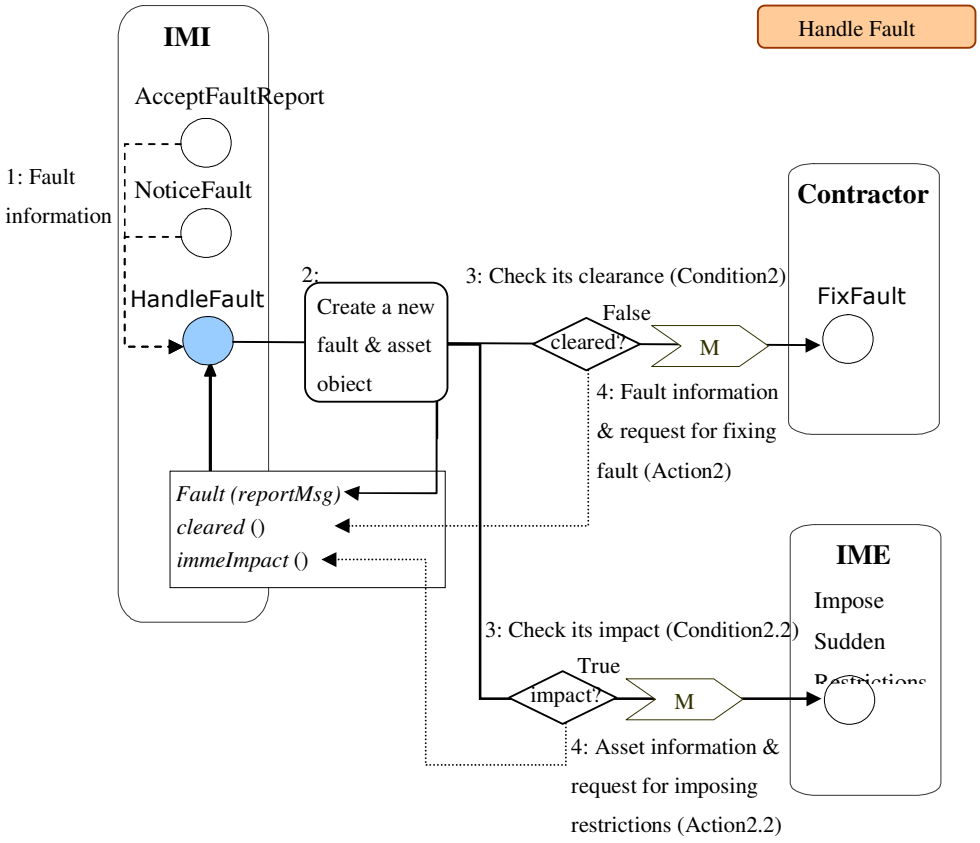

Figure 8.12: Behavioural Model for the case study centred on *IMI-HandleFault*

8.8 Business Process Rules for case study

In Figure 8.7, agents of IMI, IME, and TRI have been identified, with the three RR that belong to them contributing to the goal of managing faults. Two aspects are involved towards this goal: a) new faults are reported and then handled and b) existing faults are handled in an alternative way or eventually removed as information about them is updated. *IMI-HandleFault* is responsible for new faults handling and *IMI-UpdateFaultInformation* is responsible for handling the updating of faults. Therefore two BPRs working in parallel are required to be deployed for a shared goal. Many RRs together contribute to each of the BPRs. However both use a shared RR of *TRI-RespondToIncident*, requesting a rescheduling of train services.

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”, with an intention to handle new faults. This BPR is shown in Figure 8.13, with only the default conditions considered and assumed to be true for simplification. This is one of the two sub-goals of the top level goal “Fault Management”, as a result

of the goal decomposition shown in Figure 8.2. It extends the two Design Models shown in the previous section, connecting all related agents and their RRs that collaborate towards a shared goal.

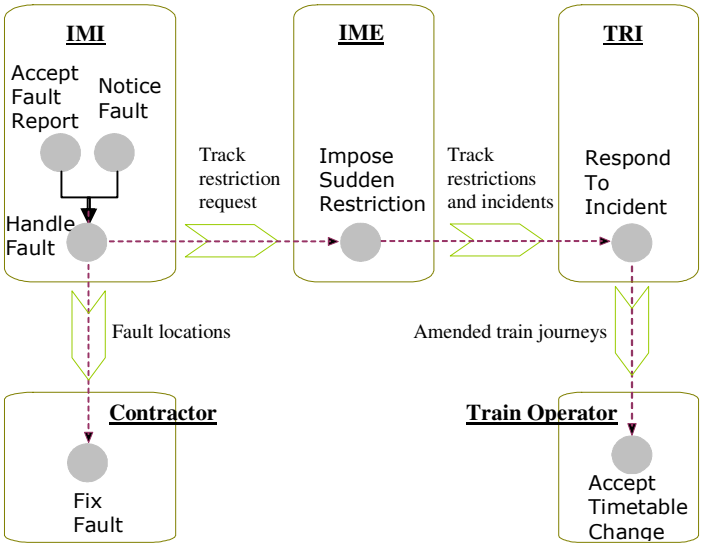

Figure 8.13: Business Process Rule "Manage New Fault" for the case study

The formation of BPR is by matching action and event pairs of interconnected RRs as described before. As shown in Figure 8.14, 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.

<<Business Process Rule>> **Manage New Fault**
businessGoal: a new fault is managed
initialAgents: IMI finalAgents: Contractor, Train Operator cause: a new fault is reported requiredEffect: a Contractor will fix the fault, Train Operator will re-schedule train services

Figure 8.14: Business Process Rule runtime instance

Figure 8.15 shows the BPR "Manage New Fault" in XML, expecting a new fault as input, and "fault fixed" and "train service re-scheduled" as results.

```
- <process>
    <name>Manage New Fault</name>
    <goal>a new fault is managed</goal>
  - <IAs>
```

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

Figure 8.15: XML representation of the BPR "Manage New Fault"

Based on the UML profile of AAM (see Figure 6.15), Figure 8.16 demonstrates that a contract is written between *IMI-NoticeFault/IMI-AcceptFaultReport*, *IMI-HandleFault*, *Contractor-FixFault*, and *IME-ImposeSuddenRestrictions*, stating that *IMI-HandleFault* is responsible for events caused by the former two RRs, and expected to generate events for the latter two RRs. For example, *IME-ImposeSuddenRestrictions* expects a message from *IMI-HandleFault* and once it receives the message it imposes sudden restrictions. That rule has its event specified to react to such an event and uses the asset information encoded in the message to impose restrictions. The overall behaviour of these contributes to the Fault Management process.

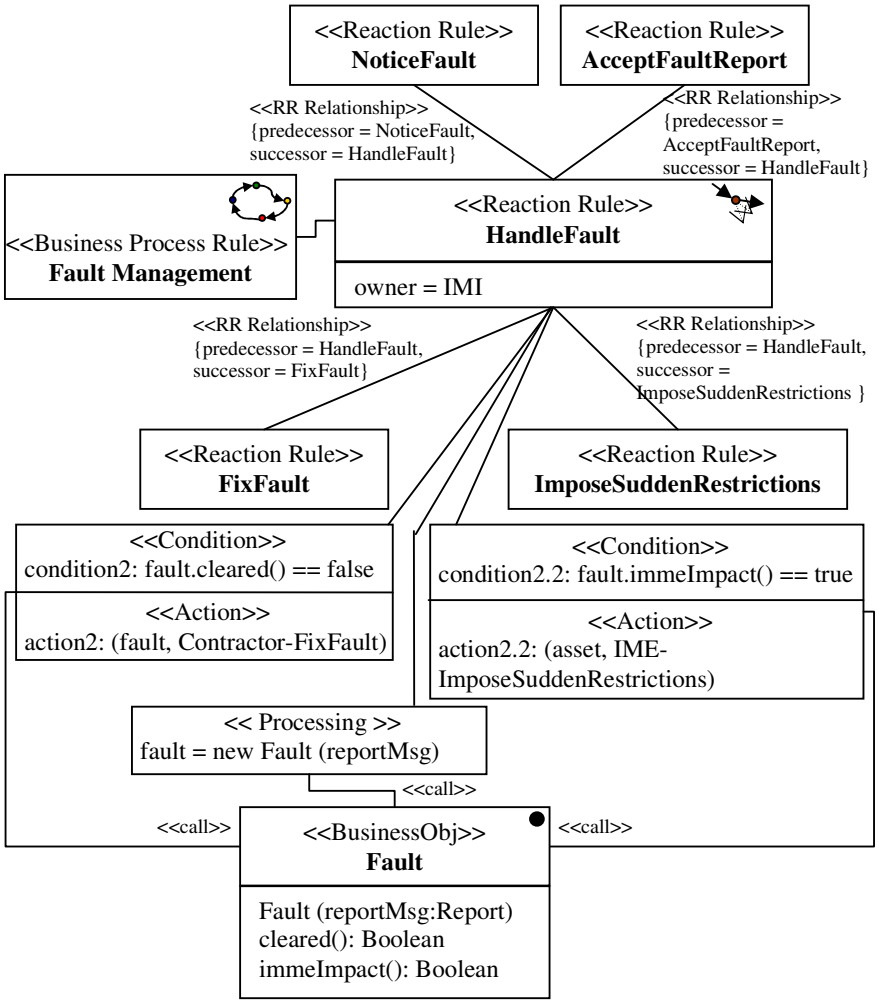

Figure 8.16: Runtime model centred on *IMI-HandleFault*

8.9 Business models working together in AAM

Sometimes RRs are not fully functional without being supplied additional knowledge to reason and make decisions. Equally PRs cannot work independently without a context to be applied in. They do complement each other in the big picture of BPRs. In these three levels the models work together to realise business goals.

BPRs are business processes initialised by the IA in the beginning, causing a series of agents to react using various RRs, and the process is finished at the FA. In the case of RR, an agent chooses a RR to react to after an event 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 must be enforced in that context.

A typical sub-process that makes up a BPR is shown in Figure 8.17. Conceptual Model, Fact Model, PR Model, and RR Model are in coordination as the knowledgebase of an ordinary agent when it is to behave, assisted by the FMA and the PRMA. The RR Model describing the agent execution of RR is integrated with others overall.

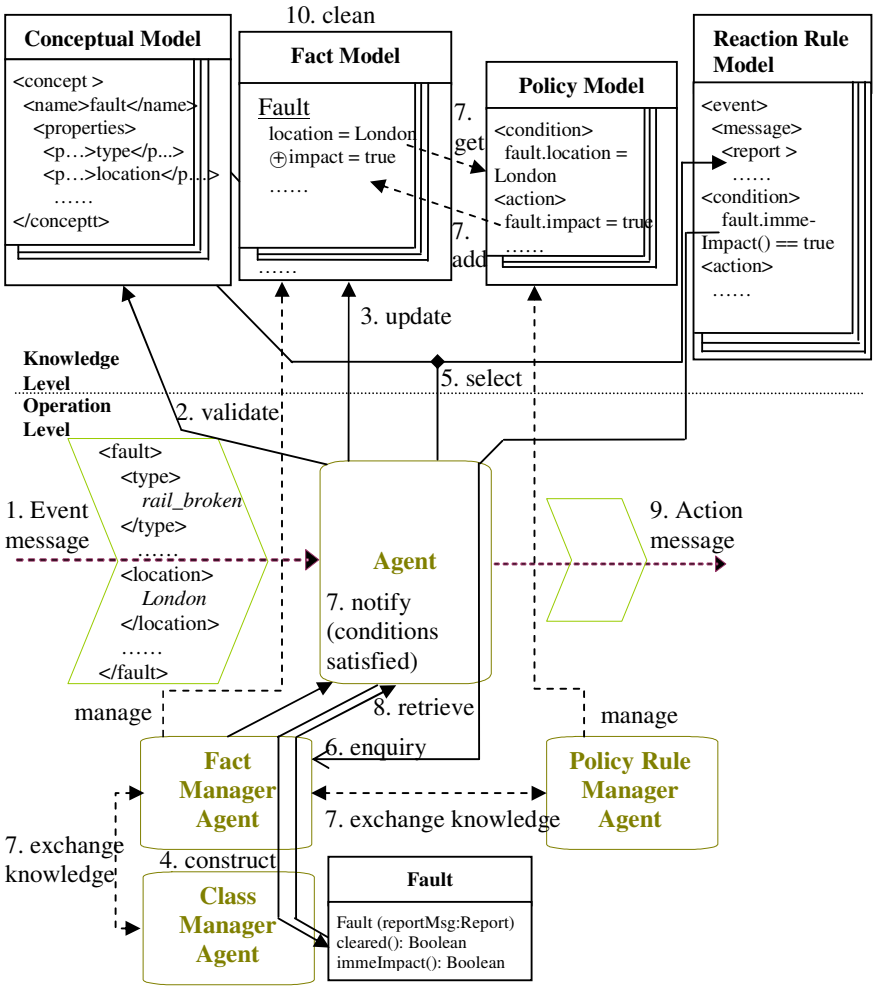

Figure 8.17: AAM runtime architecture

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 described in the abstract Agent Model described in Section 7.3. The application of it to the case study in that concrete situation is discussed below referring to Figure 8.17.

- 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.

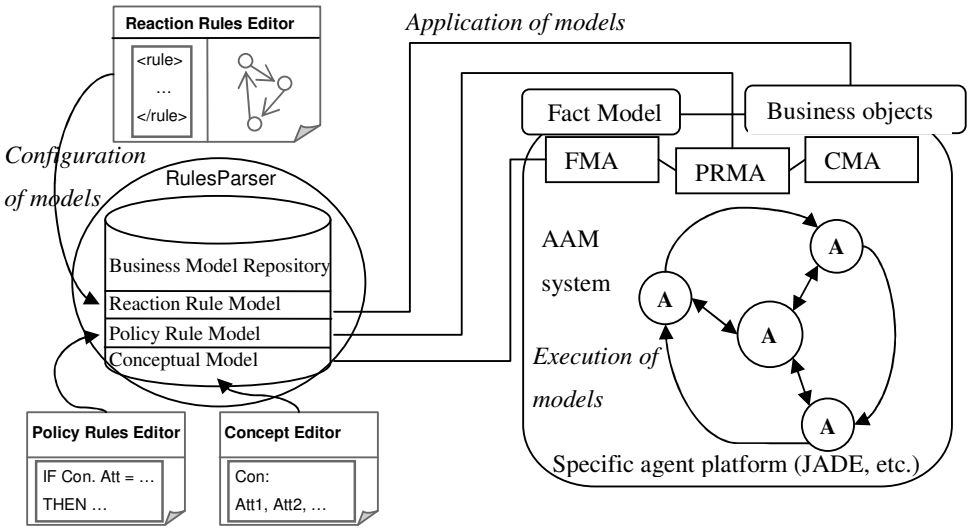

Figure 8.18: AAM software architecture

Figure 8.18 illustrates the overall AAM architecture. In summary, business goals, business processes, business rules and business concepts are modelled in AAM. The transformation from business requirements in natural language to AAM models has two major implications. On one hand, the established business knowledge models are real business assets that business customers keep maintaining. They are visualised and the requirements validation at the model level becomes easier. These have direct business representations and truly exist in the business world. They are comparable to models built in i* or Gaia which concentrate at the requirements or design level modelling. Focusing on strategic actor dependency or role modelling, they are not as straightforward or understandable as the AAM models. On the other hand, models built in AAM have behavioural semantics associated with model constructs so that agents can translate their behaviour from models. Therefore AAM covers the complete MAS development process, filling the gap nowadays seen between major agent-oriented methodology like i* or Gaia (requirements and design modelling) and agent-oriented development platforms such as JADE (implementation). In addition, AAM also adds semantics to agent-oriented modelling language. This is in contrast with UML or AUML, where the notation system has to be interpreted by human beings rather than agents to direct system behaviour. Consequently, it can be concluded that the AAM approach is practically more useful than existing agent-oriented methodologies, modelling languages, or development platforms.

Chapter 9: AAM Maintenance & Adaptation

9.1 Introduction

In the previous chapters the AAM approach, the requirements & design modelling, implementation, and an illustrative case study have been described. The advantages of AAM are through its easy adaptation brought to the system. In this chapter these are discussed in detail.

Briefly, the business models of AAM are deployed [1] [2] using a central XML repository, as opposed to implemented code. Tools are developed to support the addition/amendment/removal of business elements in models by business experts. Business models are interpreted by agents on the fly. Once models get (re-)configured, agents interpret them and behaviour is adapted at runtime. Concerns about business decisions are hence separated from concerns about the underlying IT infrastructure. Whenever business people change their mind about their business needs, the changes are made to the business models and reflected automatically in the corresponding information system. This eases the maintenance and greatly reduces the risk and effort that could be required by traditional approaches.

