Agent-oriented Requirements Modelling

Liang Xiao and Des Greer
Queen's University Belfast, Belfast, UK
email: { l.xiao, des.greer }@qub.ac.uk

Abstract

We describe how reconstructing requirements specification using agents and UML helps to build a hierarchy of requirements knowledge which can then be transformed into a running agent system. With agents as the division units for organising functional requirements structurally, the relationship between different pieces of the requirements document appears more obvious and, checks for completeness and consistency are easier to perform. The approach has four steps: i) Goal Mapping, where we relate functional requirements to system goals; ii) Structural Modelling, where agents are identified and rules for these agents are specified; iii) Behavioural Modelling, where the business processes are documented in UML and XML, aiming at realising the system goals by participating agents; iv) Implementation, where a running agent system is automatically generated from the transformed requirements document. We illustrate the approach and its tool support using a real life case study. Using the knowledge from this, we discuss and evaluate the approach.

1. Introduction

1.1. The demand for a new requirements/design modelling approach

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 requirements models, being largely textual descriptions of system functions, are separated from the design models, which lack the capability to describe behavioural semantics exactly [1]. This is a major limitation of traditional software system development. The models rapidly lose their value as soon as coding starts. This loss in value is compounded by the fact that, in practice, changes are often done at the code level only. Thus, the connection between the models and the code fades away as the coding phase progresses [2]. This situation has been recognised in Extreme Programming (XP) [3] 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* [4]. 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.

In this paper we assume that both requirements and design models are useful and seek to increase their utility by integrating them. We propose that a good structure for the requirements specification is one that facilitates a clear and comprehensible division into modules, models the system diagrammatically, captures the knowledge on system functions in detail and links them to the elements in the design models. In addition, the integrated models should be easily adaptable (or adaptive) and changes easily checked for consistency and sensibleness without regard to a particular implementation environment. Most useful would be the ability to transform the models into actual software products. In this case, the modelling becomes the primary task in software development, and time spent on it will never be wasted.

1.2. MDA, a potential solution?

The value of system models is increased, if they can be converted to executable software. Model Driven Architecture (MDA) [2] [5] is an up and coming area that seeks this goal. In MDA, models are central rather than an overhead in the development process. 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). 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 [2] 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.

Recognising that UML alone is not able to capture some semantics in its diagrams [6], a combination of UML and OCL [5] is used in MDA. However, OCL constraints are static and are external 'add-ons' to UML [7] and used in the design stages rather than the requirements stages.

1.3. Agent-oriented modelling

Adopting the objectives of MDA, we propose a modelling paradigm starting from constructing requirements models that are re-useable throughout the software development phases. Such models, in the first place, capture the overall system structure with individual functional requirements captured in the model elements. The requirements models are built from the transformation of the original requirements document in a systematic way, and they in turn can be transformed into the final software products. Supporting tools that can recognise this structured requirements and the semantics of its division units are provided to assist the transformation. The approach is agent-oriented, goal-guided, and rule-based. Systems are documented in UML and XML. We illustrate the approach using a real case study concerning the management of a rail track network.

2. Background

2.1. Business Rules – the missing piece in UML

According to the Object Management Group [5], business rules are "declarations of policy or conditions that must be satisfied". In [6], it is stated that they capture functional requirements, the decisions, guidelines and controls behind the functionality, and are the true essence of functional requirements. Also, 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 in the requirements level [8] and in the design level [9]. 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.

Although UML, accompanied with OCL [4] [5] [7], is able to specify constraint rules on its model elements, these rules are restricted to the design. 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 at the centre of the UML and can have business rules linked to them. Adding business rules to use cases provides better structured requirements than the traditional use case narratives. The use of business rules can compensate for the lack of capability of existing UML diagrams for capturing behavioural semantics.

It is suggested in [6] that business rules have the form of terms, facts, factor clauses and action clauses, expressed in natural language using a tailored taxonomy. 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.

2.2. Agents – execution engine of business rules

Business rules exist only to support business goals [6]. Therefore, the validity of a rule can be checked for mapping to a business goal. Since business rules are constraints about what must or must not be the case [4] and capture functional requirements, these rule statements not only constrain but also guide the realisation of business goals. On the other hand, groups of cooperative software agents interact with each other towards common goals [10]. Thus *agents* and *business rules* complement each other in the implementation: the former is the actor, the latter guides it and dictates the role that it plays. They work together to achieve business goals. On the Requirements Engineering (RE) level, an agent is the container and a business rule is the knowledge to fill the container. The mapping from requirements to implementation for both the agent and the business rule brings the traceability to the original requirements.

