Prioritisation mechanisms to support incremental development of agent systems

Padgham, Lin; Perepletchikov, Mikhail

Padgham, L., & Perepletchikov, M. (2007). Prioritisation mechanisms to support incremental development of agent systems. International Journal of Agent-Oriented Software Engineering, 1(3/4), 477–497. https://doi.org/10.1504/IJAOSE.2007.016269
Document Version: Accepted Manuscript

Published Version: https://doi.org/10.1504/IJAOSE.2007.016269

Prioritisation mechanisms to support incremental development of agent systems

Citation:
Padgham, L and Perepletchikov, M 2007, 'Prioritisation mechanisms to support incremental development of agent systems', International Journal of Agent-Oriented Software Engineering, vol. 1, no. 3/4, pp. 477-497.

Lin Padgham* **and Mikhail Perepletchikov**

School of Computer Science and Information Technology,
RMIT University,
Melbourne, Australia
Fax: +61 3 9662 1617
E-mail: linpa@cs.rmit.edu.au, mikhailp@cs.rmit.edu.au
*Corresponding author

Abstract: It is often necessary to partition a project into different priority levels and to develop incrementally. This paper presents a mechanism whereby a developer can prioritise scenarios on a five point scale, leading to automated, coherent partitioning of all required design entities, according to the three IEEE defined priority levels of *essential*, *conditional* and *optional*, which are used in many companies. This allows for automated support to guide the developer as to what design artefacts need to be developed at each phase. The developer can indicate the relative sizes desired for the three partitions and the algorithm described will attempt to get as close to this as possible. It is also possible to move items manually to achieve better sized partitions, as long as priority orderings are not violated. The approach is fast and easy to apply at various times during development, as needed.

Keywords: Agent Development methodologies; Prometheus; Iterative incremental development; System prioritisation.

Reference to this paper should be made as follows: L. Padgham and M. Perepletchikov, 'Prioritisation mechanisms to support incremental development of agent systems, *Int. J. Agent-Oriented Software Engineering*, Vol. x, No. x, pp.xxx–xxx.

1 Introduction

Agent oriented software systems, like any software system, are best developed incrementally, with multiple releases, where each release incorporates additional functionality, as suggested by the Rational Unified Process model (RUP) (Kroll and Kruchten, 2003; Kruchten, 2004), well known within Object Oriented Software Engineering. This allows developers to concretely demonstrate milestones to clients. It also ensures that if resources are expended before the desired system is complete, there is at least a functioning system, though perhaps without all the functionality originally hoped for. This is (usually) far preferable to a partially completed larger

system.

In developing a system incrementally there are at least two important activities. First the required functionalities must be prioritised, often into three levels (essential, conditional, optional), used by many organisations (Firesmith, 2004; Wiegers, 1999), and compatible with the IEEE standards (IEEE-Std830, 1998). Secondly the developer must identify the parts of the system that are necessary in order to realise each level of functionality. It is also desirable to be able to estimate the amount of work in each level, in order to be able to manage the project appropriately.

The Rational Unified Process (RUP) is perhaps the most widespread and well known iterative and incremental software development process. It suggests a spiralling iteration over four phases including activities of business modelling, requirements gathering, analysis and design, coding, testing, and deployment. The RUP model also suggests that there should be iterative, incremental releases of the software being developed, with each release incorporating additional functionality (Kruchten, 2004).

To the authors' knowledge there is no work yet done in the Agent Oriented Software Engineering (AOSE) community that specifically supports this kind of iterative development in any structured way. The Prometheus methodology for developing agent systems (Padgham and Winikoff, 2004) suggests that an iterative approach should be applied, but has previously had no specific mechanisms to support such. The work presented in this paper provides some structured support for iterative development by taking prioritisations given at a high level of abstraction, and then using these to partition the system into three coherent levels for iterative development. The Prometheus Design Tool (PDT) can then support the developer in focussing only on things within the current priority level.

In the absence of any mechanisms for principled estimation of size of a development task in AOSE, the approach here is to use a coarse level of granularity for partitioning the system, which has the advantage of being quick to redo if new requirements are added, or if requirements need to be re-prioritised. Future work in how to estimate development effort for agent systems, may well motivate a more detailed and fine grained approach to prioritisation and specification of development levels. However even if such mechanisms exist, it is likely that there will be situations where a fast, coarse grained approach is preferable to a time consuming more accurate approach to determining the content of the incremental iterations.