A business model hierarchy has been established: business concept – policy rule – reaction rule – business process, one layer supporting the next. Also changes at any layer spreads to layers above them automatically and take effect without the need of human intervention. The change of BCs is live in all business rules that refer to them, immediately. As concepts are put in the central repository separately from rules that refer to them, rules weave them together only at the time agents request the use of them. The other layers co-work in similar patterns. The change to a PR is reflected automatically in a RR, since RRs dynamically choose PRs and form PR chains at the time agents requests the use of them. The currently selected PR can bring to RR its changing policy immediately it is applied. The changes to RRs are reflected automatically in BPRs, as they are dynamically composed in an interaction pattern of collaborative agents, specified by RRs collectively. Change at any layer of the business model needs only be changed and is automatically available in all higher layers. The auto-composition and dynamic application of some business model elements by others hierarchically leads to adaptivity.

Referring to the case study, the RR *ImposeSuddenRestrictions*, for example, is involved in two BPRs for managing new faults and for updating existing faults. Once it gets changed, both BPRs start to impose this change in re-scheduling train services from that moment on without any manual change to the BPRs themselves. For each BPR, as long as its input, output, and goal are unchanged, involved intermediate agents are free from participation and can be removed. Agents are activated and rules selected as required by business needs, determined at the time of execution.

A RR is a central modelling element for adaptivity as described in Section 9.2. A set of PR make up an encapsulated RR as described in Section 9.1. A BPR made up of a set of RR can be encapsulated as a higher level RR as described in Section 9.3. Such free promotion and demotion among model elements is due to the high uniformity and built-in encapsulation of AAM's hierarchical layers.

The modularised business model separates the concerns of change in the aspects of concept, policy, reaction and process. Also AAM allows changes in different scopes: individual entity scope in the case of reaction patterns; system scope among global concepts and policies; cross-system scope of business processes from many organisations for interoperating. Thus the maintenance in different scopes can be achieved. In this section the level of adaptivity achieved in the different scopes, one contributing to the next, is detailed in terms of AAM's hierarchical layers. The impact of additional concepts to the application of policies has been discussed in the AAM implementation chapter (Chapter 7), i.e. the addition of a property of "credit" to a concept of "customer" and a related policy defined on the discount a customer can enjoy according to their credit has impact on discount policy rule application. This has also been discussed in [5] [7].

9.2 Adaptivity of Policy Rules

PRs representing global business policies reflect actions/facts that all agents are obliged to perform/believe if conditions of the policies are satisfied. They may form rule chains, contributing to the knowledge that RRs need in decision making. A simplified form of the PR is:

IF conditions **THEN** actions

One of such constraints can reflect a business policy. Assume that the previous rail track system now extends to provide customer service. Included in the extended system there is an online ticket sale system, and the rail track company has the

following policies for selling tickets to customers. For simplification, assume no logical inference relationship is among them.

Policy Rule 1: Group Travel

IF *A minimum of 10 people in a group*

THEN *discount 25% off standard fares, 10% discount on First Class*

Policy Rule 2: Off-peak Travelcard

IF *ticket 8:30 p.m. or later from London on weekdays, any time during weekends*

THEN *15% discount*

Policy Rule 3: Special Offer of today

IF *ticket for First Class today*

THEN *5% discount*

Suppose multiple discount offers can be applied jointly, and then when a checkout event occurs, PRMA selects all appropriate PRs to apply. The customer order information, as well as the original ticket price is used for the computation of the final price after offer. This process is modelled in Figure 9.1. Because the pre-condition of each PR shown above is satisfied in this case, they will all be applied and contribute a reduction to the total price, the final result is the post-condition of the last PR: £20 * 12 * (1-10%) * (1-15%) * (1-5%), which will be sent to interesting parties as a consequent action. Five elements collectively can be regarded as an integrated RR with no decision making branches, where the three PRs compose <processing> component, while a separate request receiving event and a result despatch action at each end composes its <event> and <action> components.

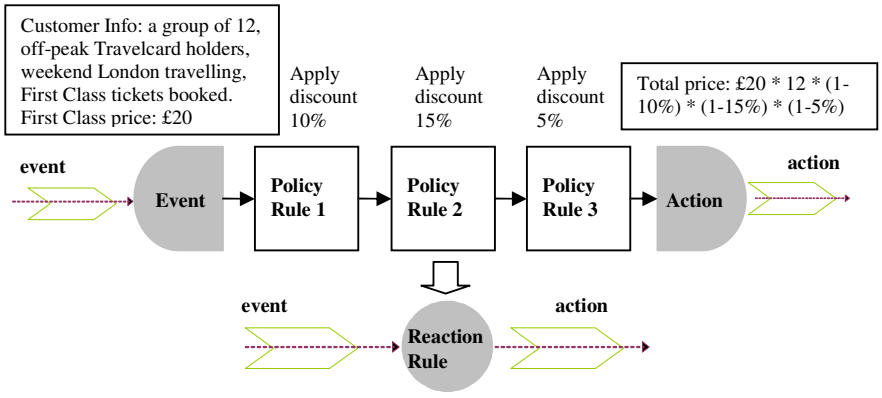

Figure 9.1: An Event, three PRs, and an Action constitutes a Reaction Rule (PR escalates to RR)

RRs applicable in local scope to individual agents and PR applicable in global scope across agents share the couplet structure of {condition, action}, making their seamless integration possible. If one day it is decided that the three example PRs

need to be localised, which means that only one dedicated agent should access them to calculate the discount value, then a RR can be formed to replace the three PRs. Conversely, it is possible to globalise a RR as multiple PRs if such policies captured by it need to be available to many agents. This mechanism provides a means to freely convert rules from being private to being public, assignable to one or many agents.

PRs let the system be adaptive to changing business policies. The addition/removal/modification of one PR has no effect on others, easy edition tools [5] [7] enabling business people to put their current policies into effect immediately. There is no need to re-build the system, because rules are not embedded in code. Instead agents will flexibly choose from them the appropriate and most up-to-date ones in that runtime context. The goal is to let systems reflect current business policies without re-development. This transfers the application of policies from humans to agents that automatically select and execute, thus eliminating the possibility of omitting to apply any policy. For example, the last PR on a special offer of today will be automatically picked up by the PRMA to execute. On the next day, this will not happen, as it becomes invalid due to the unsatisfied pre-condition of the date of application. An alternative way of customising the application of PRs is by setting different priorities for them, and let higher priority policies override the lower priority ones [5].

Change of the post-condition of a PR changes business effect, applied by agents who get the changed rules in a particular context. It is possible that such changes pass on and impact agents on the selection of subsequent rules as various pre-conditions of other PR may be satisfied or not as a result of the change of that post-condition. An algorithm in support of automatic rule chain formation by picking appropriate policies automatically in changing context and conditions follows in Figure 9.2.

```
Meta-rule RuleExecution (Agent a, eventRule eR, policyRules pR, actionRule aR) {
   RuleSet availableRuleSet = pR;
   // the initial state is the state when the eventRule is just executed
   Condition postCondition = eR. postCondition ();
   // assume the next rule after the last available one in the rule set is the first one in that set
   Rule nextRule = availableRuleSet. nextRule ();
   int invalidRules = 0;
   // check the validity of all policy rules in turn until every valid rule is executed, successive failed application of
all remained rules indicates their inappropriateness
   while (nextRule != null & invalidRules < availableRuleSet. size ()) {
      if (postCondition ≥ nextRule. preCondition ()) {
         // execute the valid rule just found, update the state, and the available policy rule set
         a. executeRule (nextRule);
         postCondition += nextRule. postCondition ();
         availableRuleSet. removeRule (nextRule);
```

```
        invalidRules = 0;
      }
      else {
        invalidRules ++;
      }
      nextRule = availableRuleSet. nextRule ();
    }
    // finally execute the actionRule if it is satisfied
    if (postCondition ≥ actionRule. preCondition ()) {
      a. executeRule (actionRule);
    }
}
```

Figure 9.2: Policy Rule chain formation algorithm

9.3 Adaptivity of Reaction Rule

Using Agent IMI and IME from the case study, it will be illustrated how adaptivity is achieved through the AAM use of a RR.

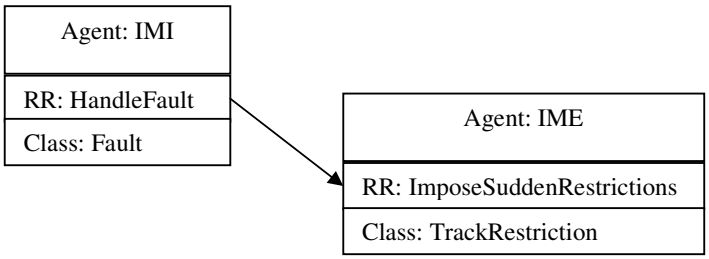

Figure 9.3: Agent, rule, and class in relationship but not actually in the presented way

Figure 9.3 shows the relationship between agents IMI and IME and their collaboration in the Fault Management scenario. IMI uses *HandleFault* to collaborate with IME, which responds using *ImposeSuddenRestrictions*. Two business classes are invoked during the function of two RR respectively in assistance of event processing and decision making. This scheme is part of the proposed ARC framework [1] [4], in which the hierarchy of Agent-Rule-Class is established. This might cause the illusion that the relationships between agents, the selection of RR, and the selection of business classes are fixed. In fact, there is no direct link between agents, or between agents and classes. Rather, such collaboration or association relationships are specified in the selected RR. In other words, it is at the time that a RR is selected for reaction to an event that an agent knows its collaborative agents and supporting classes. The configuration of two RRs can change, not only the collaboration relationship between IMI and IME, but also the use of "Fault" and "TrackRestriction" classes. Such information is completely transparent to agents and only known to them at the time of their activation by events. Thus, the agents and

classes in relation with each particular agent are replaceable at runtime by configuring RRs, with immediate effect. The accurate relationships among these modelling elements in the chosen part of the case study are shown in Figure 9.4. This shows the dynamic collaboration between agent-agent and agent-class as described at the end of Chapter 6.

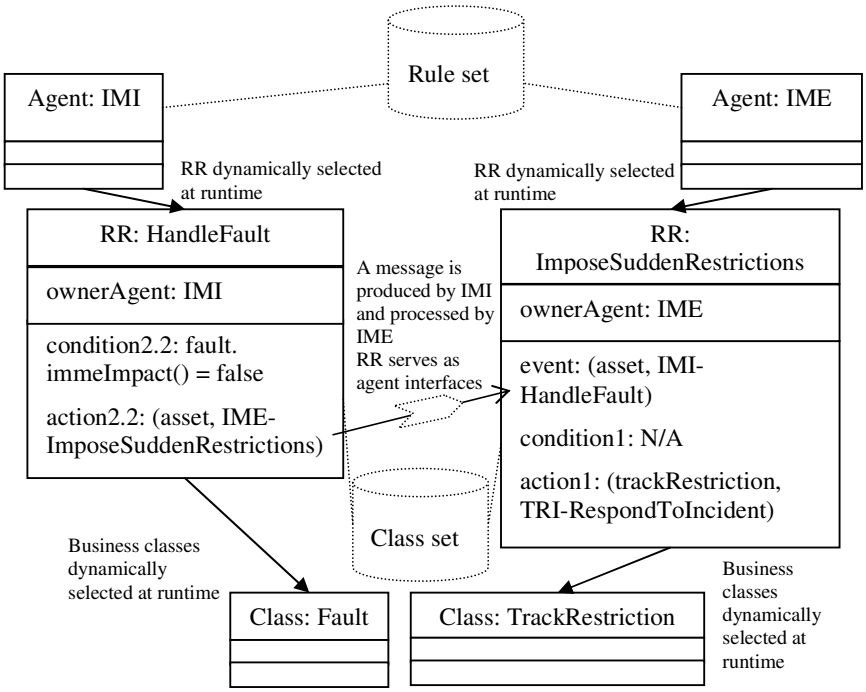

Figure 9.4: The actual relationships among agent, rule, and class

Adaptivity is thus demonstrated as achievable in different levels by reconfiguring business models. Not only agents, classes and methods are chosen dynamically, but also the link among these model elements are established dynamically, and interpreted by agents thereafter.

Connections among collaborative agents in business processes are decided at runtime and thus the enactment of business processes, as BPR composition depends on decision making at a set of successive RRs chosen one after another. A connection is required if and only if an action of one RR points to an event of another RR, because of which a partnership is established between the two agents that own the two RRs. The reconfiguration of {condition, action} pairs leads to a different partnership. This also enables the re-establishment of decision making tree, as conditions and corresponding actions have been re-built.

Figure 9.5 shows various aspects of adaptivity that RRs can help to achieve, demonstrated through it compositional parts.

1. Partnership adaptivity. In the figure, agent1 adapts its partnership from being between agent2 and agent3 as a new action is chosen. The goal of the business process is realised though the process is composed differently. A different outcome of the business process might be expected but the result serves the same aim of setting up the process in business.

2. PR application adaptivity. The Policy Rules that get applied by the PRMA are decided at runtime: a sequence of PR is established when their pre-conditions are satisfied sequentially, in the same order that PRs are formed. The re-configuration of PR may cause a different sequence of PR to execute at runtime. In the same figure, agent1 adapts its PR sequence using PR2a as a replacement for PR2b as a result of the changed PR2a/b pre-conditions or PR1 post-condition.

3 & 4. Class application adaptivity and method application adaptivity. Which Business Concepts are used is decided at the time when the business rules referring to them are executed. Correspondingly, the business classes and their methods are also determined at that point. The re-configuration of business rules in the use of new business concepts leads to the use of different business classes. In the same figure, agent1 adapts its use of ClassA1 instead of ClassA2, and a new method from ClassB comes into use as a new corresponding business concept is accommodated. Such an adaptive mechanism allows, for example, agents to be able to cope with extendable types of events, since a RR can be configured to use new types of class or new methods in alternative versions of a class.

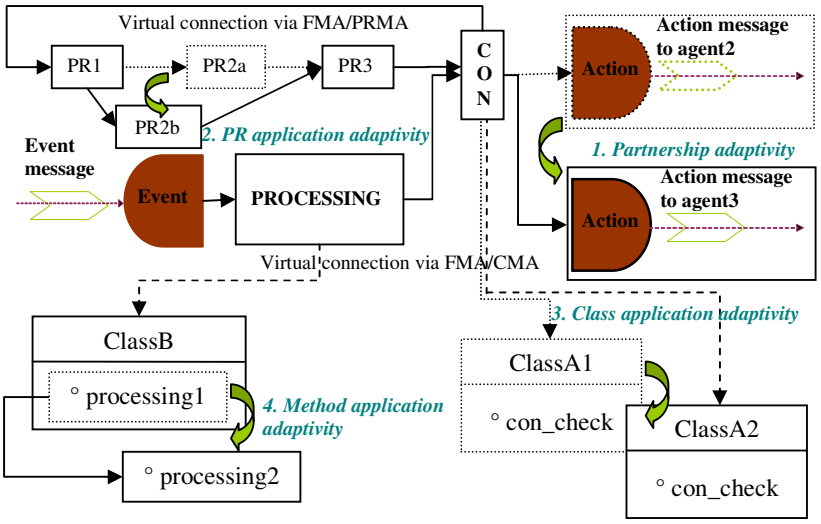

Figure 9.5: A Reaction Rule (with components split up from its original notation) owned by agent1 and its adaptivity in four aspects

A greater challenge confronted by software engineers today is that software behaviour must adapt in accordance with unforeseen contexts. For example, a new type of event triggers the use of a set of methods in a sequence previously unseen and hence the formation of a new business process built upon new agent interactions which handle the emergent event type. This might mean that a new business service comes into being as required in the real world. Comprehensive adaptivity may be required involving all the aspects of adaptation mentioned above. Despite the complexity it seems to demand for the handling of such an emergent request, the actual adaptation process is straightforward by using the adaptation mechanisms illustrated by Figure 9.5. Briefly, business people can define at the business level a set of new Reaction Rules interconnected in a BPR. The first RR deals with the initial new event type. Class methods are selected for invocation towards a goal. Appropriate agents are assigned with RR that fit into their domain responsibilities. This grants them the capability to solve additional requirements at runtime as the configuration is carried out without intervening with the running system. Even better would be that an intelligent agent can choose class methods as strategies, design new RR & BPR, look for responsible agents for the additional tasks, and assign rules to agents when it is informed of a new requirement. Such further development would strengthen AAM and grant it greater adaptivity.