The RE level concept of agent has been introduced in [11] and the agent is advocated as a guiding concept, similar to how the object has been central in the OO approach. Some agent-oriented RE frameworks have been put forward. One such is Composite System Design [12]. This uses agents as composite components. 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 agents are enabled with capabilities to behave. Similarly, KAOS [13], a goal-directed RE framework requires designers to refine goals until they are reduced to constraints that can be assigned to agents. The Albert II approach [14] expresses functional requirements using formal statements and groups them around agents to define the admissible behaviour of agents. A commonality of these frameworks is that agents are used as a way for requirements organisation. Requirements for the system feature are collected around agents as the basis to direct agents on how to behave.

2.3. Business Goals –putting agents and business rules in business processes

It is suggested that goal-based reasoning is central to requirements engineering and that goals are also abstractions that stakeholders are familiar with and are interested in [15]. The refinement of goals can help to build a comprehensible requirements structure and combined with agent assignment, alternative system solutions can be explored [16]. Many goal-oriented requirements engineering approaches are also agent-oriented, some of which are listed in section 2.2.

2.4. Business Processes – the organisation unit

The UML version 2 has thirteen official diagram types [1]. However, none of these provides entirely suitable constructs for expressing business goals.

Business process is a main/central construct for business modelling and is closely related with the requirements [4]. It follows that business processes are also closely associated with use cases, the main requirements construct in the UML. Nevertheless use cases are not good at expressing goals, while business processes describe how an organisation carries out a set of discrete but related activities to achieve business goals. Each activity can be accomplished by an agent playing a business role. Consequently, with the aid of the collaboration of all these agents, the goal of the business process is achieved. System requirements therefore can be represented by a set of business processes, each process being one that agents interact within to achieve a particular system goal.

3. Solution

3.1. Meta-model for requirements model

Figure 1 shows a meta-model of our Agent-oriented Requirements Modelling approach. In this approach, a *business process* has a *business goal* [4], *agents* collaborate towards *business goals* and *business rules* support business goals. *Agents*, therefore, have the responsibility of achieving the business goals. Using this terminology we can describe, not only what functional requirements should be together, but also how they function sequentially. Agents collaborate by playing *business roles*, which are, in turn, dictated 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 agents' input and output ports. 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, our 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 stay as internal to their master agents only, invoked by business rules, and invisible to external agents. They are owned by the lower level *business classes*, while conversely business rules stay in a higher level, owned directly by agents. Business classes are managed 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. We will show our requirements modelling approach based on this meta-model in the following sections.

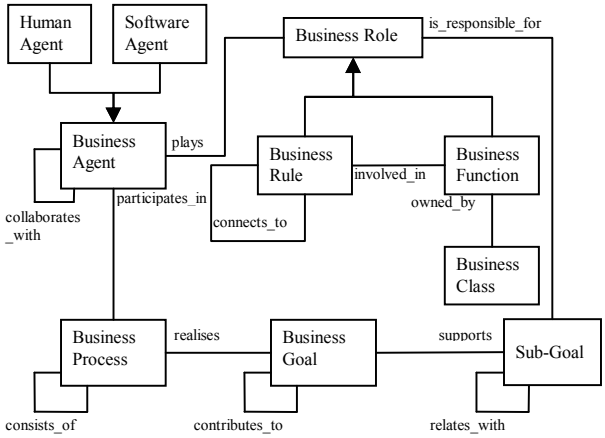

Figure 1. Meta-model of the requirements modelling approach

3.2. Case study

Underpinning the approach is a traditional functional requirements specification. These are usually documented textually, often in a form-based fashion using natural language. One actual requirements document, a national railway track system specification, has been investigated and the appropriateness of our agent-oriented requirements modelling approach assessed. 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 service by conveying issues to relevant parties for resolution. Being a very complex system, the document has more than 250 pages and contains a large number of function descriptions in a unified form as typified in Table 1. Relationship between function tables is not immediately obvious and this makes it difficult to maintain consistency.