Our approach requires an initial prioritisation of scenarios, which we then partition into three levels, allowing for developer input into the relative sizes of these levels. The algorithm we have developed then propagates this prioritisation to goals in a principled manner, and further propagates priorities to all design entities. This is made possible by the structured nature of the Prometheus design artefacts, which supports the automation of this process.

In the following sections we first briefly introduce the Prometheus methodology, and the design artefacts produced. Next, we describe the prioritisation process and algorithms, an initial version of which can be found in (Perepletchikov and Padgham, 2005a). This leads to a system specification consisting of the three prioritised levels (essential, conditional, optional), previously mentioned. We also describe how the prioritisation process is applied using incremental development through the phases of architectural and detailed design, as well as implementation and testing. This is then illustrated using the electronic bookstore case study from

"Developing Intelligent Agent Systems: A practical guide" (Padgham and Winikoff, 2004). Finally we briefly review related work in prioritisation of requirements outside of Agent-Oriented SE[a]. The prioritisation mechanisms described have been incorporated into a version of the Prometheus Design Tool (PDT)[b] and is in the process of being integrated into the publicly available version of PDT.

2 Overview of the Prometheus methodology

Prometheus consists of three main design phases, plus implementation and testing which are not currently covered in much detail[c]. The design phases are:

1. System Specification,
2. Architectural Design,
3. Detailed Design.

The prioritisation process that we introduce is based on a full system specification. However, it is possible to specify less completely those aspects of the system which are a lower priority. In this case one may want to compensate for this by having a larger system determined essential partition, recognising that those partitions which are not yet fully developed are likely to increase in size.

We describe each phase briefly, noting the artefacts that are produced.

2.1 System Specification

System Specification[d] consists of the following steps:

- Identification of *actors* and their interactions with the system;
- Developing *scenarios* illustrating the system's operation;
- Identification of the *system goals* and sub-goals;
- Specifying the interface between the system and its environment in terms of *actions*, *percepts* and any *external data*;
- Grouping goals and other items into the basic *roles* of the system.

Firstly we identify *actors* as being any persons or entities which will interact with the system, as well as any other stakeholders whose goals should be considered. Actors may be other software systems, as well as humans. The concept of actor used here is similar to that found both in UML use-case diagrams and in the Tropos agent oriented methodology (Bresciani et al., 2004), although Prometheus actors may be software as well as people. The principal difference between *actors* and *agents* is whether they are external or internal with respect to the system boundary. Actors are external to the agent system under development, whereas agents are software entities within this system.

[a] To the authors' knowledge there has not been any work done in this area specifically within AOSE.

[b] PDT is available at www.cs.rmit.edu.au/agents/pdt

[c] There is substantial work (e.g. (Padgham et al., 2005)) on using Prometheus design documents in debugging, but this is not yet fully integrated.)

[d] The system specification process has been refined and modified since (Padgham and Winikoff, 2004), and it is the most current version which is presented here. A study evaluating the benefits of the refinement can be found at (Perepletchikov and Padgham, 2005b).

Initial scenarios are then identified associated with the actors that will interact with the system, similarly to use case identification in object oriented analysis. The input from actor to agent system is then identified as a *percept*, while the outputs from system to actors are defined as *actions*.

Each scenario is associated with a goal (which the scenario is one way of achieving). Multiple scenarios can be associated with a single goal. The scenarios are then developed as a number of detailed steps, where each step is a *(sub)goal, (sub)scenario, action* or *percept*. Figure 1 provides an example of a scenario[e].

Scenario: Query Late Delivery Scenario1

Goal: Manage Late Delivery Query **Trigger:** Late Delivery Query

#	Type	Name	
1	Goal	Determine delivery status	(check records)
2	Goal	Log delivery problem	(enter in log)
3	Action	Request delivery tracking	(contact courier or P.O.)
4	Goal	Inform customer	(email that its being followed up)
5	Goal	Monitor Tracking Request	(follow up if no response)
6	Percept	Tracking info	(in this case 'lost')
7	Goal	Arrange delivery	(replacement item)
8	Goal	Log books outgoing	(keep internal stock logs updated)
9	Goal	Inform customer	(email that replacement sent)
10	Goal	Update delivery problem	update internal logs

Figure 1 Example Scenario: Query Late Delivery

The linking of scenarios and goals is inspired by the Goal-Scenario coupling framework (GSCF) (Rolland et al., 1998) which is based around the notion of a Requirement Chunk (RC) (a pair of <Goal, Scenario>). However, we allow a single goal to be linked to multiple scenarios, whereas GSCF is one-to-one.