9.4 Adaptivity of Business Process Rule

A schematic BPR made up of multiple RRs is illustrated in Figure 9.6. Here a business goal is realised by the performance of two initial RRs, five intermediate RRs (RR1.1, 2.1, 2.2, 2.3, 3.3.1) and four final RRs. Some computation results could be returned to an initial agent, assuming that RR2 and RR4 belong to the same agent. In that situation, this BPR can be seen as though an agent uses RR2 to initialise it, with the assistance of other agents, it is finalised with some results returned back to the same agent (and also possibly some others) and received by its RR4. If a BPR constituted by RRs is symbolised as a tree, then the initial RR are its roots and the final RR its leaves. Some of the leaves go back to where the roots grow, but not directly to the roots (initial and final RRs not being the same ones but being owned by the same agents). The tree has many branches going from the roots to leaves, the fact that all leaves can be reached through these branches indicate the accomplishment of the BPR, or the realisation of the business goal of the BPR. Unless all of the leaf nodes in the tree structure are completed, the goal is not realised.

Figure 9.6 also shows the encapsulation of the BPR as a single RR to external entities. Events of the RR, that being what the BPR is regarded as from outside, are triggered by the IA, and actions generated by the RR are received by the FA. Intermediate processing is encapsulated and can be dynamically determined at runtime, just like the application of a PR chain internal to a RR. As it has been discussed in Section 9.2 and Figure 9.1, multiple PRs can be promoted to a single RR as a result of localisation and a single RR can be demoted to multiple PRs as a result of globalisation. The same applies here. Since BPRs and PRs are both applicable in global scope, they may be made private as RRs and assigned to dedicated agents, if necessary, to narrow their application scope. Conversely, since RRs are only applicable in local scope, they may be decomposed into multiple PRs or RRs and so possibly assigned to multiple agents (in the case of BPR) or none but accessible by any agent (in the case of PR), if necessary to enlarge their application scope. Overall, the AAM's business model hierarchy of PR-RR-BPR is in unification and one layer can be upgraded or downgraded to another as required.

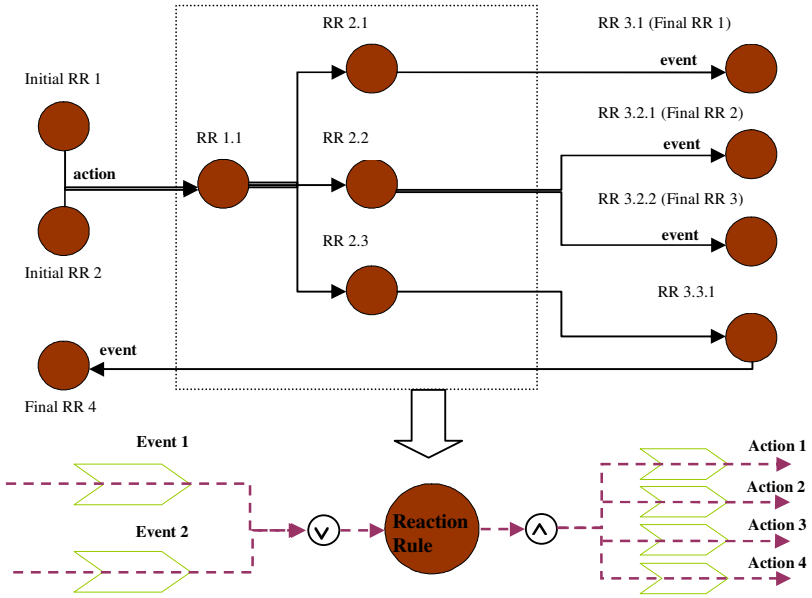

Figure 9.6: A BPR tree structure (with OR & AND notation) becomes a RR (i.e. a BPR is encapsulated into a RR)

9.5 Adaptivity of inter-BPR (across systems)

Systems built using AAM are interoperable, even though various agent running platforms are deployed in different systems, and components that agents act upon may be written in very different languages. XML messages, the exchange of which is the only means of communication among agents, are readable and have the unique meaning across networks, operating systems, and application platforms. In the case where several systems need to collaborate [10], several business processes must join together and form an integrated super business process, reflecting a partnership among several business organisations. Agents from all these systems participate in this integrated business process for a common business goal. If the shared goal is required to change, AAM instances running in multiple systems adapt and coordinate towards an overall adaptation goal. The existing business processes should change as little as possible for their participation in cross-system functions, as they must maintain the fulfilment of their internal business goals.

Figure 9.6 has shown the encapsulation of a Business Process Rule as an assembled Reaction Rule: Event 1 OR Event2 –> Processing –> Action 1 & Action 2 & Action 3 & Action 4. This mechanism enables the hiding of internal processing details of BPRs and exposes only the necessary interfaces required to request BPRs to function or retrieve outcome from BPRs. Thus among collaborative systems, the event and action messages of the RR corresponding to the BPR are the only required

information to be made known. The internal processes of each system need not to be public to external systems and independence is maintained intact for serving additional joint goals. OMG's IDL adopts a similar mechanism to enable interoperation but the AAM's use of XML-based exchange in this area is an advance and more natural to understand, easier to implement, and better supported by existing mainstream infrastructures.

To illustrate using the case study, suppose rail track faults could be detected from an external system and on detection of faults this system requests the system to resolve the fault. This interoperation is as simple as the addition of a message sending from that system to a responsible agent in the current system. The agent responsible for receiving reports to handle faults and the message format it expects to receive for that purpose should be published, so that this agent can be known to agents from all external systems and a message can be assembled and sent to it at runtime for the request of fixing faults. No matter what agent platform or supporting component language has been used by the external system, also built under the AAM framework, the XML-based information publication and exchange is commonly shared and understandable by the two collaborative parties to enable their communication. The sample published information structure for the case study is as follows.

ES1, an external system needs to know the following to collaborate with the current system.
Business goal: "Fault Management"
Event message format: "fault_report" (<report> ……</report>) – expected input XML schema
Send to: Reaction Rule: "*HandleFault*"@Agent: IMI
Action message format: "fault_in_processing" (<result>request_accepted</result>) – expected output XML (schema), if a result is required to be returned
---
The current system needs to do the following to collaborate with the external system ES1.
Permit rule "*HandleFault*" to respond message from external system ES1 as well as existing pre-defined partners

As long as events match, cross-system collaboration becomes possible and automatic, simply by granting permission to other systems. Eventually a collection of business processes could be distributed across systems, available to be triggered by internal or external agents for dynamically joining business goals. Dedicated BPRs can be registered as services, each serving one aspect of concerns. Therefore,

systems can interoperate to address concerns using services provided by looking up the registration repository for the right BPRs at runtime. Concerns are separated, and systems collaborate for providing their capability and request as needed. Maintenance in a global level is alleviated as each shared service is maintained locally. Once the new functionalities become available in one system, they get used and the results live in other systems automatically. Agents with interoperability in AAM are in contrast with agents proposed in other approaches, for example ME agents [10] (see Section 3.7), which can only interoperate but have no adaptivity.

In addition to easier services sharing across systems, a further advantage of encapsulating BPRs (as services) for easier management across systems is very important. For example, public BPRs and private BPRs can be distinguished, and associated with ownership. Public BPRs are those visible across business organisations while private BPRs are those only visible to the organisations that own them. Since no organisation wants to expose all its internal processing details, they can only publish the relevant ones as public and the interfaces to use them to enable potential collaboration. The confidential information can be set in the private ones. This leads to two other related benefits. i) The change of private BPRs has no ripple effect to external partners who use the public BPRs, even though the public BPRs may invoke the private ones. This makes inter-organisation dependencies more reliable with less intrusive IT interventions. ii) The change of business partners has no effect to a client if both the original and the new partner provide their services according to a pre-agreed BPR interfaces. This makes service provision more flexible to clients since services can be better matched to their needs and available service providers offer alternatives.

In many cases BPRs could be reused across systems. Even in the case where they have to adapt to adopt additional components from external systems, amendment of an existing business process to allow the collaboration of multiple systems is easy by (re-)configuration. For example, before the provision of a special service or year round offer of train service to a customer, a customer credit check, only available from another system, may have to be ensured. Simple re-configuration of relevant RR, as shown in Figure 9.7, lets RR3 from System2 participate in a BPR in System1.

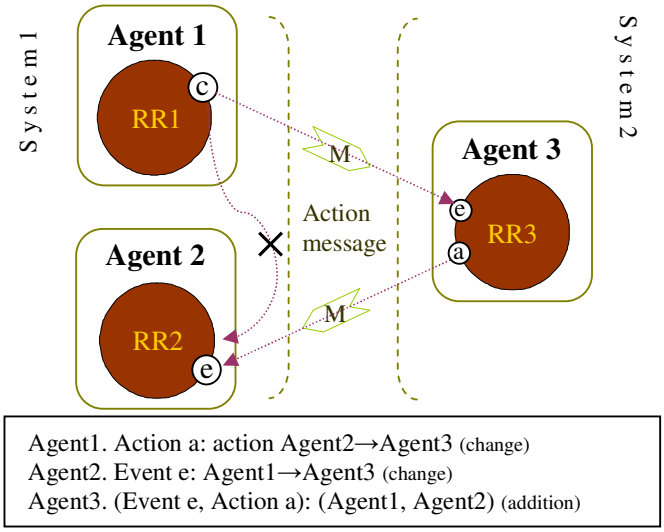

Figure 9.7: Add a RR node from an external system to an existing BPR

Obviously, collaborating systems in different enterprises/organisations should share common goals but need not be required to have the same infrastructure. AAM reflects this in that it specifies a meta-model at the knowledge level suitable for any information system, but entails no specific constraint at the operation level. Agents in different systems share their knowledge and capabilities, and collaborate for common business goals by message passing. It is the universal syntax and semantics of the messages that brings interoperability and it is the configurable interaction pattern among different systems that brings adaptivity.

9.6 Overall adaptivity evaluation

Various types of adaptivity have been presented in previous sections. 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. AAM can be used to minimise interdependencies among system elements. Table 9.1 compares the impact of each single type of typical required change on overall systems developed using AAM and traditional OO systems. The table relates to the GQM graph described in Figure 1.2 which breaks down the software adaptivity goal of AAM and tries to answer various questions. The table compares AAM with existing approaches in respect of adaptivity. Considering this table together with Table 2.1, there is strong evidence that the AAM approach is better than the current known approaches in adapting many aspects of software systems. In addition, changes to one aspect have less

impact on the other aspects of a system and less effort is required to accommodate the changes. The actual metrics of the AAM advantages are more difficult to obtain in terms of efforts saved, though the evident benefits have been demonstrated in the case study. How well the AAM performs over existing approaches shall be evaluated quantitatively according to the specified criteria as part of future work. The other set of questions shown on Figure 1.2 with respect to the adaptation role shift can be answered more easily since achieving a Model Driven Agent Architecture is a strategy in the AAM. 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.

Table 9.1: Comparison of ripple effects of changing business needs on AAM systems and traditional systems

Adaptation target	Adaptation	Impacted 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). Corresponding business classes may have to be redefined.	New classes must be developed, or existing ones modified.
	Modification	Policy Rule		
	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. Partnerships may be changed as a result of changed actions in delivering computational results. Business classes may also be used differently.	Fundamental business logic must be re-developed involving comprehension of existing code and tremendous re-implementation.
Partnership	Addition	Business Process Rule & Reaction Rule	Business Process Rules must be re-configured. 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.	Component interfaces must be redesigned. Effect is enormous, as many components have to be changed in the way of communication.
	Modification			
	Deletion			

In summary, from a high level, a major problem of traditional software development is that it can lead to inflexibility. Usually all that could happen under certain known conditions 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 to allow the appropriate response in advance [185]. Unfortunately, decisions must be made when the supporting software system is in design and before the development of the software begins. Potential future business potentials, even if foreseeable, may have to be abandoned in order to complete a working system on schedule. Variations are not allowed when competing solutions/actions conflict each other. Everything must be preset, assuming no uncertainty or future amendment. This pre-determined design with the current vision of the system later resists the emergence and accommodation of new conditions and new actions. This 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.). This situation might not be problematic where, systems tend to be relatively stable in their working environment and function in isolation from other systems. However, systems nowadays may be sophisticatedly connected, influenced by others, and change constantly. The original world view and development model is seldom valid and a means must be sought to bridge the gap. Agents inherently 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. The overall system has the full freedom to meet the challenge of change in any aspect, even after they have been constructed. 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.

9.7.1 Further discussion

In addition to the demonstrated usefulness of adaptivity in the changing business world, more advanced potentials of AAM will be discussed in the next two sub-sections.

9.7.1 Further discussion: fault tolerance, autonomy, and self-adaptation

In AAM, agent behaviour is adaptive and changes according to the changes of business models without re-coding or re-building, honestly reflecting the current functional requirements. Such adaptation is deliberate, as business people know how the system should change in functionality. Sometimes agents are required to adapt to cope with changes to non-functional requirements change such as performance or reliability, spontaneously as a system need arises rather than users say they need it. Such needs may have to be addressed automatically in an emergency. Nevertheless, given the AAM business model structure coupled with the Agent Model, it is possible to extend capabilities alike. Such adaptation is spontaneous, not predicted in advance by people, but internally necessary to ensure regular running of the system.

With such a purpose, AAM agents must be empowered to behave intelligently in reasoning and choosing the correct process to achieve a goal using their accumulated experience, if there are alternatives. This would advance AAM further forward from being adaptive according to business needs explicitly stated by users to being self-adaptive, where systems adapt automatically according to internal needs that have not been predicted previously by business people. Now suppose several routes involving different agents and RRs exist and lead to the achievement of the same goal. These routes are attached with ratings, indicating route attributes, for example, reliability, performance, and so on. The higher the rating of one route, the more "confident" agents are to take that route. The ratings could be dynamically updated. Failure of route execution would cause a reduction to the rating of the route, and could result in the selection of an alternative route the next time by agents. This can be easily implemented as the selection of actions concerning both conditions and rating in each RR. Ultimately, the system tries to avoid problems automatically and lets itself always go through the route it has the most confidence in based on its

dynamic performance rating. This additional feature of AAM is based on the following basis.

- Alternative business processes would exist and could be defined aiming at a common goal
- There is one monitoring agent in an AAM based system, responsible for the global self-adaptation (fault management) of the system wherever appropriate
- All agents send reports to the monitoring agent after execution indicating successfulness or otherwise
- A sequence (as a route) of business rule collection has a rating
- Ratings indicate confidence of execution of business rules
- The monitoring agent changes the ratings of business rule sequences to reroute the business processes if the original route is detected as failure after given times
- Business process composition in a global scale is optimised to avoid faults as a result of adaptation

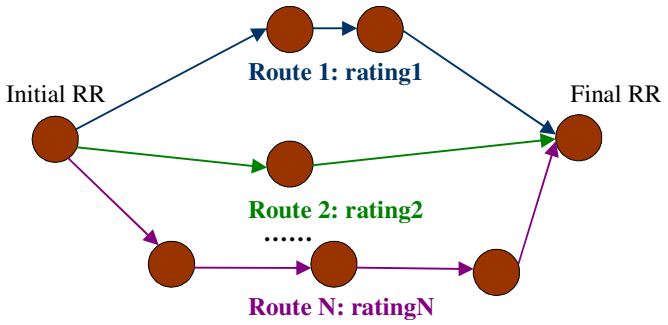

Figure 9.8: The same goal may be achieved through different routes

In Figure 9.8, suppose N routes exist from an initial RR to a final RR to complete a BPR that realises a business goal. Self-healing capability is enabled by using the algorithm illustrated in Figure 9.9. The best route that agents believe to be for a business process would always be chosen dynamically. Thus failure of particular system components need not stop agents from realising business goals. Reports on component failure can be generated for further analysis if necessary.

```
Meta-rule RouteExecution (Route [] routes) {
   int [] rating;
   for (int i = 0; i < routes. length; I ++) {
      rating [i] = routes [i]. rating;
   }
   // find the index of the route that has the highest rating
   int highestRatingIndex = MaxIndex (rating);
   // try alternative routes starting from the one with the highest rating until one route is successfully executed,
note the ratings are dynamically updated in the process
```

```
   while() {
      try {
         executeRoute (routes [highestRatingIndex]);  // go through this route
         break;
      }
      catch (RouteExecutionException ree) {
         routes [highestRatingIndex]. rating - -;  // the rating of the failing route is reduced by 1
         rating [highestRatingIndex] = routes [highestRatingIndex]. rating;
         int k = ree. faultIndex ();  // the index of node fails on the route
         for (int j = 0; j < k; j ++) {
            //  each agent ahead of the faulty point restores its status
            routes [highestRatingIndex]. agentsOnRoute [j]. undoLastOp ();
         }
         // update the index of the route that currently has the highest rating, prepare the route to be executed in
the next run of the loop
         highestRatingIndex = MaxIndex (rating);
      }
   }
}
```

Figure 9.9: Algorithm for dynamic selection of alternative routes

This extension facilitates developing AAM agents as artificial intelligent agents. It adds a reflective sub-system to AAM as an accomplishment of the planned advancement of Reflective and Adaptive Agent Model (RAAM) [2]. This module can also be used for self-adaptation of other non-functional requirements. The use of a dedicated monitoring agent, separated from the other agents for a different purpose, but integrated in AAM, is much easier and cheaper than the usual use of an external controller system that monitors and modifies system behaviour by approaches toward self-adaptation. The recent Rainbow [72] framework, for example, requires system-specific adaptation knowledge together with a reusable infrastructure. For every system under adaptation, separate analysis and development is necessary to allow the use of the framework in specific domains. Further, not every target system provides access to its internal structure for monitoring and change for adaptation. While for systems built with AAM, knowledge is structured in the same fashion, the system and controller are integrated as a whole and open for communication in a normal manner just like ordinary cooperation is carried out within the system. The universal rule schema simplifies both monitoring and adaptation.