Table 1. The original requirements documented in functional requirements tables

Accept Late Addition

Description	To handle a late request for a train journey.
Cause	Receipt of a request for a train journey directly from a **Train Operator** or from the driver entering the production function's area. ***The request is provided in the form of a combination of relevant train details, locations and desired timings***.
Assumption	The crew is competent for the route requested.
Information Used	Relevant locations.
Outputs	A new train journey, to **Train Operator** and others.
Required Effect	A new train journey is created from the request, and validated (**PF.TR.B-ValidateTrainPlan**). If the train journey is acceptable then it is distributed to all interested parties; otherwise the request is rejected or renegotiated. Having been accepted, the new train journey is known to the Production Function.
Impact of Unavailability	The unavailability of this function would render the Production Function unable to respond at short notice to customer requests for additional train paths.
Identifier	PF.TR.B-AcceptLateAddition

Validate Train Plan

Description	To validate a train journey.
Sub-Reqt of	**PF.TR.B-AcceptLateAddition**
Information Used	Train journey and relevant train details, sectional running times, locations and track restrictions.
Required Effect	The Production Function checks that the train journey: • complies with the Rules of the Route and, where relevant, the Rules of the Plan; • does not conflict with other train journeys and planned possessions; and • neither introduces unacceptable disruptions to this or other services nor increase the sensitivity of the Train Plan to disruption beyond "an acceptable level". The Production Function must also balance the needs of all its customers.
Identifier	PF.TR.B-ValidateTrainPlan

Two fundamental models are involved. Firstly, a *structural model*, with the UML Class Diagram as its counterpart in the OO world, is developed for structural relationship modelling. Secondly, a *behavioural model*, with the UML Sequence Diagram as its counterpart in the OO world, is proposed for behavioural interaction modelling. Agents are used to model the conceptual domain units, business rules are used to model the behaviours, and message passing is used to model the interactions.

3.3. Step one: goal mapping

Our approach assumes that the main goals of the system under development are explicitly stated in the user requirements document. We believe that organising functional requirements in terms of their goals can build a hierarchal requirements structure, where those at the bottom support those at the top. Therefore, we use a goal-decomposition technique, similar to [17], to expose all relevant functional requirements tables. Top level requirements are those most valued by the business people and reflect the final business goals. Subordinate goals can be derived using our goal decomposition technique down to the lowest level goals, which map to individual functional requirements. This is necessary because we need to model the business processes in terms of all participating functions in order to later implement those processes. One important business goal documented in the user requirements for the rail track system is to "deliver train journeys safely". Whereas this goal may have been used as the basis to construct multiple functional requirements tables, the link from individual functional requirements may not have been maintained in the requirements document. Figure 2 demonstrates the decomposition process for the goal "deliver train journeys safely".

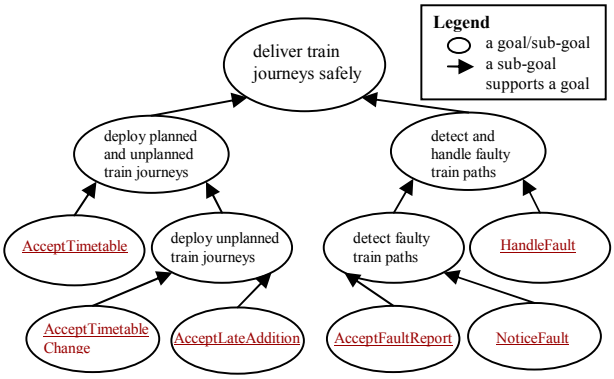

Figure 2. Goal decomposition graph for the case study

This 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 and the latter into the detection and handling of faults. These sub-goals can be further decomposed into smaller sub-goals and considered just like ordinary goals in the subsequent decomposition process. Finally, when the business goal is decomposed into the smallest granularity, the process terminates and we find that all the leaf nodes, underlined and highlighted in the figure, are presented as functional requirements tables. Among these is "AcceptLateAddition" as documented in Table 1. We map one table for the functional requirements to a sub-goal in a leaf node or, alternatively, a combination of such tables to a goal/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", and also to an intermediate goal, as a relatively complex case, such as "detect and handle faulty train paths", if its sub-goals are inter-related. We will only discuss the first case because of limited space.

3.4. Step two: structural modelling, including agent identification and rule transformation

Agents can be actors or information systems, identified in the knowledge domain, aiming at accomplishing the business goals by participating in business processes. To realise the goal "AcceptLateAddition", an independent goal identified in Step One, a business process will be delegated, the detail of which will be presented in the next section. Before that the involved agents should be identified.