Initial goals are identified partially from the initial scenarios, as well as by consulting project stakeholders, and by examining the initial system description. Further goals are then identified by a process of abstraction and refinement (van Lamsweerde, 2001). For each goal, asking the question *how?* and *why?*, potentially identifies new subgoals and parent goals, forming a goal hierarchy. This process continues until all the stakeholders' (external) goals have been elicited and then sufficiently refined in order to reach operational level (internal) goals of suitable granularity for assigning to a role. Elicitation processes for goals and scenarios are interlinked and highly iterative.

We allow the (optional) notion of two kinds of goal refinement: *AND-refinement* and *OR-refinement* as described by van Lamsweerde (van Lamsweerde, 2001). If a goal is AND-refined, subgoals (or answers to the question *how?*) are steps in achieving the overall goal, and each step must be done. If it is OR-refined, then subgoals are alternative ways of achieving the goal, and doing any one of them is sufficient. Agent systems typically have both these kinds of refinements. OR-refinements allow for choice in the way of achieving goals, while AND-refinements allow for breaking down into smaller pieces. For example, figure 2 shows a partial goal hierarchy related to the 'Query a Late Delivery' scenario. If neither refinement is specified, AND-refinement is assumed for the purpose of prioritisation.

[e]Additional information is also attached to the scenario which is not shown here, and which is not required for this discussion.

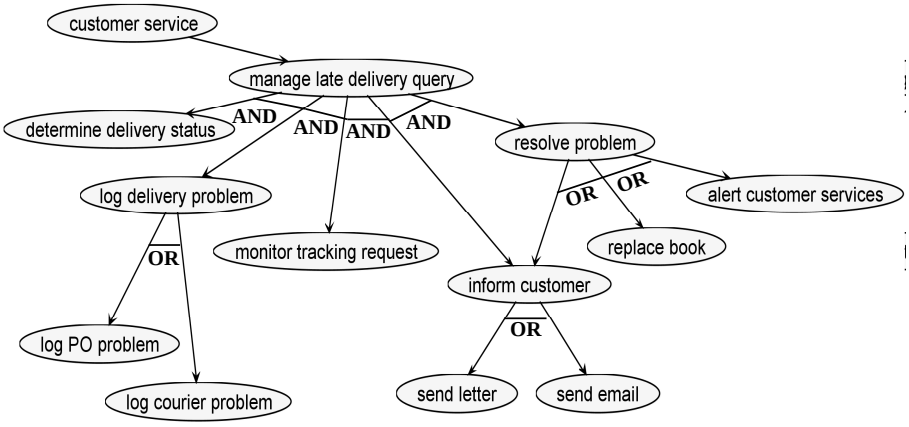

Figure 2 Partial goal hierarchy for the Electronic Bookstore

After goals and scenarios are sufficiently developed[f], goals are grouped based on their functional relatedness and then associated with roles. Actions and percepts are also allocated to roles and scenarios are then annotated with information about which role each step belongs to.

2.2 Architectural Design

The architectural design phase refines the system specification to determine the agent types[g] within the system, and specifies the interactions between these agent types. The main steps in this phase are as follows:

- Deciding what *agent types* will be implemented and developing the *agent descriptors*
- Describing the dynamic behaviour of the system using *interaction diagrams* and *interaction protocols*.
- Capturing the system's overall (static) structure using the *system overview diagram*.

The major decision to be made during the architectural design is which agent types should exist. Waiting until architectural design to decide types, by grouping the more limited and abstract roles, allows for consideration of a range of design issues although space restrictions prevents us from describing these here. They include data coupling and cohesion, as well as issues specific to particular types of application, such as the coupling arising due to participation in decision making in control systems (Bussmann et al., 2004).

[f]The point at which goal and scenario development is sufficient is subjective, and based on experience.