9.7.2 Further discussion: Aspect Service

Another extension focuses on crosscutting concerns – an important principle in Software Engineering, which also impacts adaptivity, from another perspective. Traditional approaches like aspect-oriented programming (AOP) [71] separate “aspects”, the code that implements concerns, from the main system. Aspects are weaved in “pointcuts”, locations in the main system where concerns are needed. A

specialised compiler “aspect weaver” is necessary for the combination of aspects with the main system. AOP proponents argue that disentangling crosscutting concerns leads to simpler software development, maintenance, and evolution. However, in traditional AOP the compiled programme is still tangled [70].

As approaches such as AOP have some disadvantages that should be addressed. In response to these some desirable features are as follows: weaving of the aspects is at run-time; no additional weaving and compiling tools, or changed Virtual Machine, or added complexity of code is introduced for the weaving; un-weaving is as easy as weaving and also at run-time with immediate effect; normal system functionalities are not affected at all either by weaving or un-weaving. To this end the intention is to encapsulate crosscutting concerns in modularised aspect services, which should be easily turned on and off as specified in (two-dimensional) rules, integrated into the system but not treated differently from normal services brought by agents. This service should be turned off and back on anytime without shutting down the whole system. The service should be adaptive itself.

It is much easier, better and more natural than many other approaches to address crosscutting concerns in AAM with its built-in architecture. An AAM based agent system running in a distributed environment has system functions split up among responsible agents in a modularised manner. Therefore, some agents can be modularised and dedicated to address crosscutting concerns, called Aspect Service Agents (ASA). ASA are configured with rules to cope with concerns. For example, one ASA with a set of authorisation rules, another ASA with a set of security rules, and the other ASA with a set of logging rules, among others, can be developed and deployed. They are ready to listen to event messages requesting their services and reply with the results, such as granting authorised access, read/write operation permission or logging of events. The universally used RR for agents has an event structure, a processing procedure to follow after the receipt of event, different actions to take in different conditions. These elements help to identify major pointcuts. For example, request events may need to be associated with access control, after completion of actions these may need to be logged for later check or analysis. Therefore former RR schema can be enriched from one dimension to two-dimensions, in that along the $\{\text{event, processing, } \{\text{condition, action}\}_n\}$ axis different aspects of concern can be defined, either for an individual rule, or a group of rules. Following such an extension, it would be possible and useful to define weaving in a

logging aspect after every action is completed, or an authentication aspect before the processing of events in RR owned by external agents. In certain conditions a cache aspect could also be defined to foster later quicker access of information. These potential associations of the two dimensions towards new RR definitions are described in Figure 9.10. After such definitions the establishment of communication between Ordinary Agents (OAs) and ASAs in need is set up virtually. At runtime respective ASAs are invoked automatically during the application of RRs by OAs.

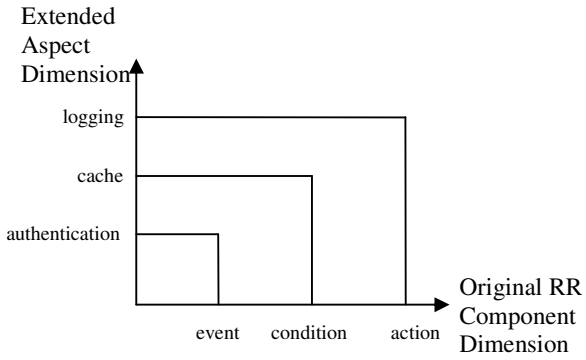

Figure 9.10: Sample separation of aspects from core business as defined in two-dimensional RR

Apart from weaving at the points between RR components, a finer grain for pointcuts definitions within them is available as well. Class methods are registered to be invoked in RRs. Pointcuts definitions therefore could be pushed down to a lower level. An agent can get an advice from a RR that on meeting points during the execution of specific strategies in individual class methods within components (processing or condition) of RR, aspects services are served before or after the invocation.

Since concerns are separated as services, the adverse situation where concerns are spread over the system is alleviated as system components access concern services at specified points. To achieve different levels of Aspect Service, a Monitor Agent is required to acknowledge the required adaptation in various conditions. For example, entrusted customers are not required to be authenticated for certain services; when system is in heavy load, reduce the logging frequency by dropping logging aspects in certain processes. This allows the system to adapt dynamically in another dimension. In addition to Aspect Service Agent (ASA) and Ordinary Agent (OA) as discussed already, specifically dedicated agents in respect to AAM Aspect Service are a Service Directory Agent (SDA) and a Global Context Manager Agent (GCMA).

Several processes run in parallel to bring Aspect Service to AAM by the interaction of these agents.

- Aspect service registration with SDA. This process would run continuously using a Service Registration Tool.
- Rule specification (original dimension) for core business purpose.
- Rule specification (extended dimension) for aspect insertion/change/removal. This process will run continuously using a Rule's Aspect Configuration Tool.
- The GCMA advises global context change of weaving/un-weaving of aspects for a group of agents. For example, at a certain stage, the system removes the restriction of imposing authentication check to a set of agents before services as a changed policy. This alleviates manual edition of many individual rules one by one. The registration of a group of OAs with GCMA and the change of global context is carried out using a Global Context Tool. If any context change occurs, GCMA pushes the change to relevant RRs. OAs update their weaving or un-weaving behaviour accordingly and automatically.
- Weaving/un-weaving for implementing aspects in the main AAM system. This can take place at an individual or group level.
    - Individual: Weaving/un-weaving occurs before/after the rule compositional elements (event, processing, condition, action) and within the rule composition elements before/after atomic operations (class methods, etc.)
    - Group: The same type of rules gets affected together. For example. rules belong to the same agent (a roaming agent always has a mobility aspect); rules involved in a particular process (a process that has a high demand of security has a logging aspect, while a real time process has a performance aspect); and rules that require particular services (confidential services have an access control aspect related with them).

Eventually, adaptive aspects are implemented in OA in several phases.

1. An OA is taking an action/reaction.
2. The OA retrieves a RR as an appropriate reaction rule.
3. It is specified in the rule what aspects are to be weaved into the normal actions.
4. The OA contacts SDA to reach the appropriate ASA.
5. The OA contacts respective ASA for each aspect.

6. Aspects mixed into the normal actions are realised in collaboration with the ASA.

With such an extension of the AAM, the separation of ASA from OA minimises the management of concerns and eases the application of concerns. There is no such problem as code tangling or code scattering. The dynamic addressing of concerns in the OA makes the evolution of the ASA actually the evolution of crosscutting concerns, as deployed in the whole system. The removal of aspects is as easy as addition is in re-configuration of rules. No additional components like a specialised compiler are needed. ASAs are reused, and maintenance of concerns is simplified. The monitoring agent described in the previous section can also be seen as a specific ASA with a concern of fault tolerance and performance being modularised.

Chapter 10: Evaluation & Conclusions

The contributions of the AAM in alleviating the software maintenance burden have been shown, and its advantages over other relevant approaches described. This last chapter further examines and evaluates the AAM approach.

10.1 Evaluation

A convincing quantitative analysis of the AAM approach against the others would be ideal to demonstrate how this agent-oriented solution compares with object-oriented approaches. However, such analysis can only be carried out conceptually at the moment and concrete data is merely unavailable just like most generic agent-oriented approaches and even more established object-oriented approaches [168]. Nevertheless, the benefits of using AAM can be justified qualitatively from various views of software maintainability in relation with user's viewpoint, requirements and design model evolution, business model evolution, software re-configurability and modifiability. The criteria set in Section 1.5 in addition to a useful user perspective justification have been used, where high level adaptivity aims of AAM can now be assessed of their successfulness. From these aspects AAM is compared with other relevant approaches and its advantages and potential weakness are discussed as follows.

10.1.1 User's viewpoint

It is clear that users are dissatisfied with constant upgrades and fixes to their software systems, many of which may be irrelevant to their individual needs. The undesirable cost of upgrading includes re-installation, training, code upgrading, and database maintenance with regard to new releases. This leads to a high cost of ownership of the software. However, users simply want to use software to achieve results they need, but many have no interest in owning the software. The idea of moving software from something that is owned to something that is used, and software being a service is proposed as a solution [95]. In the proposed service-oriented view, upgrade can be broken down into small parts, and only those needed are purchased. This requires late binding, where services are dynamically composed

at the instant of need and then potentially discarded in favour of new ones the next time, as a result of changed needs.

The service-oriented view of software systems aimed at reducing cost is compatible with the AAM architecture. The most valued assets of software systems, the business models that represent business needs are externalised and maintained separately from the IT infrastructure. Such services as running agent systems, or object components are not the real concern of the business, but rather tools that realise their requirements. The replacement of these for better performance, security, or other advanced features is desired but should not affect the current business models. For example, a decision may be made to replace the current agent platform for improved performance. This is decided as a need of the users and better services are later provided independently from the intact existing business models. Also, the maintenance of business models aimed at business advantages should not have an impact on the running IT systems. The separation of the two and their late binding allows a distinction to be made on which is indeed to be maintained, avoiding system degradation traditionally caused by the code-based maintenance, eventually reducing the cost of maintenance. No additional installation, training, and so on is introduced as a result of the maintenance. Users can concentrate on what they want and not be distracted by technical issues.

10.1.2 Requirements evolution

Requirements change is inevitable [87]. The maintenance of a detailed specification is costly and unjustified in situations where requirements change quickly. Failure to define precisely what is required in other situations results in disputes between the client and developers. The need for business to respond quickly to opportunities and challenges leads to a change of the requirements of the system functions. Business merging and restructuring results in the requirement of interoperability. These challenges, among others that are confronted today, motivate the increasingly recognised Integrated Requirements Engineering [87], where it is stated that Requirements Engineering has to be more tightly integrated with system implementation to take advantages of reuse and to let systems evolve to reflect changing requirements. One approach, the agile development method Extreme Programming (XP) integrates processes of RE, design, and development, delivering systems in increments with continuous feedback. This method is useful for the

development of volatile business systems. However, what the agile method value is "individuals and interactions over processes and tools" and "working software over comprehensive documentation" [208]. These two value judgements are based on the historical case of models and the processes used in developing them. They will become out of date and irrelevant over time. This is not the case with the AAM. Also since by using the method, no high level model of systems is maintained, and the development and maintenance remains on a low level of coding, it is more error prone and may require more effort than an approach such as Model-Driven Architecture, where systems are generated from models and maintained in models. AAM has more in common with MDA in this perspective, but at the same time integrates RE into other processes of software development and like agile methods it accepts the inevitabilities of change.

Representing business requirements in hierarchical business models, AAM assigns requirements maintenance to individual elements of layered models. Diagrammatical and textual configuration of various constituent rules to reflect the change of processes, policies, decision making processes, partnerships, and concepts is supported by tools. Minimal effort is required for requirements evolution due to the direct tracking of these business elements to be maintained. Since business models and adaptive agents are bound on the fly, the integration of RE and implementation/deployment is inherently supported in AAM and so completed after requirements evolution. Change to business models has immediate effect on the running system. In addition to the maintainability of standalone applications, the explicit ontological modelling of Business Concepts and inter-BPR across business organisations enables interoperability. This has been described in previous sections.

When requirements evolve, it usually becomes more and more difficult to understand the implemented code. In addition, the impact of the proposed change on systems must be analysed. For the purpose of minimising risks associated with the evolution of component-based systems and tracing change impacts, an intermediate modelling layer comprising services and constraints has been introduced in [107] to map from requirements to components. Alternatively, better code organisation could assist the understanding of evolving systems. A software clustering approach [100] proposes to cluster functional requirements and recover from the resulting formation a hierarchical structure of source code. The goal of partitioning system code is to reflect distinctive domain concepts and indicate roles. In constructing the structure, a

mapping between component code and related requirements is established, and the source code is partitioned off. These approaches are in contrast with AAM, where the requirements organisation persists and assists the ongoing comprehension of systems. Among the business models, the central components of RRs are organised around responsible agents, describing actions to take for events after decision making processes. Agents represent conceptual domains that respective requirements naturally fall into. Such requirements organisation helps with requirements comprehension during the maintenance phases. Evidently, the explicit organisation and externalisation of the requirements and the maintenance of a direct mapping from the requirements to the running agent knowledge in AAM are advantageous. The hierarchical agent organisation mechanism [1] supports even better requirements comprehensibility while systems evolve.

10.1.3 Design model evolution

When large complex systems evolve, the extraction of specific subsets of design models that direct the maintenance tasks could be difficult. A model slicing approach [94] that extends the classical program slicing supports sub-model extraction by iterative discovery of relevant elements from traversal paths of model element relationships. This technique is useful to identify all classes requiring maintenance due to a particular class that they are related with having been changed. Thus the maintenance task can be constrained within the scope affected.

An abstraction mechanism has been presented in [1] for AAM that reduces the complex interdependencies between agents by decomposing agent responsibilities and building a hierarchical agent organisation. Any agent only directly relates with other agents in the same organisational level, or the one and only agent in a higher level from which that agent originally derives. Consequently, cross-boundary coupling has been minimised, and slicing for relevant sub-models is immediate. Changes are managed locally and prevented from rippling. Eventually, maintenance tasks are minimised and limited within the scope of directly affected parties.

10.1.4 Business model evolution

i) Business process

In practice, business processes are maintained separately from their supporting software systems in organisations, though both are triggered by changing business

requirements. Misalignment between business processes and software systems may arise when changes performed on two parties do not match. Impact analysis [107] that assesses the complete set of affected items caused by a particular modification is a technique useful for identifying required additional interventions to restore alignment. An approach [97] aimed at aligning business processes and software systems models dependency relationships among items from both sides: business processes, process activities, software systems, software components, and so on. Thus the modification of one item recursively propagating to all other affected items can be analysed to keep both business processes and software systems evolving concurrently and accurately. However, the analysis of the modification impact propagation has to be performed manually, which is time-consuming and error-prone. This is in contrast with the AAM, where business processes are explicitly modelled and kept maintained in the BPRs. The models along with any change made on them are interpreted automatically and effects are reflected immediately. The maintenance of BPR is the maintenance of the whole systems and so they are always aligned.

ii) Business rule

In classifying software evolution and maintenance types in a finer granularity [106], the business rule cluster has been recognised as the most frequent and most significant in supporting enhancive, reductive and corrective software evolution and maintenance. The maintenance of business rules, including adding, fixing, replacing, or removing them can address many software maintenance issues. The terms ‘adaptive’ and ‘corrective’, as it is proposed, should be redefined by narrowing their applicability to software properties and business rules respectively. This perspective is shared by the AAM, business rules being the primary maintenance target. Correction and adjustment of them is the norm in regulating business strategies for business operations. Overall, the system is adaptive as a consequence of flexibly defined business rules and immediately reflected software behaviour.