In our case study, "AcceptLateAddition" (see Table 1), "AcceptTimetable" and "AcceptTimetableChange" belong to a single "Train Running" business domain. (This is indicated by the shared prefix "TR.B" field in their specification table identifier.) Consequently, they belong to the same agent, which we name "TrainRunning". Note that it is a coincidence that the common goal of, or the one supported by the three, is the same - "deploy planned and unplanned train journeys". In many cases one goal is realised by multiple agents, as a result of cross functional division interactions [17]. Notice that the concept of agent should be distinguished between the requirements modelling and the implementation. Agents in this phase are used to organise the requirements and when implemented as software units they are responsible to meet their corresponding requirements.

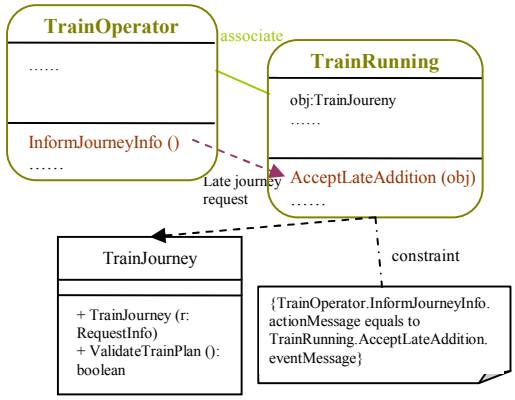

Figure 3. Agent Diagram for case study

The static structure of the system is modelled using the Agent Diagrams, an example of which is shown in Figure 3. The identified agents represent conceptual domains and are used to organise functional requirements tables. This diagram has the UML Class Diagram as its counterpart in OO models, but in our approach, the agent is regarded as the first class citizen rather than the class. In an Agent Diagram, each box that represents an agent is divided into three compartments, just like a class box in a UML Class Diagram. Despite this similarity, they have different content, the compartments in the Agent Diagram being: the name of the agent, the classes managed by the agent along with their instantiation, and rules that govern the functions of the agent.

As the meta-model in Figure 1 shows, to accomplish a sub-goal, an agent in fact plays a role, and that role is responsible for the sub-goal. A role can be either a rule or

a function. In most cases, it would be a rule, which may connect to other rules and have functions involved. For the case study requirement "AcceptLateAddition", a single functional requirements table represents a sub-goal and its owner agent "TrainRunning" is responsible to realise it by playing a role. We can call this role "AcceptLateAddition", the same name as the goal. Also in the case where the role becomes a rule the same name given to it. In contrast would be the case that multiple roles are played to contribute to a higher level goal. The simple goal in our case is to make "late addition" accepted, while the single role/rule involved is defined in the functional requirements table telling the agent how to do it. "ValidateTrainPlan" is involved to assist the rule "AcceptLateAddition" to realise the goal, by receiving some information from the rule, and returning some results back to it. In the case, train journey information is received and a boolean value is returned. Although both are presented as functional requirements table in Table 1, they are distinguished, because "ValidateTrainPlan" has "AcceptLateAddition" in its "Sub-Reqt of " section.

The requirements from the case study captured by the Agent Diagram in Figure 3 are: the agent "TrainRunning" and "TrainOperator" are the only agents involved for the business goal. The two recognised agents represent the "train running planning" domain and the "train operating company" domain respectively. "TrainOperator" has a rule of "InformJourneyInfo" that will construct a "TrainJourney" object, package it into a "Late journey request" message and send the message to the "TrainRunning". This may reflect increasing demand of train journeys during special dates and events in real life. To respond to such requests, "TrainRunning" will plan additional journeys using the rule of "AcceptLateAddition". The required goal of deploying additional train journey is documented in the functional requirements table, which also describes how the goal is achieved by using such a procedure.

During processing of the rule "AcceptLateAddition", a "TrainJourney" object will be constructed according to the journey request information encoded in the message. The constructed object should pass a validation check before being put into use. Therefore it is a necessity that a business class "TrainJourney" is managed by the "TrainRunning" agent and the class has a constructor method and a validation method to be invoked by the agent. "ValidateTrainPlan", presented in Table 1 as one of the functional requirements tables, represents that validation method of the business class. It is involved in the rule "AcceptLateAddition" to assist its function rather than work as an independent rule.

As we can see, such an Agent Diagram can be used to structure the requirements specification with agents, rules, classes and messages and provide details of their relationship. Agents are higher level entities having rules that govern their behaviours. Business classes with functions are in a lower level, used by the agents.