[g]We refer to *agent types* rather than simply agents, as it is possible that there will be multiple instantiations of any given agent type. For example a customer assistance agent may be instantiated each time a new customer logs into the system.

Once the agent types have been decided it is possible to start to define the interactions between them. The scenarios developed earlier assist in this process. The first step is to convert each scenario to an agent interaction diagram, which is similar to a UML sequence diagrams but with agents rather than objects. This is not something which can be automated, but there are heuristics which can be used to assist in the process. Due to lack of space we do not explain this here, but refer the interested reader to (Padgham and Winikoff, 2004).

The developed interaction diagrams are then generalised to interaction protocols which fully define the interactions between agents. This is done by merging related interaction diagrams, and by considering at each point in the interaction diagram, what else could occur at that point. AUML-2 (Huget and Odell, 2004) is the current notation used to specify interaction protocols as it appears to be an emerging standard. However any similar notation would be suitable.

The system overview diagram is perhaps the single most important product of the design process. It ties together agents, data, external input and output, and shows the communication between agents. It is obtained by linking interface entities (percepts, actions and external data) to specific agent types, and by showing the interaction protocols connecting agent types. Shared internal data can also be shown.

Figure 3 shows an example system overview diagram. It should be noted that all information for the system overview diagram exists within the design specified so far, and can be automatically assembled via a support tool. The system overview diagram simply brings information together in an easy to visualise summary.

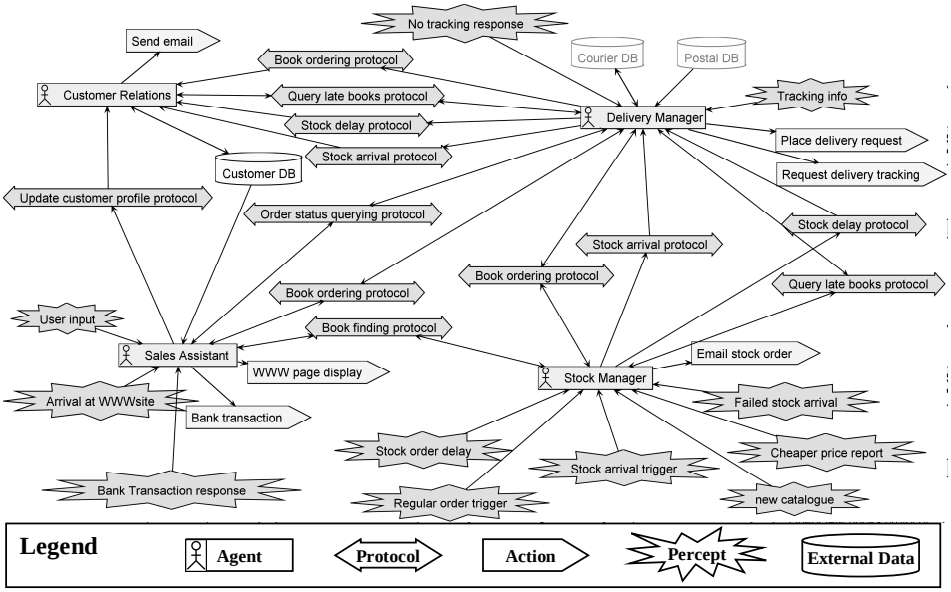

Figure 3 System overview diagram for electronic bookstore

2.3 Detailed Design

The focus of detailed design is on developing the internal structure of each agent and how it will achieve its functioning within the system. The details of agent functioning are specified using *plans*, which are essentially recipes for agent acting. Plans may be abstract, referring to subgoals, or subtasks. The process allows for progressive refinement, first defining *capabilities* (modules within the agent), and then *plans* along with internal *messages* or *events*, and detailed *data structures*. We also use the protocols to define process diagrams showing the internal processing within each agent. Process diagrams in turn guide the development of plans. The various aspects of the detailed design process are as follows:

- Identifying and developing *capabilities* and their inter-relationships, resulting in an *agent overview diagram*.
- Development of process diagrams showing the *internal processing* of each agent related to the protocol specifications.
- Development of *plans, events* and *data* and their inter-relationships.