Unfortunately, in many cases rules are embedded in systems. This makes such systems difficult to maintain, provided that business rules are subject to change according to changing business needs. This in turn requires the change of the software systems that implement the rules. Scattered code across systems that together implement the rules has to be inspected completely to ensure accurate update, which is effort intensive and liable to omission. One approach [96] that

facilitates the evolution of hard-coded rules requires the identification of event sets that may violate rules, guard conditions that determine if these events will violate rules, and actions that must be taken if a violation occurs. The business rule evolution involves a complex process in which old code that implements guards and actions for events as defined in old rules is removed, and new code that implements guards and actions for events as defined in new rules is added. Mappings from high level rules into code fragments that implement rules are defined to support the evolution process. One shortcoming of the approach pointed out by the authors themselves is the problem of mapping maintenance when changes are made to the code. Also the mappings have to be established manually and so extra complication is introduced in the evolution process. Although this approach supports evolution of rules already implemented in systems and potentially reduces the usual evolution effort by using a systematic method, the externalisation of events, conditions, and actions supported by automatic interpretation of them as proposed by AAM has an evident advantage over it.

iii) Business concept

An emerging crucial maintenance task is the integration of existing software systems into an interoperating larger system. Many ontology based methodologies are hence proposed. The simple but clear example of OASIS service sharing methodology [99] uses the components of Domain Ontology and Conceptual Service Adapters to enable interoperation. The outcome is that participants of the integration can work independently, they are changeable without affecting the integration and ultimately maintenance of them is localised and separate. The Business Concept Model in AAM supports the maintenance of interoperable systems in a similar means.

iv) Co-evolution of business and IT systems

AAM provides a solution to the widely observed gap between the business systems in demand and the IT systems in operation. IT systems usually lag behind the current business and become legacy systems little by little [83] [209]. Co-design of business and IT systems is put forward, argued in [83], for an organic integration of IT into the business processes, so that both systems (co-)evolve naturally. This adopts a view that a business organisation is essentially an information system [84]. An organisation is concerned about business functions being performed and business objectives being achieved, through the proper use of information. IT systems can

only extend human capability, and only when they are incorporated properly into the existing business infrastructure. The deployment of an IT system in an organisation should only change the way the business is conducted but not its nature. A model of business organisation is described as having three categories of activities: substantive activities, communication (message passing) activities, and control activities are distinguished [83] [84]. In the first class, tasks are derived from organisational objectives to contribute to business goals. They are carried out in informal (a sub-culture established of intentions, responsibilities, commitments, etc.), formal (rules define procedures, behaviours, formats, etc., replacing informal construct meanings), and IT (automates the formal rules) sub-systems. This class of activities consist of the core business, and so they must consume most resources to direct a "healthy" organisation [83]. In the second class, communication messages as "input" and "output" of communication processes express intentions of communication parties, supporting substantive activities. In the last class, control activities ensure that behavioural subjects act properly in performing substantive tasks. The IT solution of AAM has its models perfect matched with this model of business system. BPR, RR, and PR are various types of rules in an organisation that replace the cultural constructs and ready to be interpreted in an automation means. AAM communication infrastructure supports the interaction of organisational parties with agreed meanings and intentions. Agents take the business responsibilities by message passing. These three aspects of AAM modelling exactly implement the three categories of activities of business organisations. AAM therefore integrates business and IT systems. The evolution of business systems and their supporting AAM systems is to be carried out concurrently. Thus systems built using the AAM will never be legacy systems and the maintenance of AAM systems will always reflect the changes to the actual business system.

10.1.5 (Re-) configurability

Software (re-) configurability is a focus of current research [85] [88] [89] [98] on software maintenance. This work on AAM towards the non-functional attributes of adaptivity and maintainability of software system is also closely related with re-configurability and modifiability. One may define configuration as such [89]: "A configuration is all of the entities that are present in a system and the interrelationships between them". In other words, it models a structure of

components and their linkages. Configuration has been widely used in various areas of computing world with the potential to achieve dynamics: Architecture Description Language (ADL) describes system components constitution; UML diagrams model class component relationships in collaborations, activity component relationships in processes, and so on; LCFG [90] allows automated installation and configuration of distributed Linux computers using configuration files stored on a configuration server. Related work concerned with the particular configuration target of software behaviours have been evaluated and compared with AAM, as follows.

i) Business process self-reconfiguration

Business processes have to be constantly customised to reflect changing business environment, structure, and objectives. Multiple business roles accomplish various tasks by using specialised and associated business applications in workplaces sequentially, realising business processes. As a consequence of changed business requirements and hence business processes, such changes such as additional tasks and corresponding applications, re-binding between tasks and applications, and re-assignment of tasks to roles are required. One approach attempts to handle this challenge by proposing a runtime reconfiguration framework [88]. The framework compares the differences between the business process before change and the business process after change, in their configuration of task control flow, associated with roles accomplishing the tasks and applications fulfilling the tasks. The generated configuration updates are populated to appropriate workplaces. Constraints of the framework include the consistency issue that must be considered. Reconfiguration points are identified with regards to upload of new applications. If the current application is in the "inconsistency spot", it must terminate and the process rolls back. If reconfiguration is carried out when the tasks/applications to be reconfigured are running, inconsistency and invalidity of the entire process may occur, and the current application processing disturbed.

In AAM, agent workers replace human workers to carry out the required tasks by executing rules, supported by business components, realising business processes. New tasks can be added freely through additional rules, specifying sub-processes with decision making and input/output data. New components can also be accommodated in relation with rules, facilitating agents to behave. The binding between agents and tasks that agents must accomplish are specified in re-configurable rules, which can dynamically assign new tasks to agents or change the

nature of existing tasks. Therefore, the framework in the AAM is more flexible and free of such constraints in supporting the reconfiguration of business processes in a changing business environment. This advance is attributed to the more fine-grained business models in a layered style, associated with an agent-class computing component hierarchy.

ii) Re-configurability in agents

Bond agent framework [82] built upon JADE [73] supports re-configurability and mutability. Agents are assembled from components, initialised knowledge, and mutation operation descriptions. A re-configurable multi-plan state machine is used as an agent strategy model. The multi-plan state machine is described in Blueprint script, declaring agent states, plans, and transition strategies. Mutation actions such as adding or removing the states and transitions of the state machine, called "agent surgery" operations, are allowed to perform to adapt agent strategies. This paradigm achieves re-configurability by externalising configurable agent strategy model, similar with AAM using the business models to capture agent behaviours. However, the proposed agent surgeries that transform agents require writing new script which is essentially code. More importantly, the Bond framework only provides a means that dynamically changes between existing states and plans of agents, and so offers limited adaptivity in agent behaviours. On the contrary, models in the AAM envision business needs and potential changes from every perspective using hierarchical business models and so can tune agent strategies at the various business levels.

iii) Configuration modelling and analysis

Stride [89] establishes configuration models of all social and technical components within a system. Its process model interrelates activities operationally, each of which is associated with Initiators, Targets, Inputs, and Outputs. Analysis is performed by Stride on all available paths leading to any given objective, following control flow within configuration models. This assists the investigation of system configuration with regards to the attainment of desirable objectives and blocking of undesirable objectives. Due to the incorporation of all system elements in its configuration models and analysis of them at a high level, the appropriateness of system configuration can be appreciated by analysts early and quick, this being especially useful for complex real world systems. Investigation of reconfiguration is made easier as systems evolve.

AAM brings comparable benefits to system analysts in using the Business Process Rule Model. The documentation of BPRs allows the establishment of a system blueprint. Following investigation at an overall system level all required system goals can be verified and validated. The reconfiguration of BPRs is fully supported in the AAM. The clear compositional layers of BPRs enable the change of all their lower level composing elements. This allows the reconfiguration of a system in all possible aspects: business partnerships, business policies, business decision makings, and so on. Alternative communication paths forming the same BPR enable fault tolerance and ensure the best possible performance, as described earlier. Although agents can also be regarded as machines or even people working in organisations with rules that constrain their behaviour, AAM does not model fully the combination of technical and social components. The interplay of them could involve complexity and unpredictability into the system. One advantage of Stride that could be added to AAM in the future is the capability to examine the outcome of a reconfiguration before it is put to use. This is especially useful in the AAM, since the reconfiguration of business models will be reflected directly in the system and there is no appropriate way to evaluate the changes made by business people at the business level. At the moment business people must be responsible for their own operation on the reconfiguration of business models. Supporting tools may need to be empowered to assist in validating the consistency and sensibility of the proposed reconfiguration. This could incorporate a presentation of a visualised high level business analysis of the configuration to the business people, and help them to ensure that the result reflects what they want in the real world. This will form part of future work on the AAM.

iv) Construction by configuration

The reuse and configuration of existing components or systems has been successful in ERP systems [98], COTS systems [107], software system families, service-oriented systems, and even programming languages such as Java. Construction-by-configuration (CbC) [85] [87] proposes that software is configured rather than programmed to cope with specifications and environment change. Software evolution by re-configuration is in contrast with the conventional practice, where code analysis and maintenance is the focus. Traditionally, such work becomes increasingly difficult when the system becomes more and more complex. The separation of the development and maintenance teams also leads to uneasy

understanding of the system by maintainers. Little research work, if any, addresses maintenance issues using this effective method, though some techniques such as program visualisation can offer support for construction by configuration.

Extensive reuse and re-configuration is the norm and a predominant approach in business systems development (ERP systems, etc.) [86]. The focus is transient from technology-oriented abstractions to built-in business-oriented domain abstractions. High level functions are provided to operate directly on the business abstractions to achieve a direct effect on business. In CbC, processes, services, and components might need to be adapted but minimal effort is required. Very often configuration and adaptation is in glue code that links these entities. Though such a paradigm is useful in accelerating software development, reuse is only applicable within specific application domains, generic functionalities being configured for specific use. For example, a generic COTS package developed for hospitals in England has been adapted for a patient management system used in Edinburgh, Scotland under the paradigm [86]. Application of reuse and configuration in CbC requires the identification of generic environments to choose the reusable system, and then the identification of specific requirements from the setting of use, as the basis of configuration. Many CbC methods are limited in the capability to handle cross-domain reuse and configuration, and the runtime assembly of components [82].

AAM supports the same idea of construction by configuration. Its business models are all configurable at runtime to support continuous software maintenance. At the same time it is not limited in the application domains that it is applied to. Both business knowledge and existing business components can be reused across domains, agents integrating and interpreting these dynamically. Software systems do not need to be built from scratch but the business knowledge does, which is essential and dependent on the particular application domain. Once business knowledge is specified by business people and available to AAM, the software components and knowledge assemble together as a complete system automatically, no matter what application domain it is used in. The maintenance of business knowledge is the maintenance of the AAM based systems.

10.1.6 Software architecture required by modifiability

Tactics are introduced as architectural design strategies that software architects can use to achieve quality requirements [91]. Modifiability tactics are described as

including three categories: “localise modifications”, “prevent ripple effects”, and “defer binding time”. The tactics of “localise modifications” aims at restricting modifications to a limited set of modules. A typical example of a tactic in this category is to “maintain semantic coherence”, which ensures responsibilities in a module work together without excessive reliance on other modules. Sub-tactics include “abstract common services”, where if common services have been abstracted, modifications will need to be made only once rather than in each module where the services are used. The tactics of “prevent ripple effects” aims at minimising module dependencies by information hiding and maintaining existing interfaces, and using intermediaries to manage dependencies. The tactics of “defer binding time” aims at deferring time to deploy and allowing non-developers to make changes by using additional supporting infrastructure. Sub-tactics include “configuration files”, “runtime registration”, and “component replacement”.

In AAM, these tactics are reflected in various forms. Firstly, the ontology level of changes are localised in its Business Concept model; the policy level of changes are localised in its PR model; the changes of partnerships, decision makings, and internal processing of individuals are localised in its RR model; and finally, the changes of interactions and global goals in its BPR model. Any change in a certain level is localised and won’t ripple to other levels or require corresponding additional changes. For example, the adjustment of a policy in increasing the priority of dealing with faults associated with a particular location allows the handling of forthcoming events that occur at the corresponding site by the agent that previous ignore such events. Changes to RR or BPR that directly relates to or encapsulates this PR are not necessary. Agents take changes separately and automatically. Moreover, updated classes can bring changed effects to systems, supporting the same untouched agents that will dynamically use the classes. These two aspects of modification localisation attribute to the nature of layered hierarchy of both business models and computing components of AAM. Additionally, the interactions among collaborative agents or among agents and objects are hidden away from agents in a separate knowledge source to prevent fixed relationships and so dependencies. Rather, such information is externalised in the adaptable business models accessible only when required, so that the AAM component interfaces are dynamically maintainable. Business models serve as an intermediary that is responsible for managing the changeable relationships among computing components by encapsulating their dependencies

explicitly. The effect of model change is immediately reflected in the overall system behaviours. Lastly, because the business models as a whole are dynamically bound with agents who execute the models at runtime, the binding time is deferred to the last moment possible, the time when the model knowledge actually needs to be used and applied. Binding at runtime means that systems are prepared for the binding without need for testing or distribution steps to follow, so that changes take effect in systems immediately. Thus, all the three proposed tactics that are useful for the quality requirements of software architecture have been integrated into the architecture of AAM.

10.2 Discussion and Conclusions

This thesis has compared OO approaches with AO approaches with respect to adaptivity. A related argument is that technology-oriented maintenance should shift to business-oriented maintenance. It has been argued that the inherent design of objects in OO methodologies complicates the achievement of adaptivity. Another crucial reason that adaptivity has not been addressed efficiently is that most solutions are technology-oriented, while the original need of adaptivity and the driving force of technology, the changing businesses are ignored.

Compositional adaptation, its key technologies currently available including separation of concerns, computational reflection and component-based design, is generally regarded a good approach for software adaptation, an interesting survey with review of them being found in [70]. These support dynamic re-composition of system components, structurally or strategically, by using a "composer", an entity that is used to adapt program entities. It could be a human (developer, administrator, etc.), or a piece of software (aspect weaver, component loader, meta-object, etc.). *How*, *when*, and *where* re-composition occurs are the primary concerns and criteria to compare various technologies.

In all of these technologies the emphasis is on dynamic component architecture, and little attention is given to *why* system components need to change. It is hard to change the behaviour of already coded programmes even with a "composer" to help since it adds complexity and uncertainty. In contrast, the AAM externalises the source of change in business models, as easily adaptive models that can be specified and changed explicitly by business people any time any where by any means according to changing business needs. These executable specifications are the

knowledge for agents, deployed immediately they are available without human intervention or system re-builds. Such an alternative perspective distinguishes the AAM from other approaches.

The principle that adaptivity should be supported at the point of its origin, i.e. business needs, ironically brings AAM more advanced adaptivity, technically, than other approaches that emphasise the technical perspective. Many of those and their supporting platform/architecture have no concern of some important respects of adaptivity. Configurable dependencies and relationships among system components are usually the primary concern among many approaches towards easy change of systems. Various efforts are put towards configurable connectors or interfaces. C&C viewtype [92] is defined in documenting software architecture. Component types representing principle processing units and data stores, and connector types representing interaction mechanisms constitute the C&C viewtype, in which components interact over connectors. Similarly, CADL architectures [107] consist of components, services, connectors, component ports, and so on. Also, ACME [93] uses a combination of nodes and connectors to construct a structural model of system composition. Coordination Contract and more recently Service-Oriented Architecture are based on similar ideas. However such interplay between system components is not the only possibility that causes required change. Internal processing in individual components also needs to change from time to time. Moreover, ontologies used in systems need to be changeable as systems may be enlarged and take in additional concepts. In many agent-oriented systems, interfaces of agents are not well enough defined to cope with dynamic collaboration; individual agent behaviour is incapable of accommodation of additional functionalities or decision making; as well as the vocabularies that spoken by agents are not extendable at runtime. These difficulties represent a barrier to using agents to adapt systems. Currently no single solution targets all these perspectives. In AAM, all three identified aspects of changing requirements are addressed.

10.2.1 AAM reuses existing technologies as well as knowledge

When a new agent-oriented solution is determined, the importance of reusing existing technologies (OO development) and knowledge (knowledge modelling) has been emphasised. A useful approach that provides a language facility to support agent-oriented software development, called the *caste* [205] [206] [207], is closely

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

The AAM organises requirements around conceptual agents, supported by lower level objects. Business Process Rules realise common goals in a global view, and lower-level business rules deal with sub-processes or policies for every single agent. This hierarchical abstraction structure, agent-object in the IT level, and BPR-business rule in the business model level, breaks down problems and simplifies the degrees of complexity. A higher level of abstraction is achieved not only in externalised requirements as rules, but also in the implementation architecture that carries out the requirements. In another aspect, the Software Engineering discipline of indirection is used to insulate separate concerns from being too tightly connected with each other [59]. Leveraging ontology as indirection builds in semantic interoperability to manage adaptation. AAM achieves indirection in that the selection and use of underlying modelling elements in upper level elements is dynamic, reflecting the current business needs across all its layers: the composition of BC in PR, PR in RR, and RR in BPR.

10.2.2 AAM has been developed as a complete methodology.

The AAM methodology guides the full agent-oriented development process; provides artefacts such as meta-model, structural and behavioural diagrams, XML schema, notations; and offers supporting tools that automate the process and facilitate the design and development. A top-down and a bottom-up approach are used together so that (additional) requirements can be easily assigned to responsible agents and existing systems or system components can be fully reused. Therefore not only agent-focused developers can benefit from the AAM by reusing objects but also object-focused developers can quickly understand, master, and adopt the agent approach by an extension of the object construct. AAM bridges the gap between object technology and agent technology and promotes agents by offering practically and pragmatically usable development methods and supporting tools. It is expected that AAM can be an entry point for traditional developers entering the adaptive agent world and reduces the development time significantly by reusing the existing infrastructure and knowledge.