A rule essentially captures a functional requirement in that it lays out action that should be taken, on receipt of an event (normally modelled as a message), if certain conditions are satisfied. This is analogous to the traditional components of function input, function use context, function process and function output. From this basic structure which is shared by all rules, we split each rule into several related compositional parts and each captures one aspect of the requirements for a function.

Figure 4 illustrates the rule specification template, formulated according to functional requirements tables documented in Table 1. A "Cause" section is used to make the rule "event"; its sections of "Information Used" and "Required Effect" are used to make the rule "processing"; its "Required Effect" and "Outputs" are used to make the rule "condition" and "action". Thus events cause agents to execute rules, if certain conditions are satisfied, some actions are triggered which in turn include generated events for other agents. We also conceive "belief" as an integral part of the rule structure. "Belief" is a collection of knowledge that an agent can learn from the messages received from other agents.

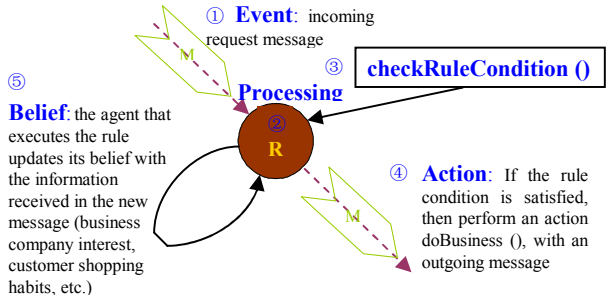

Figure 4. Rule template (with only one pair of {condition, action} considered)

Thus the rule "AcceptLateAddition", transformed from the functional requirements table in Table 1, would be specified as:

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

3.5. Step three: behavioural modelling, including business process specification

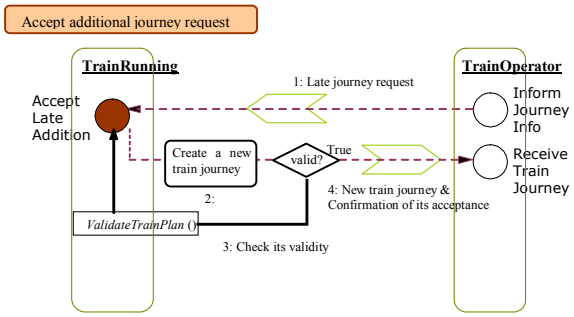

Figure 5. Agent Communication Diagram describing a Business Process Rule (with only the default condition considered)

The agent/rule and class/method identification is completed when the captured requirements are documented in the Agent Diagrams. Agent Diagrams are capable of capturing the structure of related agents and rules, as reflected in the requirements. Like UML Class Diagrams, the Agent Diagrams focus exclusively on structure and ignore behavioural details. Therefore, additional modelling methods are required to represent behaviour [1]. For OO systems, the Sequence Diagram is a commonly used means to capture the behaviour of a scenario, showing the participant objects and the messages that are passed between these objects [1]. Similarly, we need to organise associated agents and their rules to show how their behaviours can finally achieve the goals belonging to the business processes. Thus, we group agents by business goals/processes and each agent can appear in multiple groups using different rules for different purposes. Agents and their rules are organised according to the business processes they participate in. Requirements specification therefore can be divided according to business processes. For each business process, rules that dictate agent behaviours can be documented.

In order to ensure that behavioural models of the requirements are understandable for computers and can facilitate future automation, we introduce the Agent Communication Diagrams as a means to visualise agent behaviour alongside individual rule definitions. Higher level Business Process Rules (BPR) can be formed to specify business processes. They are the division units for organising the specification, well-defined in UML. One such BPR is composed of a collection of lower level ordinary rules, each defined in XML. Tools for the transformation of these models into implementation are developed and described in section 3.6.