The agent overview diagram is similar in structure to the system overview diagram, but focuses on agent internals. The content of the internals of each agent is defined by the functionalities developed during system specification. Internals can be initially conceptualised as capabilities. These are eventually described in terms of plans and triggers for those plans. Triggers can be percepts from the environment, messages from another agent, or internally instantiated goals, subgoals or events.

Process diagrams are similar to UML activity diagrams, with some modifications. They describe the processing of a single agent, with respect to a single protocol within the system. Consequently, they can be related back to scenarios.

3 Prioritisation Process

The prioritisation process that we propose uses the scenarios as the primary artifact for prioritisation, and then prioritises goals, and other artefacts based on this. The goal(s) associated with a scenario is clearly an important factor in the prioritisation of the scenario. However in some cases a particular goal may have multiple alternative scenarios, some of them of high priority, and some of them capturing more advanced, but lower priority behaviour. This makes it more appropriate to prioritise the scenarios than simply the goals. Also the nature of scenarios ensures that a particular prioritised partition has a completeness in terms of expected system processes. In addition there are usually fewer scenarios than goals, and they are more detailed, enabling a better understanding of what is being prioritised.

3.1 Prioritising System Specification Artefacts

After discussing with various stakeholders (such as clients, users or managers), the developer is required to assign a *priority ranking of one to five* to each scenario that has been identified. Where a scenario is included as a step in another scenario

it must be assigned at least the ranking of the parent scenario. If it is used in multiple scenarios it must receive at least the ranking of the highest ranked parent. It may also be ranked higher than the parent(s), but cannot be ranked lower as it is needed to support achievement of its parent scenario.

We will eventually arrive at *three priority-based partitions*, according to common practice. However we start with five initial rankings, in order to provide a broader range of options and to assist in avoiding the trap of insufficiently prioritised requirements, where, for example "more than 90% of the requirements are classified as high priority" (Wiegers, 2000)[h].

Once all scenarios have been assigned a priority, we apply a prioritisation algorithm which attempts to partition the scenarios into three suitably sized partitions based on the rankings obtained. The relative sizes of the *essential*, *conditional* and *optional* partitions can be specified as ranges by the developer, depending on the particular application constraints. For illustrative purposes we use an *essential* partition of 35-45% of use-case scenarios, a *conditional* partition of 20-40%, and an *optional* partition of 25-35%.

The prioritisation algorithm will attempt to stay as close to these partition sizes as is possible, given the rankings supplied, maintaining the constraint that ranking 1 should map to the *essential* partition, and ranking 5 should map to the *optional* partition. Consequently, if too many level one (or level five) priorities are given, it may not be possible to stay within the recommended partition sizes.

The rankings from one to five give us six different possibilities (A-F) for how to assign rankings to partitions as is shown in figure 4. We calculate the percentage of scenarios in each partition, for the different options and then score the various options by giving a point for each partition that is within the desired range as shown in figure 5.

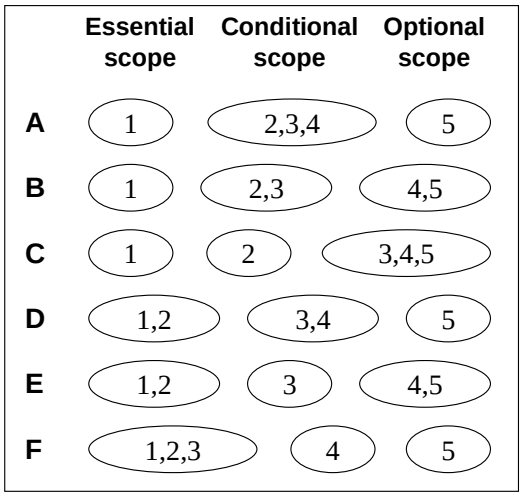

Figure 4 Possible groupings into prioritised partitions

[h]While it is still the case that too many scenarios can be prioritised at level one, giving a finer granularity is likely to encourage use of the full scale, most likely resulting in fewer level one categorisations.