10.2.3 AAM makes software maintenance easier

The AAM provides a means for easy software maintenance and is practically useful to a business-centred world. The architecture of the AAM is decided by reusable business models that can be adapted at runtime, via the Agent Model, which carries out the business models. AAM provides a concrete and pragmatic solution in accordance with MDA. Therefore, maintenance can be concentrated on the real business asset, being the business models themselves. Also, software must go on [101]. Business in the real world depends upon continuous working software and inter-system interdependencies. Supporting continuous software maintenance [105] without downtime is becoming more and more important. The AAM contributes a full elaboration agent framework and executable business models not only of pragmatic value for applications that need adaptivity, but also can be tailored for specific use (ARC framework, business models, etc.) due to its well-definition and clear de-composition of models.

The current state of Software Engineering research has been examined in [195] using 369 papers in six leading research journals in the SE field. The result was that SE research is “diverse regarding topic, narrow regarding research approach and

method, inwardly-focused regarding reference discipline, and technically focused regarding level of analysis". An implication for SE research in general is that, research in the SE field is severely decoupled with the state of the practice of the field. An "assimilation gap" between the first acquisition of a new technology and its 25% penetration into practical software development is found to be a very long period: 9 years for relational database, 12 years for Fourth Generation Languages and even longer for CASE tools. Therefore it is very important for a SE research methodology to be practically useful and adoptable to existing practitioners who need to be willing to apply the research results.

In AAM, the adaptation and maintenance is in the hands of business people themselves and so they can change the system according to their ever changing business needs, with immediate effect, anytime they want. This convenient adjustment of business strategy directly by business people allows them to enjoy the immediate benefits of the new strategies. Businesses are reluctant to shift their computing systems, in which they have already invested and are continuously producing value to the business only because of the aesthetic attraction of new technology. Use of the AAM gives re-configurability, provides extensibility of system architecture, strategy, and concept and at the same time integrates with current object-oriented systems. In this way, the acceptance of new agent technology should be accelerated, both by business people who own the existing OO assets and by existing OO developers who are experienced with OO development.

10.2.4 Conclusions

Summarising the above arguments, conclusions can be made against problems stated in Chapter 4.

- A new MAS approach for software adaptivity rather than OO based approach has been demonstrated as necessary (Chapter 1, 2, 3, 4).
- The methodology of AAM has been developed, reusing existing knowledge and technologies and so easing its adoption (Chapter 5, 6, 7).
- The AAM supports a complete MAS development process including concepts, models, and tools facility, and so makes such an approach practically useful in real applications (Chapter 6, 7, 8).
- The AAM significantly alleviates maintenance cost of software systems (Chapter 9, 10).

No prior research has been undertaken using agent-oriented software development and business knowledge modelling in the area of software adaptivity. Relevantly, little prior work has been reported on realising all the aspects of adaptivity documented in Table 2.1. In the thesis the approach and experience has been illustrated with a real large business application. As it has been described in Chapter 1, contributions of this research about AAM include the following.

- AAM has been developed as a methodology that puts Model-Driven Architecture paradigm into practice in agent-oriented software development.
- The maintenance of business knowledge models diagrammatically and textually becomes the maintenance of the overall software system and so software adaptivity is achieved, with effect of adaptation reflected dynamically and immediately at runtime.
- Business people are enabled to directly control their own business as system maintenance becomes a daily routine of business practice. This shift of responsibility represents a significant change of role playing in software development.
- AAM supports a full development process for agent-oriented software systems towards adaptivity.
- A hierarchical structure of business process, business rule, and business concept is organised as business representation across business domains, elicited from business requirements and documented in UML and XML for easy business maintenance and dynamic agent execution.

10.3 Future work

Two useful extensions of AAM, self-adaptation and aspect service, have been described in Chapter 9. More powerful additions could be accommodated by the AAM's flexible architecture. Some additions are in line with the envisioned potential weakness of AAM and others can make the approach more useful. Whatever, they are mainly in two aspects as the compositional structures of the AAM.

On the agent aspect, AAM agents may later potentially have capabilities of reasoning and learning. The existing use of Policy Rule chain already allows agents to make inference to some extent. Agents reason about the satisfaction of rules and, little by little, use a chain of rules to accumulate knowledge. Later their capability

can be extended so that when facts/actions are targeted, the relevant knowledge that may satisfy the conditions to trigger them may be sought by agents. The existing use of Reaction Rule also allows agents to learn the knowledge about what to do when something happens in various circumstances. The outcome of the preset behaviour can be seen and evaluated. When such experience is built up little by little, possibly through a dedicated intelligent agent, such actions may be adjusted automatically to improve their performance as a result of intelligent decisions and self-adaptation. This is in contrast with the current situation where agents honestly turn required business needs into reality, despite the possible conflicts or inconsistency residing in the requirements. Another potential of adding intelligence lies in the built-in structures of the business rules. Since the actions to be performed by both forms of rules (PR & RR) are imparted by the business people with business knowledge, actions of the rules can be inferred and evaluated before hand by agents against the potential results and the most productive ones selected before they are executed, leading to a more robust and optimistic architecture.

On the business knowledge modelling and the model driven aspect, two directions could be followed to improve the AAM, overall. On one hand, the tools supplied to business people could be made more understandable using natural languages. Improvement on smarter user interface in the context of the AAM supporting tools depends much on the state of the art of Ontology and Intelligence technologies. Another related issue is that a method for testing the models built through the tools needs to be developed. Currently, the AAM covers requirements, design, implementation, deployment and maintenance but not testing. Business customers are totally responsible for their models and the agent system always executes the specified models. The validation of the models as well as the testing of the running system towards the actual business needs is necessary since customers could make mistakes while configuring the models. This is a completely separate area of exploration, since the AAM requires model testing rather than code testing. On the other hand, business rules can be enriched in format to represent as wide a scope of business requirements as possible. This can enhance the applicability of AAM to various systems. In addition, the current XML format of business 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 builds up its behaviour by: looking for the appropriate rules according to the current models decided by the current

requirements; parsing the relevant business information; building computing components; and finally executing what the business model tells it to do. All these happen at runtime. Although such a mechanism brings about adaptivity to the AAM it could slow down the whole system. In addition to parallel computing architecture and powerful hardware investment, solutions that could ease this problem include pre-building object components for agents and cache technology. At present, related to this, computation intensive applications as well as those where requirements are unlikely to change frequently can be considered inappropriate application targets for AAM. Other specific types of software systems where the application of AAM is limited will be a matter for further investigation.

More complex situations, where agents execute rules that are being updated by business people need to be considered. The integrity of the AAM system must be ensured. Another issue that needs further investigation is related with concurrency, where a rule could be requested for execution at the same time by two agents. A problem would occur if the rule has to be executed for only once in the system, operating upon a single temporary task or exclusively utilisable resource.

It has been demonstrated in recent work published during this PhD research the suitability and usefulness of AAM to achieve adaptivity, in the areas of requirements modelling [3], design modelling [4], and implementation [5] [7] [210], for building agent-oriented systems [2] [6] and agent-oriented knowledge modelling [162]. The methodology covering full software development life cycle with formal guidelines is discussed in [1]. The AAM framework will be made more powerful and adapted to apply in other domains. Work so far indicates that it will contribute in a novel and substantive way to adaptive agent-oriented systems.

References

[1] Xiao, L. & Greer, D., "The Agent-Rule-Class Framework for Multi-Agent Systems", Special Issue on Agent-Oriented Software Development Methodology, *Multiagent and Grid Systems - An International Journal, Number 4, Volume 2*, IOS Press, to appear in 2006.

[2] Xiao, L. & Greer, D., "Externalisation and Adaptation of Multi-Agent System Behaviour", Chapter IX in Siau, K. (Eds.), *Advanced Topics in Database Research, Volume 5*, Idea Group, Inc., pp. 148-169, 2006.

[3] Xiao, L. & Greer, D., "Agent-oriented Requirements Modelling", *Proceedings of the First International Workshop on Requirements Engineering for Business Need and IT Alignment (REBNITA'05)*, pp. 28-37, Paris, France, 29-30 August, 2005. In conjunction with the Thirteenth IEEE Requirements Engineering Conference (RE'05).

[4] Xiao, L. & Greer, D., "Modelling Agent Knowledge with Business Rules", *Proceedings of the Seventeenth International Conference on Software Engineering and Knowledge Engineering (SEKE'05)*, pp. 566-571, Taipei, Taiwan, Republic of China, 14-16 July, 2005.

[5] Xiao, L. & Greer, D., "The Adaptive Agent Model: Software Adaptivity through Dynamic Agents and XML-based Business Rules", *Proceedings of the Seventeenth International Conference on Software Engineering and Knowledge Engineering (SEKE'05)*, pp. 62-67, Taipei, Taiwan, Republic of China, 14-16 July, 2005.

[6] Xiao, L. & Greer, D., "Modeling, Auto-generation and Adaptation of Multi-Agent Systems", *Proceedings of the Tenth CAiSE/IFIP8.1 International Workshop on Exploring Modeling Methods in Systems Analysis and Design (EMMSAD'05)*, pp. 605-616, Porto, Portugal, 13-14 June, 2005. In conjunction with the Seventeenth Conference on Advanced Information Systems Engineering (CAiSE'05).

[7] Xiao, L. & Greer, D., "Software Adaptivity through XML-based Business Rules and Agents", *Proceedings of the PREP2005*, pp. 287-288, Lancaster, UK, 30th March-1st April, 2005.

[8] Cockburn, A., "Structuring Use Cases with Goals", *Journal of Object-Oriented Programming*, pp. 35-40, Sep/Oct, 1997 and pp. 56-62, Nov/Dec, 1997.

[9] Fowler, M., *UML Distilled: A Brief Guide to the Standard Object Modeling Language, Third Edition*, Addison-Wesley, 2004.

[10] Gou, H., Huang, B., Liu, W. & Li, X., "A framework for virtual enterprise operation management", *Computers in Industry* 50(3): 333-352, 2003.

[11] Bajec, M. & Krisper, M., "A methodology and tool support for managing business rules in organizations", *Information Systems* 30(6): 423-443, 2005.

[12] Nilsson, B.E., "On why to model what and how: concepts and architecture for change", in: Nilsson, A.G. et al. (Ed.), *Perspectives on Business Modelling—Understanding and Changing Organisations*, Springer, New York, 1999.

[13] Bubenko, J.A., Persson, A. & Stirna, J., "D3 Appendix B: EKD User Guide", Royal Institute of Technology (KTH) and Stockholm University, Stockholm, Sweden, 2001.

[14] Müller, J. P., Bauer, B. & Friese, T., "Programming Software Agents as Designing Executable Business Processes: A Model-Driven Perspective", *PROMAS 2003*, LNCS 3067, pp. 49-71, Springer, 2004.

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

[16] Andrade, L. & Fiadeiro, J.L., "An Architectural Approach to Auto-Adaptive Systems", *Proceedings of the 22nd International Conference on Distributed Computing Systems Workshops*, 2002.

[17] Yoder, J.W., Balaguer, F. & Johnson, R., "Architecture and Design of Adaptive Object-Models", *ACM Sigplan Notices* 36(12): 50-60, 2001.

[18] Yoder, J.W. & Johnson, R., "The Adaptive Object-Model Architectural Style", *Proceedings of the 3rd IEEE/IFIP Conference on Software Architecture: System Design, Development and Maintenance*, pp. 3-27, 2002.

[19] Yim, S.H., Ahn, J.H., Kim, W.J. & Park, J.S., "Agent-based adaptive travel planning system in peak seasons", *Expert Systems with Applications* 27(20): 211-222, 2004.

[20] Jia, Z.H., Ong, K.S., Fuh, J.Y.H., Zhang, F.Y. & Nee, A.Y.C., "An adaptive and upgradeable agent-based system for coordinated product development and manufacture", *Robotics and Computer-Integrated Manufacturing* 20(2): 79-90, 2004.

[21] Goh, T.W. & Zhang, Z., "An intelligent and adaptive modelling and configuration approach to manufacturing systems control", *Journal of Materials Processing Technology* 139(1-3): 103-109, 2003.

[22] Laleci, G.B., Kabak, Y., Dogac, A., Cingil, I., Kirbas, S., Yildiz, A., Sinir, S., Ozdikis, O. & Ozturk, O., "A Platform for Agent Behavior Design and Multi Agent Orchestration", Agent-Oriented Software Engineering Workshop, the Third International Joint Conference on Autonomous Agents & Multi-Agent Systems, 2004.

[23] Wagner, G., "The Agent-Object-Relationship Metamodel: Towards a Unified View of State and Behavior", *Information Systems* 28(5): 475-504, 2003.

[24] Lamsweerde, A., Requirements engineering in the year 00: A research perspective, *Proceedings of the 22nd International Conference on Software Engineering*, pp. 5-19, ACM Press, 2000.

[25] Yu, E., "Towards Modelling and Reasoning Support for Early-Phase Requirements Engineering", *Proceedings of the 3rd IEEE Int. Symp. on Requirements Engineering (RE'97)*, pp. 226-235, 1997.

[26] Kavakli, V. & Loucopoulos, P., "Goal-Driven Business Process Analysis - Application in Electricity Deregulation", *Information Systems* 24(3): 187-207, 1999.

[27] Yu, E., "Why Agent-Oriented Requirements Engineering", *Proceedings of the 3rd International Workshop on Requirements Engineering: Foundations for Software Quality*, Barcelona, 1997.
[28] Morgan, T., *Business Rules and Information Systems*, Addison-Wesley, 2002.
[29] Lamsweerde, A., "Goal-Oriented Requirements Engineering: A Guided Tour", *Proceedings of the 5th IEEE International Symposium on Requirements Engineering*, pp. 249-263, 2001.
[30] Feather, M.S., "Composite System Design", ICSE-16 Workshop on Research Issues in the Intersection Between Software Engineering and Artificial Intelligence, International Conference on Software Engineering, Sorrento, Italy, 1994.
[31] Dardenne, A., Lamsweerde, A. & Fickas, S., "Goal-directed Requirements Acquisition", *Science of Computer Programming* 20(1-2): 3-50, 1993.
[32] Heymans, P. & Dubois, E., "Scenario-Based Techniques for Supporting the Elaboration and the Validation of Formal Requirements", *Requirements Engineering Journal* 3(4):202-208, Springer, 1998.
[33] Donzelli, P. & Bresciani, P., "Domain Ontology Analysis in Agent-Oriented Requirements Engineering", KES 2003, pp. 1372-1379, LNCS 2773, Springer, 2003.
[34] Donzelli, P. & Bresciani, P., "Goal-Oriented Requirements Engineering: a Case Study in E-Government", *Proceedings of the 15th Conference on Advanced Information Systems Engineering (CAiSE'03)*, Velden, Austria, 1377, 2003
[35] Bresciani, P. & Donzelli, P., "The Agent at the Center of the Requirements Engineering Process: A Framework for Complex Socio-technical Systems", SELMAS 2003, pp. 1-18, LNCS 2940, Springer, 2004.
[36] Riehle, D. & Perry, A., "Framework Design and Implementation with Java and UML", OOPSLA 2002.
[37] Korthaus, A., "Using UML for business object based systems modelling", In M. Schader and A. Korthaus, editors, *The Unified Modeling Language - Technical Aspects and Applications*, pp. 220-237, Physica-Verlag, Heidelberg, Germany, 1998.
[38] Object Management Group, "Business Object DTF Common Business Objects", *OMG Document bom/97-12-04 Version 1.5*, 1997.
[39] Kleppe, A., Warmer, J. & Bast, W., *MDA Explained: The Model Driven Architecture: Practice and Promise*, Addison-Wesley, 2003.
[40] Wooldridge, M., Jennings, N.R. & Kinny, D., "A Methodology for Agent-Oriented Analysis and Design", *Proceedings of the 3rd International Conference on Autonomous Agents (Agents-99)*, pp. 69 - 76, 1999.
[41] Wooldridge, M., Jennings, N.R. & Kinny. D., "The Gaia methodology for agent-oriented analysis and design", *Journal of Autonomous Agents and Multi-Agent Systems* 3(3): 285-312, 2000.
[42] Jennings, N.R., Sycara, K. & Wooldridge, M., "A Roadmap of Agent Research and Development", *Autonomous Agents and Multi-Agent Systems* 1(1): 7-38, 1998.
[43] Wasserman, A.I., "Toward a Discipline of Software Engineering", *IEEE Software* 13(6): 23-31, 1996.
[44] Pfleeger, S.L. & Atlee, J.M., *Software engineering : theory and practice, third edition*, Prentice Hall, 2005.
[45] Jennings, N.R., Norman, T.J., Faratin, P., O'Brien, P. & Odgers, B., "Autonomous Agents for Business Process Management", *International Journal of Applied Artificial Intelligence* 14(2): 145-189, 2000.
[46] Rai, V.K. & Anantaram, C., "Structuring business rules interactions", *Electronic Commerce Research and Applications* 3(1): 54-73, 2004.
[47] Rosca, D., Feblowitz, M. & Wild, C., "Decision Making Methodology in Support of the Business Rules Lifecycle", *Proceedings of the 3rd IEEE International Symposium on Requirements Engineering (RE'97)*, p.236, January 05-08, 1997
[48] Date, C.J., *What Not How: The Business Rules Approach to Application Development*, Addison-Wesley, 2000.
[49] Papamarkos, G., Poulovassilis, A. & Wood, P.T., "Event-Condition-Action Rule Languages for the Semantic Web", Workshop on Semantic Web and Databases, at VLDB'03, Berlin, September 2003.
[50] Paton, N.W. (Ed.), *Active Rules in Database Systems*, Springer, New York 1999.
[51] Gatziu, S. & Dittrich, K.R., "Events in an active object-oriented database system", *Proceedings of the 1st International Workshop on Rules in Database Systems*, Springer, 1993.
[52] Iglesias, C.A., Garijo, M. & Gonzalez, J.C., "A survey of agent-oriented methodologies", In M¨uller, J.P., Singh, M.P. & Rao, A.S., editors, *Intelligent Agents V - Proceedings of the Fifth International Workshop on Agent Theories, Architectures, and Languages (ATAL'98)*, LNAI, Springer, 1999.
[53] Odeh, M. & Kamm, R., "Bridging the gap between business models and system models", *Information and Software Technology* 45(15): 1053-1060, 2003.
[54] Kruchten, P., *The Rational Unified Process: an Introduction, second edition*, Addison-Wesley, Reading, MA, 2000.
[55] Jacobson, I. & Bylund, S., *The Road to the Unified Software Development Process*, Cambridge University Press, Cambridge, 2000.
[56] Goh, A., Koh, Y.-K. & Domazet, D.S., "ECA rule-based support for workflows", *Artificial Intelligence in Engineering* 15(1): 37-46, 2001.
[57] Boinot, P., Marlet, R., Noye, J., Muller, G. & Consel, C., "A declarative approach for designing and developing adaptive components", *Proceedings of the Fifteenth IEEE International Conference on Automated Software Engineering*, p.111, 2000.
[58] Wermelinger, M., Koutsoukos, G., Avillez, R., Gouveia, J., Andrade, L. & Fiadeiro, J.L., "Using Coordination Contracts for Flexible Adaptation to Changing Business Rules", *Proceedings of the 6th Intl. Workshop on Principles of Software Evolution*, pp. 115-120. IEEE Computer Society Press, 2003.