A sample Agent Communication Diagram is given in Figure 5. Each BPR defined in the diagram specifies the reaction of all involved agents for their common goal. Related reaction rules are grouped in BPR. Transformed from the traditional requirements specification, each composite rule is composed of four essential parts as discussed earlier. These components are described using four XML tags: <event>, <processing>, <condition>, and <action>. They reflect respectively four steps an agent take to execute a rule in the Agent Communication Diagram, numbered from 1 to 4 (a "belief" component would be added in the future for additional agent intelligence). The XML definition of the rule for this case is shown in Figure 6.

```
- <local-rule>
    <name>AcceptLateAddition</name>
    <business-process>
     Late train journey request handling
    </business-process>
    <owner-agent>TrainRunning</owner-agent>
    <global-variable>
      <name>
        trainJourney
      </name>
      <type>
        TrainJourney
      </type>
    </global-variable>
  - <event>
      <type>receipt of message</type>
    - <message>
        <from>TrainOperator.InformJourneyInfo</from>
        <to>TrainRunning.AcceptLateAddition</to>
        <title>Late journey request</title>
      - <content>
        - <requestInfo>
          - <trainDetail>
              ……
              </trainDetail>
              <locations>
                <from>Belfast</from>
                <to>Dublin</to>
              </locations>
              <date>2005/07/28, 10:00 a.m.<date>
                ……
          </requestInfo>
        </content>
      </message>
    </event>
    <processing>
      trainJourney = new TrainJourney (requestInfo)
    </processing>
    <condition>
      trainJourney.ValidateTrainPlan () = = true
    </condition>
  - <action>
      <type>send a message</type>
```

```
- <message>
    <from>TrainRunning.AcceptLateAddition</from>
    <to>TrainOperator.ReceiveTrainJourney</to>
    <title>confirm the "Late journey request" is accepted</title>
  - <content>
    - <responseTo>Late journey request</responseTo>
    - <result>accepted</result>
    - <trainJourney>
      - <journeyId>
          200510010100
        </journeyId>
      - <journeyDetail>
          <from>Belfast</from>
          <to>Dublin</to>
          <date>2005/07/28, 10:00 a.m.<date>
          ……
        <journeyDetail>
      </trainJourney>
    </content>
  </message>
  (Also send this message to other interested parties)
</action>
<priority>5</priority>
</local-rule>
```

Figure 6. XML definition for the rule "AcceptLateAddition"

Agent Communication Diagrams are good at showing collaboration among agents, while rules are good at precise definition of agent behaviours, which is not possible in UML Sequence or other Diagrams [1].

The definition of rule "AcceptLateAddition" in XML (Figure 6) is based on the rule specification given in section 3.4 and is executable by computers. In the case of rule "Accept Late Addition", suppose all previous rules managed by the agent "TrainRunning" are not applicable and it's time to check the applicability of this rule. The agent parses the message just received and finds out that the message has come from an agent with the name "TrainOperator". The agent knows its rule "Accept Late Addition" is defined to deal with the message received from that agent, because according to the rule definition, the content of XML element <event>/<message>/<from> matches with that agent in the name. Also the message content has the same structure as specified in the rule (in rule definition shown in Figure 6, the content between tags in italic is possible message content, where the rule specifies it will only accept events with this kind of structure). Then, by invoking business classes, managed by the agent, a business object of "trainJourney" can be constructed (the name of which is declared and shared as a global variable), and its validity can be evaluated. The "trainJourney" object can be built from the content of the <event>/<message>/<content> structure of the received message. Its validity can be checked using the "ValidateTrainPlan ()" method of the business class. If the "trainJourney" is valid then the condition for executing the rule is satisfied. A corresponding message will then be structured using the created "trainJourney" object (the name of which refers to the global variable) and sent to all interested agents, including "TrainOperator", again as it is specified in <action>/<message>/<to>. Finally the "TrainRunning" agent adds the knowledge that the "TrainOperator" agent has sent a late train journey request at this time to its own beliefs. When enough such information is collected, business reports can be built for analysis for the later use. In the whole rule execution process, if the event does not match or the condition is not satisfied, the next candidate rule with the highest priority will be tested for applicability and executed in the same way.

After the behavioural models are built, the requirements are documented in a set of Business Process Rules, each delegated for a high level business goal. The details of business rules that constitute the business processes are further defined in XML, representing system functionalities that realise the goals. With the Agent Communication Diagrams and XML-based rules documented, any later change to the traditional specification will be mapped to XML element content, according to the mapping from the four main sections of functional requirements to the four main XML elements, and the corresponding UML model structures. The advantage of this is that the maintenance of the requirements on UML diagrams and XML definitions is much easier than the text-based descriptions and the UML/XML-based format is less likely to bring ambiguousness, incompleteness and inconsistency. These changes will be further transformed to the implementation so that maintenance of code is avoided. The details of this are in the next section.