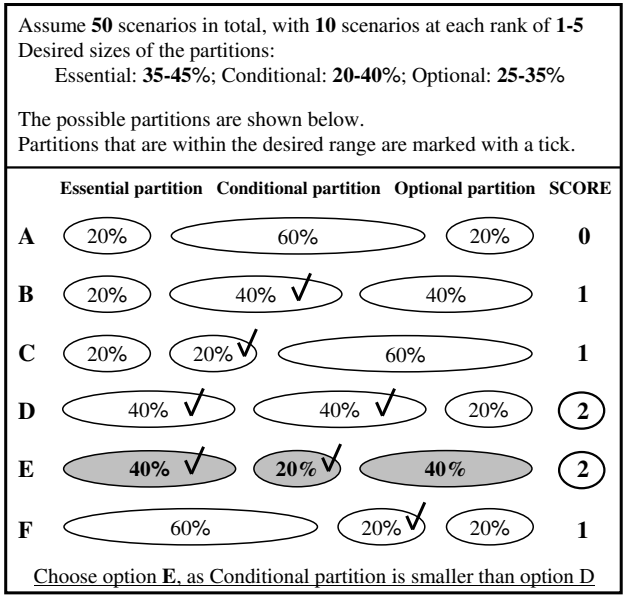

Figure 5 The process of selecting the 'best' partitioning option

If more than one option shares the highest score, we by default choose the one with the smallest essential partition. However, it is also possible to present the options visually to the developers (or other stakeholders) and allow them to choose. If no options have all partitions within desired ranges, we prefer partitions with two partitions of appropriate size, and within this set, we prefer those with the essential partition of the approved size. Other things being equal, we also prefer, by default, to have fewer scenarios in higher ranked partitions. Again, it is possible to present all groupings to the developers and allow them to choose the preferred one. Alternatively, one could introduce additional information such as ranking of actors (or stakeholders), with prioritisation of scenarios involving actors' rankings.

Although it has not been done here, scenarios within a given rank (other than 1 or 5) could be split and placed in different partitions, while maintaining the relative partial ordering given by the original ranking. For example some scenarios initially ranked three may be placed in the *essential* partition and some in the *conditional* partition. However none should be placed in the *essential* partition if there are scenarios ranked two, placed in the *conditional partition* (as these are above the level three scenarios). It would also be possible to provide support whereby the developer could move particular scenarios between partitions in order to achieve better distributions, again maintaining the originally specified relative rankings.

Once scenarios are assigned to partitions, other artefacts are assigned based on their connections to scenarios. Actions and percepts are assigned to the same partition as the scenario(s) in which they occur. If they occur in multiple scenarios, from different partitions, then they are assigned to the highest priority partition.

Goals are somewhat more complex, as not all goals will be mentioned in scenarios. Also goals are structured with respect to each other, and the prioritisation

must be consistent with this structure. The initial step is that for each goal that is linked to a scenario, the goal is assigned the same priority as the scenario. The remaining goals are then assigned priorities according to the rules described in the following sub-section.

Roles are assigned to a partition according to the highest priority goal that is included in the role. However roles will be developed incrementally, taking first only those aspects which are related to high priority goals. We prioritise the roles themselves as, if it is possible to exclude an entire role from a partition, this simplifies prioritisation of other artefacts such as protocols.

Prioritising Goals

Many goals can be prioritised directly according to associated scenarios. If the goal is linked to a scenario, the priority of this goal is equal to the priority of the linked scenario. Where a goal is a step in a scenario it is assigned to the same priority partition as the scenario it is a step in. Where a goal receives different priorities from different scenarios, the highest priority level is used.

The assignment of remaining goals to some partition involves considering both parents and children of the goal to be assigned. The Prometheus guidelines require that all goals should be *covered* by some scenario, where being covered involves either a parent goal being included in or linked to a scenario, or subgoals being included in a scenario in a way that adequately covers the parent goal. With the introduction of AND vs OR refinement, adequate coverage by subgoals implies either one OR-refined goal being included in or linked to a scenario, or all AND-refined goals being included in or linked to some scenario. If these guidelines regarding coverage are not followed it will be necessary for the developer to explicitly assign priorities to non-covered goals. In describing the prioritisation rules we assume that all goals are covered by scenarios. The system checks this and generates a warning if it is not the case.

There are three rules which must be maintained for a set of allocations of goals to the partitions to be acceptable:

- **All** goals must be in a partition at least as high as that of the scenario they are included in or linked to.
- **All** AND-refined subgoals must be assigned to a partition at least as high as their parent.
- **Some** OR-refined subgoal must be assigned to a partition at least as high as the parent.