[59] Pollock, J.T. & Hodgson, R., *Adaptive Information: Improving Business Through Semantic Interoperability, Grid Computing, and Enterprise Integration*, Wiley-Interscience, 2004.
[60] Alhir, S.S., "Guide to Applying the UML", Springer, 2002.
[61] Rumbaugh, J., Jacobson, I. & Booch, G., *The Unified Modeling Language Reference Manual*, Addison-Wesley, 1999.
[62] Hogg, J., "Applying UML 2 to Model-Driven Architecture", *Proceedings of OMG Workshops: MDA Implementers' Workshop*, 2003.
[63] Sommerville, I., *Software Engineering - 7th Edition*, Addison-Wesley, 2004
[64] Bruegge, B. & Dutoit, A.H., *Object-Oriented Software Engineering: Using UML, Patterns and Java, Second Edition*, Prentice Hall, 2004.
[65] Tutorial on the Jade Agent Development Framework (JADE), IEEE International Conference on Systems, Man and Cybernetics (SMC), 2004.
[66] McLaughlin, B., *Java & XML Data Binding*, O'Reilly, 2002.
[67] Haas, H. & Brown, A. (Ed.), W3C Working Group Note, 11 February 2004.
[68] Smith, H., "Business process management - the third wave: business process modelling language (bpml) and its pi-calculus foundations", *Information and Software Technology* 45(15): 1065-1069, 2003.
[69] Petrie, C. & Bussler, C., "Service Agents and Virtual Enterprises: A Survey", *IEEE Internet Computing* 7(4): 68-78, 2003.
[70] McKinley, P.K., Sadjadi, S.M., Kasten, E.P. & Cheng, B.H.C., "Composing Adaptive Software", *IEEE Computer* 37(7): 56-64, 2004.
[71] Kiczales, G., et al., "Aspect-Oriented Programming", *Proceedings of the European Conference Object-Oriented Programming (ECOOP)*, pp. 220-242, LNCS 1241, Springer, 1997.
[72] Garlan, D., Cheng, S.W., Huang, A.C., Schmerl, B. & Steenkiste, P., "Rainbow: Architecture-Bases Self-Adaptation with Reusable Infrastructure", *IEEE Computer* 37(10): 46-54, 2004.
[73] JADE - Java Agent DEvelopment Framework, http://jade.tilab.com/.
[74] Coronato, A., d'Acierno, A. & Pietro, G.D., "Automatic implementation of constraints in component based applications", *Information and Software Technology* 47(7): 497-509, 2005.
[75] Wan-Kadir, W.M.N. & Loucopoulos, P., "Relating evolving business rules to software design", *Journal of Systems Architecture* 50(7): 367-382, 2004.
[76] Molina, J.G., Ortín, M.J., Moros, B., Nicolás, J. & Toval, A., "Towards Use Case and Conceptual Models through Business Modeling", *Proceedings of 19th International Conference on Conceptual Modeling (ER'2000)*, Salt Lake City, USA, 2000.
[77] Korson, T., "The Misuse of Use Cases", *Object Magazine* 8(3): 18-20, 1998.
[78] Dikel, D.M., Kane, D. & Wilson, J.R., *Software Architecture: Organizational Principles and Patterns*, Prentice Hall, 2001
[79] Oreizy, P., Medvidovic, N. & Taylor, R.N., "Architecture-Based Runtime Software Evolution", *Proceedings of the 20th International Conference on Software Engineering (ICSE'98)*, Kyoto, Japan, 1998
[80] Warmer, J. & Kleppe, A., *The Object Constraint Language: Getting Your Models Ready for MDA, Second Edition*, Addison-Wesley, 2003.
[81] Foundation for Intelligent Physical Agents, http://www.fipa.org/
[82] Bölöni, L., Khan, M.A., Bai, X., Wang, G., Ji, Y. & Marinescu, D.C., "Software Engineering Challenges for Mutable Agent Systems", SELMAS 2003, pp. 149-166, LNCS 2940, Springer, 2004.
[83] Liu, K., Sun, L. & Bennett, K., "Co-Design of Business and IT Systems - Introduction by Guest Editors", *Information Systems Frontiers* 4(3): 251-256, Kluwer Academic Publishers, 2002.
[84] Stamper, R.K., Althaus, K., Backhouse, J., "MEASUR: Method for eliciting, analyzing and specifying user requirements", in *Comupterized Assistance During the Information Systems Life Cycle*, Olle, T.W., Verrijn-Stuart, A.A. & Bhabuts, L., eds., pp. 67-115, North-Holland: Elsevier Science Publishers, 1988.
[85] Sommerville, I., "Software Construction by Configuration: Challenges for Software Engineering Research", *Proceedings of the 21st IEEE International Conference on Software Maintenance (ICSM'05)*, p. 9, 2005.
[86] Sommerville, I., "Construction by Configuration: A New Challenge for Software Engineering Education", invited lecture, JENUI'05, Spain, 2005.
[87] Sommerville, I., "Integrated Requirements Engineering: A Tutorial", *IEEE Software* 22(1): 16-23, 2005.
[88] Zhang, Q. & Zou, Y., "Using Self-Reconfigurable Workplaces to Automate the Maintenance of Evolving Business Applications", *Proceedings of the 21st IEEE International Conference on Software Maintenance (ICSM'05)*, pp. 219-229, 2005.
[89] Lock, S., "Strider: Configuration Modelling and Analysis of Complex Systems", *Proceedings of the 21st IEEE International Conference on Software Maintenance (ICSM'05)*, pp. 495-504, 2005.
[90] Anderson, P. & Scobie, A., "LCFG: the Next Generation", UKUUG Winter Conference, UK, 2002.
[91] Bass, L., Clements, P. & Kazman, R., *Software Architecture in Practice (2nd ed.)*, Addison-Wesley, 2003.
[92] Clements, P., Bachmann, F., Bass, L., Garlan, D., Ivers, J., Little, R., Nord, R. & Stafford, J., *Documenting Software Architectures: Views and Beyond*, Addison-Wesley, 2002.
[93] Garlan, D., Monroe, R. & Wile, D., "ACME: An Architecture Description Interchange Language", *Proceedings of CASCON'97*, pp. 169-183, Toronto, 1997.

[94] Kagdi, H., Maletic, J.I. & Sutton, A., "Context-Free Slicing of UML Class Models", *Proceedings of 21st IEEE International Conference on Software Maintenance (ICSM'05)*, pp. 635-638, 2005.
[95] Bennett, K.H. & Xu, J., "Software services and software maintenance", *Proceedings of the Seventh European Conference on Software Maintenance and Reengineering (CSMR'03)*, pp 3-12, 2003.
[96] Lin, L., Embury, S.M. & Warboys, B.C., "Facilitating the Implementation and Evolution of Business Rules", *Proceedings of the 21st IEEE International Conference on Software Maintenance (ICSM'05)*, pp. 609-612, 2005.
[97] Aversano, L., Bodhuin, T. & Tortorella, M., "Assessment and impact analysis for aligning business processes and software systems", *Proceedings of the 2005 ACM Symposium on Applied Computing (SAC'05)*, pp. 1338-1343, 2005.
[98] Jansen, S., Brinkkemper, S., Ballintijn, G. & Nieuwland, A.V., "Integrated Development and Maintenance of Software Products to Support Efficient Updating of Customer Configurations: A Case Study in Mass Market ERP Software", *Proceedings of the 21st IEEE International Conference on Software Maintenance (ICSM'05)*, pp. 253-262, 2005.
[99] Jin, D. & Cordy, J.R., "Ontology-Based Software Analysis and Reengineering Tool Integration: The OASIS Service-Sharing Methodology", *Proceedings of the 21st IEEE International Conference on Software Maintenance (ICSM'05)*, pp. 613-616, 2005.
[100] Zhao, W., Zhang, L., Mei, H. & Sun, J., "Requirements Guided Dynamic Software Clustering", *Proceedings of the 21st IEEE International Conference on Software Maintenance (ICSM'05)*, pp. 605-608, 2005.
[101] Parikh, G., "Software Support, Management, and Evolution (SSME) in the Coming Decade and Beyond...Opportunities and Challenges", *Proceedings of the 21st IEEE International Conference on Software Maintenance (ICSM'05)*, pp. 10-11, 2005.
[102] Maciaszek, L.A., "Developing Supportable Enterprise Information Systems - Architectural, Managerial, and Engineering Imperatives", *Proceedings of the 21st IEEE International Conference on Software Maintenance (ICSM'05)*, pp. 721-722, 2005.
[103] Demeyer, S., Ducasse, S. & Nierstrasz, O., "Object-Oriented Reengineering: Patterns and Techniques", *Proceedings of the 21st IEEE International Conference on Software Maintenance (ICSM'05)*, pp. 723-724, 2005.
[104] Lehman, M.M., "Laws of software evolution revisited", In Montangero, C., editor, *Software Process Technology - Proceedings of the 5th European Workshop*, pp. 108-124, LNCS 1149, Springer, 1996.
[105] Chapin, N., "Continuous Evolution: Practices and Issues", *Proceedings of the 21st IEEE International Conference on Software Maintenance (ICSM'05)*, p. 717, 2005.
[106] Chapin, N., Hale, J., Khan, K., Ramil, J. & Than, W., "Types of Software Evolution and Software Maintenance", *Journal of Software Maintenance: Research and Practice* 13(1):3-30, John Wiley & Sons, Inc., 2001.
[107] Kotonya, G. & Hutchinson, J., "Managing Change in COTS-Based Systems", *Proceedings of the 21st IEEE International Conference on Software Maintenance (ICSM'05)*, pp. 69-78, 2005.
[108] The Institute of Electrical and Electronics Engineers, "IEEE standard for software maintenance", IEEE Std 1219-1998, 1998.
[109] Lovrek, I., Jezic, G., Kusek, M., Ljubi, I., Caric, A., Huljenic, D., Desic, S. & Labor, O., "Improving Software Maintenance by using Agent-based Remote Maintenance Shell", *Proceedings of the 19th IEEE International Conference on Software Maintenance (ICSM'03)*, p. 440, 2003.
[110] Kwon, O.B. & Lee, J.J., "A multi-agent intelligent system for efficient ERP maintenance", *Expert Systems with Applications* 21(4): 191-202, 2001.
[111] Murphy, S.N., Rabbani, U.H. & Barnett, G.O., "Using software agents to maintain autonomous patient registries for clinical research", *AMIA Annual Fall Symposium Proceedings*, pp. 71-75, 1997.
[112] Lehman, M.M. & Ramil, J.F., "Software evolution—Background, theory, practice", *Information Processing Letters* 88(1-2): 33-44, 2003.
[113] Manna, M., "Maintenance Burden Begging for a Remedy", Datamation, pp. 53-63, 1993.
[114] Lieberherr, K., "Workshop on Adaptable and Adaptive Software", *Proceedings of the Tenth Conference on Object Oriented Programming Systems Languages and Applications*, pp. 149-154, 1995.
[115] Bosch, J., "Design Patterns as Language Constructs", *Journal of Object Oriented Programming* 11(2): 18-32, 1998.
[116] Seiter, L., Palsberg, J. & Lieberherr, K., "Evolution of Object Behavior Using Context Relations", *IEEE Transactions on Software Engineering* 24(1): 79-92, 1998.
[117] Schultz, U., Lawall, J. & Consel, C., "Specialization Patterns", *Proceedings of the 15th IEEE International Conference on Automated Software Engineering*, p. 197, 2000.
[118] Dantas, A. & Borba, P., "Adaptability Aspects: An Architectural Pattern for Structuring Adaptive Applications", Third Latin American Conference on Pattern Languages of Programming, SugarLoafPLoP'2003.
[119] Dantas, A., Yoder, J., Borba, P. & Johnson, R., "Using Aspects to Make Adaptive Object-Models Adaptable", ECOOP'2004 Workshop on Reflection, AOP and Meta-Data for Software Evolution.
[120] Zambonelli, F. & Omicini, A., "Challenges and research directions in agent-oriented software engineering", *Autonomous Agents and Multi-Agent Systems* 9(3): 253-283, 2004.
[121] Zambonelli, F., Jennings, N.R. & Wooldridge, M.J., "Developing Multiagent Systems: the Gaia Methodology", *ACM Transactions on Software Engineering and Methodology* 12(3): 317-370, 2003.
[122] DeLoach, S.A., Wood, M.F. & Sparkman, C.H., "Multiagent Systems Engineering", *International Journal on Software Engineering and Knowledge Engineering* 11(3): 231-258, 2001.