To illustrate one of the benefits the approach brings, suppose there is a change that requires the function production being available to a different entity, as described in the <Required Effect> section of the traditional specification. This means that an agent needs to send its actionMessage to a different agent, hence the matching XML tag <action>/<message>/<to> of the function's corresponding rule will change. This also requires us to check the validity of the association between this agent and its previously associated agent. This check compares the XML tag <event>/<message>/<from> of its associated rule and the <Cause> section of that rule which matches the original functional requirements. Moreover, the change can be visualised in our UML diagrams, the direction of a message switching from the changed rule to a different agent. This example demonstrates how the consistency of the requirements can be checked at any time.

3.6. Step four: implementation

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

Figure 7. Pseudo code for the case study, transformed from the requirements models

Agent-oriented systems can be generated from the models obtained in step three. It should be noted that, although agent-oriented implementation is a straightforward and also the recommended one, it is not restricted to that, due to the fact that the requirements models built are neutral requirements documentation. In fact they have no assumptions on the technology used to implement them.

Nevertheless, implementing each conceptual agent in our models as a software agent is natural. One running unit should be used to represent the domain and several such units collaborate in an information passing fashion to achieve a common goal, with each contributing its own knowledge or knowledge processing capability. A tool has been developed which generates agent systems running on the JADE platform [18]. Each generated agent represents a corresponding agent in the Agent Diagram.

According to the steps that agents follow for the rule execution processes, there will be an agent behaviour generated for each rule of the agent. Thus a functional requirement maps to an agent behaviour in this approach. These behaviours can function simultaneously so that agents are multi-threaded. A shared module "Rule" is used by all behaviours with the ability to access the XML definition of rules and assemble corresponding objects. Methods of getPriority(), getEvent(), and getAction() are provided in the "Rule". On receiving an incoming event, an agent reacts by performing one of the defined behaviours. The one with the highest priority is retrieved to check its applicability. It will be executed if it passes the check; otherwise the same procedure will be carried out to check the next one, and so on, until one of the available behaviours is performed. For each single behaviour, first of all, the tool generates a "setPriority()" statement, then retrieves all global variables used in the rule and declares these.

After that the tool generates an "if" statement to evaluate if the received message matches with the expected incoming message, as specified in the rule. If this is the case, the received message is processed and some business objects are assembled at this stage (they are usually the declared global variables). A check is then made to test whether the rule condition is satisfied. Finally the tool generates an instantiation statement for a new message, a set content statement to encode some business objects declared as the global variables to the message, a send message statement to send it, and an add belief statement to update the agent's beliefs .

Figure 7 shows the pseudo code for an agent behaviour generated by the tool (Figure 8). Only minor human effort has been put to adjust the code. This sample is for the rule "Accept Late Addition", agent "TrainRunning".

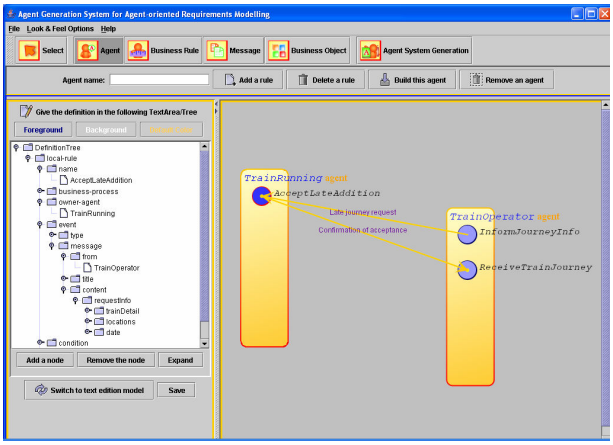

Figure 8. Screenshot from supporting tool

The overall result is, one piece of the functional requirements in Table 1 is modelled as an event-based rule "AcceptLateAddition", and then an agent behaviour as shown in Figure 7. Another piece is modelled as an internal business method "ValidateTrainPlan ()", for the "TrainJourney" business class. The later implementation element is invoked to assist the former implementation element in the above agent behaviour code, reflected in the original requirements as the later is a "Sub-Reqt of" the former. Thus functional requirements have been modelled properly in two different levels: rules and methods, reflecting their relationship in the original requirements.

4. Evaluation and Conclusion

Extra notations are defined in our modelling approach to complement UML in capturing structural and behavioural semantics as well as internal and external collaboration. As opposed to traditional requirements models, the agent-oriented modelling approaches as proposed have an advantage over the plain UML models in that they are not a development overhead. Executable systems are actually automatically generated from our models. This is due to the fact that accompanying XML rule definitions are used to explain function details for each UML element and to describe their behaviours. These requirements can not be well documented in graphical UML diagrams. With this approach, there will be no excuse for skimping on requirements/design and accurate translation from the diagrams to the running systems is possible. Maintenance of the models is actually maintenance of the running systems.