The first constraint is fulfilled by the initial allocation described above. We then ensure the second constraint is satisfied by allocating all unallocated goals with AND-refined subgoals, and all AND-refined subgoals, as illustrated in Figure 6. If an AND-refined subgoal is unallocated it receives the priority of it's parent. If an AND-refined parent is unallocated it receives the priority of its lowest prioritised child. The Prometheus rule that all goals are covered by a scenario disallows both a parent and its AND-refined child to be unallocated.

We also reallocate any AND-refined subgoals whose initial allocation from scenario linking was too low to meet the constraint. This leaves us with potentially unallocated OR-refined subgoals along with any unallocated top level goals that are OR-refined.

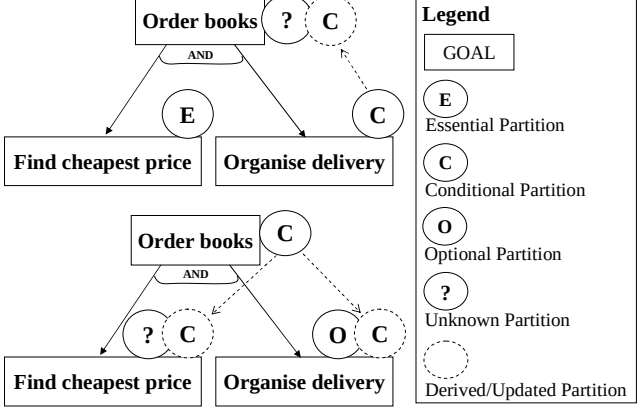

Figure 6 Allocations involving AND-refinement

When an unallocated top level goal is OR-refined we assign it to the same partition as its highest priority subgoal, as this is the partition in which it will be achieved. However, we note that this is an *inherited priority*. In the event that a role is receiving its priority level based on this goal, further assessment may need to be done. Each unallocated OR-refined subgoal is assigned to a partition depending on whether the current prioritisation fulfils the third constraint above. Consequently, if a sibling subgoal is already assigned to a partition at least as high as the parent, then the goal under consideration is assigned to the lowest partition. Otherwise it is assigned to the same partition as the parent goal. Figure 7 illustrates these allocations.

Prioritising Roles

The priority of a role is determined by the highest priority goal assigned to this role. For example, if the role X includes three goals that belong to the *essential*, *conditional*, and *optional* partitions respectively, the priority of role X will be *essential*. However, the role will only be developed to the extent required within each partition. The one exception to this is the case where the priority of the role is being determined by a goal which has an inherited priority. In this case further analysis needs to be done, to determine whether the inherited priority was justified.

We recall that the inherited priority occurs only when a top level goal is OR-refined, and is initially unallocated, in which case it inherits the priority of the highest child. This is justified if the child has no other parent (in which case it must have received its priority directly from a scenario). However, if the child has some other parent which reflects its priority, then the prioritisation of the goal in question should be reassessed. It should receive the priority of the child (with the highest priority) for which it is the sole parent.

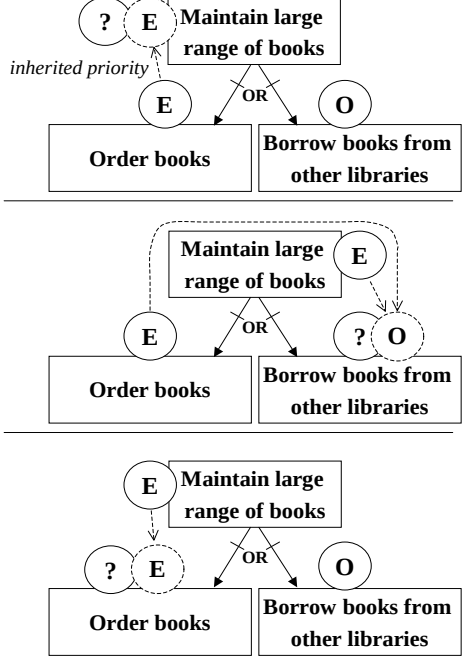

Figure 7 Allocations involving OR-refinement

3.2 Prioritising Architectural Design Artefacts

In moving to architectural design, it is necessary to decide the agent types within the system, based on the full set of roles[i]. Once this is done, agents can themselves be prioritised. The agent type needs to be placed within the priority partition of its highest prioritised role. There is a potential risk of having all/most agent types in the essential partition, but as the agents are also prioritised internally, only those aspects which relate to goals and scenarios within the essential partition will be developed as part of the initial phase.