[123] Castro, J., Kolp, M. & Mylopoulos, J., "Towards Requirements-Driven Information Systems Engineering: The Tropos Project", *Information Systems* 27(6): 365-389, 2002.
[124] Bergenti, F., Gleizes, M. & Zambonelli, F., "Methodologies and Software Engineering for Agent Systems: The Agent-Oriented Software Engineering Handbook", Kluwer, 2004.
[125] Kinny, D., Georgeff, M. & Rao, A., "A methodology and modelling technique for systems of BDI agents", *Agents Breaking Away: Proceedings of the Seventh European Workshop on Modelling Autonomous Agents in a Multi-Agent World*, MAAMAW'96, Velde, W. & Perram, J. (Eds.), LNAI 1038, Springer, 1996.
[126] Cossentino, M., Burrafato, P., Lombardo, S. & Sabatucci, L., "Introducing Pattern Reuse in the Design of Multi-Agent Systems", AITA'02 workshop at NODe02, 2002.
[127] Arai, T. & Stolzenburg, F., "Multiagent systems specification by UML statecharts aiming at intelligent manufacturing", *Proceedings of the First International Joint Conference on Autonomous Agents & Multi-Agent Systems*, pp. 11-18, 2002.
[128] Lotzsch, M., Bach, J., Burkhard, H.-D. & Jungel, M., "Designing Agent Behavior with the Extensible Agent Behavior Specification Language XABSL", *RoboCup 2003: Robot Soccer World Cup VII*, pp. 114-124, LNAI 3020, Springer, 2004.
[129] Bauer, B., Muller, J.P. & Odell, J., "Agent UML: A Formalism for Specifying Multiagent Software Systems", *International Journal of Software Engineering and Knowledge Engineering* 11(3): 207-230, 2001.
[130] Odell, J., Parunak, H. & Bauer, B., "Representing Agent Interaction Protocols in UML", *Agent-Oriented Software Engineering: First International Workshop (AOSE 2000),* Ciancarini, P. & Wooldridge, M. (Eds.), pp. 121–140, LNCS 1957, Springer, 2001.
[131] AUML web site, http://www.auml.org/
[132] Object Management Group, Inc., "Agent Technology Green Paper", *OMG Document agent/00-09-01 Version 1.0*, 250 First Ave, Suite 201, Needham, MA 02494, USA, 2000.
[133] Silva, V., Choren, R. & Lucena, C., "Using the MAS-ML to Model a Multi-Agent System", *Software Engineering for Multi-Agent Systems II*, Lucena, C. et al. (Eds.): SELMAS 2003, pp. 129-148, LNCS 2940, Springer, 2004.
[134] JADE platform, http://jade.tilab.com/.
[135] Beck, K., *Extreme Programming Explained: Embrace Change*, Addison-Wesley, Boston, 2000.
[136] Ambler, W.S. & Constantine, L.L., *The Unified Process Inception Phase: Best Practices in Implementing the UP*, R&D, 2000.
[137] Object Management Group, Inc., 250 First Ave. Suite 100, Needham, MA 02494, USA.
[138] Rosca, D. & Wild, C., "Towards a flexible deployment of business rules", *Expert Systems with Applications* 23(4): 385-394, 2002.
[139] Liu, W., Easterbrook, M.S. & Mylopoulos, J., "Rule-based detection of inconsistency in UML models", Workshop on Consistency Problems in UML-Based Software Development, 5th International Conference on the Unified Modeling Language, Dresden, Germany, 2002.
[140] Gimenez-Lugo, G., Sichman, J.S. & Hubner, J.F., "Integrating knowledge centered MAS through organizational links", *Proceedings of the International Conference on Integration of Knowledge Intensive Multi-Agent Systems*, pp. 237-242, 2003.
[141] Chandrasekaran, B., Josephson, J.R. & Benjamins, V.R., "What Are Ontologies, and Why Do We Need Them?", *IEEE Intelligent Systems* 14(1): 20-26, 1999.
[142] Fowler, M., "Using Metadata", *IEEE Software* 19(6): 13-17, 2002.
[143] Bohrer, K.A., "Architecture of the San Francisco Frameworks", *IBM Systems Journal* 37(2): 156-169, 1998.
[144] Johnson, B., Woolfolk, W., Miller, R. & Johnson, C., *Flexible Software Design: Systems Development for Changing Requirements*, Auerbach Publications, 2005.
[145] Wand, Y. & Weber, R., "Mario Bunge's ontology as a formal foundation for information systems concepts", in Weingartner, P. & Dorn, G.J.W. (Eds.), *Studies on Mario Bunge's Treatise*, Rodopi, pp. 123-149, 1990.
[146] Luftman, J., Papp, R. & Brier, T., "Enablers and inhibitors of business-IT alignment", *Communications of the AIS* 1(3es): 1-33, Association for Information Systems, 1999.
[147] Neill, C.J. & Laplante, P.A., "Requirements engineering: the state of the practice", *IEEE Software* 20(6): 40-45, 2003.
[148] Keen, P.W., *Every manager's guide to information technology: a glossary of key terms and concepts for today's business leader*, 2nd ed., Harvard Business School Press, Cambridge, MA, 1995.
[149] Yourdon, E., Whitehead, K., Thomann, J., Oppel, K. & Nevermann. P., *Mainstream Objects: An Analysis and Design Approach for Business*, Prentice Hall, 1995.
[150] Dumas, M., Aalst, W. & Hofstede, A., *Process-Aware Information Systems: Bridging People and Software through Process Technology*, Wiley-Interscience, 2005.
[151] Bourque, P., Dupuis, R., Abran, A., Moore, J.W. & Tripp, L.L., *Guide to the Software Engineering Body of Knowledge: 2004 Edition – SWEBOK*, IEEE Computer Society, 2005.
[152] Sutherland, J., "The Emergence of a Business Object Component Architecture", Patel, D., Sutherland, J., Miller, J., (Eds.), *Business Object Design and Implementation III: OOPSLA'99 Workshop Proceedings*, Springer, 1999.
[153] Herzum, P. & Sims, O., *Business Component Factory*, John Wiley & Sons, 1999.
[154] Hendryx, S., "Introduction to BRWG", *OMG Document ad/02-07-01*, 2002.

[155] Rule Interchange Format Working Group Charter, http://www.w3.org/2005/rules/wg/charter, W3C, 2005.
[156] Vincent, P., "OMG Production Rule Representation - Context and Current Status", W3C Workshop on Rule Languages for Interoperability, 2005.
[157] Hung, K., Simons, T. & Cvetkovic, S., "A Dynamic Business Object Architecture for Supporting Strategic Management Planning", Patel, D., Sutherland, J., Miller, J., (Eds.), *Business Object Design and Implementation II: OOPSLA'96, OOPSLA'97, and OOPSLA'98 Workshop Proceedings*, Springer, 1998.
[158] Rostal, P., "From Business Objects toward Adaptive Agents", Patel, D., Sutherland, J., Miller, J., (Eds.), *Business Object Design and Implementation II: OOPSLA'96, OOPSLA'97, and OOPSLA'98 Workshop Proceedings*, Springer, 1998.
[159] Nakamura, H. & Johnson, R., "Adaptive Framework for the REA Accounting Model", Patel, D., Sutherland, J., Miller, J., (Eds.), *Business Object Design and Implementation II: OOPSLA'96, OOPSLA'97, and OOPSLA'98 Workshop Proceedings*, Springer, 1998.
[160] Digre, T., "Business Object Component Architecture", *IEEE Software* 15(5): 60-69, 1998.
[161] Wagner, G., "A UML Profile for Agent-Oriented Modeling", *Proceedings of the Third International Workshop on Agent-Oriented Software Engineering (AOSE'02)*, Autonomous Agents and Multi-Agent Systems (AAMAS'02), Springer LNCS 2585, 2003.
[162] Xiao, L. & Greer, D., "A Hierarchical Agent-oriented Knowledge Model for Multi-Agent Systems", accepted for publication in *Proceedings of the Eighteenth International Conference on Software Engineering and Knowledge Engineering (SEKE'06)*, 2006.
[163] The Institute of Electrical and Electronics Engineers, *IEEE recommended practice for software requirements specifications*, IEEE Std 830-1998, 1998.
[164] Damian, D., Jonker, C., Treur, J. & Wijngaards, N., "Integration of behavioural requirements specification within compositional knowledge engineering", *Knowledge-Based Systems* 18(7): 353-365, 2005.
[165] Griss, M., Fonseca, S., Cowan, D. & Kessler, R., "Using UML State Machine Models for More Precise and Flexible JADE Agent Behaviors", *Proceedings of the Third International Workshop on Agent-Oriented Software Engineering (AOSE'02)*, pp. 113-125, Springer LNCS 2585, 2003.
[166] Lano, K., Fiadeiro, J. & Andrade, L., *Software Design Using Java 2*, Palgrave Macmillan, 2002.
[167] Wooldridge, M., "Agent-based software engineering", *IEE Proceedings on Software Engineering* 144(1): 26-37, 1997.
[168] Jennings, N. & Wooldridge, M., "Agent-Oriented Software Engineering", Handbook of Agent Technology, Bradshaw, J., (ed.), AAAI/MIT Press, 2000.
[169] Jennings, N., "On agent-based software engineering", Artificial Intelligence 117(2): 277-296, 2000.
[170] Wooldridge, M. & Ciancarini, P., "Agent-Oriented Software Engineering: The State of the Art", *Agent-Oriented Software Engineering: First International Workshop (AOSE'2000),* Ciancarini, P. & Wooldridge, M. (Eds.), LNCS 1957, Springer, 2001.
[171] Etzioni, O., "Moving up the information food chain: Deploying softbots on the world-wide web", *Proceedings of the Thirteenth National Conference on Artificial Intelligence (AAAI'96)*, 1996.
[172] Rao, A. & Georgeff, M., "BDI Agents: from theory to practice", *Proceedings of the First International Conference on Multi-Agent Systems (ICMAS'95)*, pp. 312-319, 1995.
[173] Rao, A. & Georgeff, M., "Modeling rational agents within a BDI-architecture", *Proceedings of Knowledge Representation and Reasoning (KR&R'91)*, Fikes, R. & Sandewall, E., (Eds.), pp. 473-484, Morgan Kaufmann Publishers, 1991.
[174] Rao, A. & Georgeff, M., "Formal models and decision procedures for multi-agent systems", *Technical Note 61*, Australian AI Institute, Level 6, 171 La Trobe Street, Melbourne, Australia, 1995.
[175] Chellas, B., *Modal Logic: An Introduction*, Cambridge University Press, 1980.
[176] Wooldridge, M. & Jennings, N., "Intelligent agents: Theory and practice", *The Knowledge Engineering Review* 10(2):115-152, 1995.
[177] Luck, M. & d'Inverno, M., "A formal framework for agency and autonomy", *Proceedings of the First International Conference on Multi-Agent Systems (ICMAS'95)*, pp. 254-260, 1995.
[178] Luck, M., Griffiths, N. & d'Inverno, M., "From agent theory to agent construction: A case study", *Intelligent Agents III — Proceedings of the Third International Workshop on Agent Theories, Architectures, and Languages (ATAL'96)*, Springer, 1997.
[179] d'Inverno, M. & Luck, M., *Understanding Agent Systems*, Springer, 2001.
[180] Huget, M., "Agent UML Notation for Multiagent System Design", *IEEE Internet Computing* 8(4): 63-71, 2004.
[181] Odell, J. Parunak, H. & Bauer, B., "Extending UML for Agents", *Proceedings of the Agent-Oriented Information Systems Workshop, 17th National Conference on Artificial Intelligence,* Wagner, G., Lesperance, Y. & Yu, E. (Eds.), iCue Publishing, 2000.
[182] Bauer, B., "UML Class Diagrams Revisited in the Context of Agent-Based Systems", *Proceedings of the Agent-Oriented Software Engineering,* LNCS 2222, Wooldridge, M., Ciancarini, P. & Weiss, G. (Eds.), pp. 1-8, Springer, 2001.
[183] Foundation for Intelligent Physical Agents, "FIPA Modeling: Interaction Diagrams", 2003.
[184] Tveit, A., "A survey of Agent-Oriented Software Engineering", *Proceedings of the First NTNU CSGS Conference*, Norwegian University of Science and Technology, 2001.

[185] Yu, E., "Agent-Oriented Modelling: Software Versus the World", *Proceedings of the Second International Workshop on Agent-Oriented Software Engineering (AOSE'01)*, pp. 206-225, LNCS 2222, Springer, 2002.
[186] Odell, J., Nodine, M. & Levy, R., "A Metamodel for Agents, Roles, and Groups", *Proceedings of the Fifth International Workshop on Agent-Oriented Software Engineering (AOSE'04)*, Odell, J., Giorgini, P. & Müller, J. (Eds.), pp. 78-92, LNCS 3382, Springer, 2005.
[187] Shaw, M. & Garlan, D., *Software Architecture: Perspectives on an Emerging Discipline*, Prentice Hall, 1996.
[188] Object Management Group, Inc., "CORBA 3.0 - IDL Syntax and Semantics chapter", *OMG Document formal/02-06-07*, 250 First Ave., Needham, MA 02494, USA, 2002.
[189] Letier, E. & Lamsweerde, A., "Agent-based Tactics for Goal-Oriented Requirements Elaboration", *Proceedings of the 24th International Conference on Software Engineering (ICSE'02)*, pp. 83-93, ACM Press, 2002.
[190] Lamsweerde, A., "Requirements Engineering in the Year 00: A Research Perspective", *Proceedings of the 22nd International Conference on Software Engineering (ICSE '00)*, pp. 5-19, 2000.
[191] Lamsweerde, A., "Goal-Oriented Requirements Engineering: A Guided Tour", *Proceedings of the Fifth IEEE International Symposium on Requirements Engineering (RE'01)*, pp249-262, 2001.
[192] Kristensen, B., "Architectural Abstractions and Language Mechanisms", *Proceedings of the Third Asia-Pacific Software Engineering Conference (APSEC'96)*, p. 288, IEEE Computer Society, 1996.
[193] Bock, C., "UML 2 Activity and Action Models", *Journal of Object Technology* 2(4): 43-53, 2003.
[194] Eberhart, A., "Towards semantically enriched business logic", *Electronic Commerce Research and Applications* 2(4): 288-301, 2003.
[195] Glass, R., Vessey, I. & Ramesh, V., "Research in software engineering: an analysis of the literature", *Information and Software Technology* 44(8): 491-506, 2002.
[196] Grosof, B., Labrou, Y. & Chan, H., "A Declarative Approach to Business Rules in Contracts: Courteous Logic Programs in XML", *Proceedings of the 1st ACM Conference on Electronic Commerce (EC'99)*, Wellman, M. (Ed.), pp. 68-77, 1999.
[197] Guessoum, Z., "Adaptive agents and multiagent systems", *IEEE Distributed Systems Online* 5(7), 2004.
[198] Liu, L. & Yu, E., "From Requirements to Architectural Design - Using Goals and Scenarios", *Proceedings of the ICSE'01 (STRAW'01)*, pp. 22-30, 2001.
[199] Michiels, C., Snoeck, M., Lemahieu, W., Goethals, F., Dedene, G., "A Layered Architecture Sustaining Model-Driven and Event-Driven Software Development", pp. 58-65, LNCS 2890, Springer, 2003.
[200] Thomas, D., "MDA: Revenge of the Modelers or UML Utopia?", *IEEE Software* 21(3): 15-17, 2004.
[201] Meservy, T. & Fenstermacher, K., "Transforming Software Development: An MDA Road Map", *IEEE Computer* 38(9): 52-58, 2005.
[202] Fayad, M. & Cline, M., "Aspects of Software Adaptability", *Communications of the ACM* 39(10):58-59, 1996.
[203] Floch, J., Hallsteinsen, S., Stav, E., Eliassen, F., Lund, K. & Gjorven, E., "Using Architecture Models for Runtime Adaptability", *IEEE Software* 23(2): 62-70, 2006.
[204] Durn, A., Bernrdez, B., Ruiz, A. & Toro, M., "A Requirements Elicitation Approach Based in Templates and Patterns", *Proceedings of the WER'99*, 1999.
[205] Zhu, H., "SLABS: A Formal Specification Language for Agent-Based Systems", *International Journal of Software Engineering and Knowledge Engineering* 11(5): 529-558, 2001.
[206] Zhu, H., "The role of caste in formal specification of MAS", *Proceedings of the PRIMA'2001*, pp. 1-15, LNCS 2132, Springer, 2001.
[207] Zhu, H. & Lightfoot, D., "Caste: A step beyond object orientation", *Modular Programming Languages - Proceedings of the JMLC'2003*, Boszormenyi, L. & Schojer, P. (Eds.), pp. 59-62, LNCS 2789, Springer, 2003.
[208] Manifesto for Agile Software Development, http://agilemanifesto.org/, accessed on September 14, 2006.
[209] Henderson, P., *Systems Engineering for Business Process Change*, Springer, 2000.
[210] Xiao, L. & Greer, D., "Adaptive Agents in Java", submitted to *Science of Computer Programming*, 2006.
[211] Basili, V.R., Caldiera, G. & Rombach, H.D., "The Goal Question Metric Approach", *Encyclopaedia of Software Engineering*, 1994.
[212] Electronic Business using eXtensible Markup Language, http://www.ebxml.org/, accessed on September 14, 2006.
[213] Business Process Management Initiative, http://www.bpmi.org/, accessed on September 14, 2006..
[214] Business Process Execution Language for Web Services, http://www-128.ibm.com/developerworks/library/specification/ws-bpel/, accessed on September 14, 2006.
[215] Web Services Choreography Description Language, http://www.w3.org/TR/2004/WD-ws-cdl-10-20041217/, accessed on September 14, 2006.
[216] World Wide Web Consortium (W3C), www.w3.org.