As the requirements model itself is also intended as a design model, the requirements and design phases are in effect merged. Because of this, our approach should reduce the cost of maintenance. When requirements change from the baseline system specification, the only manual changes needed are in the UML diagrams and the XML. These are much less painful to modify than the code. They can be updated using the same steps given for the requirements model transformation wherever there are changes. The implementation would be automatically re-generated by tools. Hence, we do not need to look for places in the code to reflect every change in the requirements. This helps us to avoid the risk of touching code frequently during the maintenance and guides us to an effective way to change the system. Business classes used by rules, however, may need to be maintained manually. Since they are independent simple components, each representing one aspect of the business domain, they are relatively stable. The interaction and logical combination of these is at the agent level and can be re-configured frequently and easily, supported by methods and tools.

Our investigations using real requirements specifications show promise in providing a method for constructing requirements specifications using agents and ultimately generating running agent systems which precisely conform to their requirements models. At the moment, they have limited autonomy and intelligence after the transformation, truthfully reflecting the requirements demand. However, the integral design of agent beliefs leaves the possibility of adding advanced autonomous features to agents. For example, agents may resolve requirements conflicts or may reject some requirements from a judgement based on their experience. This add-on will be developed as a future enhancement to strengthen the approach.

5. References

[1] Fowler, M., "UML Distilled: A Brief Guide to the Standard Object Modeling Language, 3rd Edition", Addison-Wesley, 2004.

[2] Kleppe, A., Warmer, J. & Bast, W., "MDA Explained: The Model Driven Architecture: Practice and Promise", Addison-Wesley, 2003.

[3] Beck, K., "Extreme Programming Explained: Embrace Change", Addison-Wesley, Boston, 2000.

[4] Morgan, T., "Business Rules and Information Systems", Addison-Wesley, 2002.

[5] Object Management Group, Inc., 250 First Ave. Suite 100, Needham, MA 02494, USA.

[6] Ambler, W. S. & Constantine, L. L., "The Unified Process Inception Phase: Best Practices in Implementing the UP", R&D, 2000.

[7] Warmer, J. & Kleppe, A., "The Object Constraint Language: Getting Your Models Ready for MDA, Second Edition", Addison-Wesley, 2003.

[8] Rosca, D. & Wild, C., Towards a flexible deployment of business rules, Expert Systems with Applications, Vol 23, Issue 4, 385-394, 2002.

[9] Liu, W., Easterbrook, M. S. & Mylopoulos, J., Rule-based detection of inconsistency in UML models, Proc. Workshop on Consistency Problems in UML-Based Software Development, 5th Int. Conference on the Unified Modeling Language, Dresden, Germany, 2002.

[10] Jennings, N. R., Sycara, K. & Wooldridge, M., A Roadmap of Agent Research and Development, Autonomous Agents and Multi-Agent Systems, 1 (1998) 7-38.

[11] Yu, E., Why Agent-Oriented Requirements Engineering, Proc. 3rd Int. Workshop on Requirements Engineering: Foundations for Software Quality, Barcelona, 1997.

[12] Feather, M. S., Composite System Design, ICSE-16 Workshop on Research Issues in the Intersection Between Software Engineering and Artificial Intelligence, Int. Conference on Software Engineering, Sorrento, Italy, 1994.

[13] Dardenne, A., Lamsweerde, A. & Fickas, S., Goal-directed Requirements Acquisition, Science of Computer Programming, Vol 20, Issue 1-2, 3-50, 1993.

[14] Heymans, P. & Dubois, E., Scenario-Based Techniques for Supporting the Elaboration and the Validation of Formal Requirements, Requirements Engineering, Springer, 3, 202-218, 1998.

[15] Lamsweerde, A., Requirements engineering in the year 00: A research perspective, Proc. 22nd Int. Conference on Software Engineering, 5-19, ACM Press, 2000.

[16] Lamsweerde, A., Goal-Oriented Requirements Engineering: A Guided Tour, Proc. 5th IEEE Int. Symposium on Requirements Engineering, Toronto, IEEE, 249-263, 2001.

[17] Kavakli, V. & Loucopoulos, P., Goal-Driven Business Process Analysis - Application in Electricity Deregulation, Information Systems, Vol 24, No 3, 187-207, 1999.

[18] Jade platform, http://sharon.cselt.it/projects/jade/