At the architectural design phase, it is the development of protocols which is potentially most affected by the prioritisation. Interaction diagrams should be developed only for scenarios in the prioritised partition. When generalising the interaction diagrams to protocols (by merging interaction diagrams from alternative scenarios, and by asking at each point, "what other message may be sent here"?), it is necessary to consider options only to do with goals and scenarios that are in the desired partition. For example, consider an alternative scenario to that shown in figure 1 for the goal *Manage Late Delivery Query*, that is as shown in figure 8, and assume that only the goals *Determine Delivery Status* and *Inform Customer* are in the essential partition. This scenario is associated with a simpler version of the system which automates fewer aspects and relies on manual coverage of some functionality.

[i]We note however that if as requirements change in later iterations, possibly adding new goals and scenarios, a new role can be added, and this can then be assigned to a new or existing agent type.

Scenario: Query Late Delivery Scenario2

Goal: Manage Late Delivery Query **Trigger:** Late Delivery Query

#	Type	Name	
1	Goal	Determine delivery status	(check records)
2	Goal	Inform customer	(email that its being followed up)
3	Goal	Alert customer service	(email to follow up)

Figure 8 Alternative Scenario: Query Late Delivery

The interaction diagram and the generalisation to an AUML protocol for this is shown in figure 9. When the partition containing the scenario in figure 1 is developed, this protocol will need to be further developed. Messages are defined in line with the protocols, and as these are developed in more detail, new messages are added. Some protocols (and agents) may not exist at all in the essential partition. Others are likely to exist in a reduced form which then need to be modified as a new partition level is added.

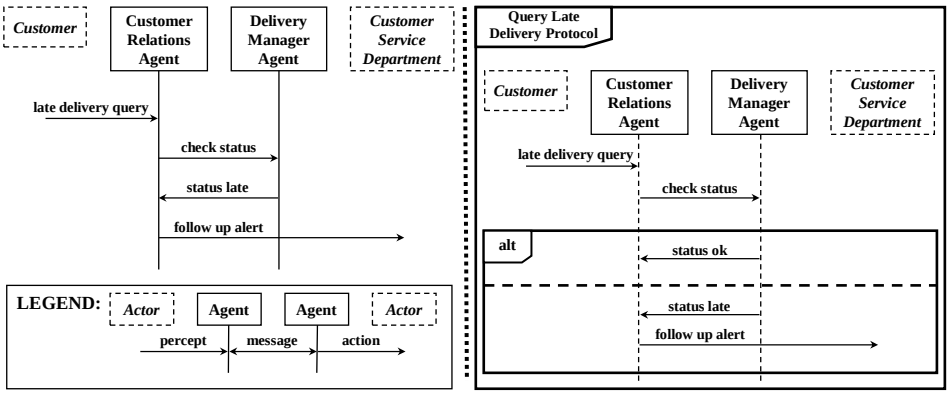

Figure 9 An example interaction diagram (on the left) and interaction protocol (on the right)

3.3 Prioritising Detailed Design Artefacts

In developing the detailed design, only the aspects of the agent that have to do with the goals from a current partition are addressed. Consequently, agent overview diagrams will be incrementally developed, with new capabilities and plans being added in successive increments to address the goals within the particular partition. Process diagrams follow directly from the reduced protocols, and thus are also developed according to the relevant prioritised partition.

As with the protocols and agents in architectural design, capabilities, process diagrams, and even plans may exist in a reduced form within a given partition, as well as being absent from higher priority partitions.

Dividing the system into three clearly separated priority levels results in the identification of three separate releases. In the first incremental iteration (release) we develop the system specification of an agent system. We then apply the prioritisation process to identify the *essential* partition which will go through the whole Software Development Life Cycle (SDLC) resulting in the first release. The second incremental iteration includes the development and integration of the *conditional* features, while the third release covers the *optional* features.

Including prioritisation mechanism within the Prometheus Design Tool (PDT) allows iterative development to be supported in a structured manner. In the process of developing the partitions, the developer (or any other project stakeholder) can interact with the tool to come to a desired partitioning, with the tool ensuring that necessary constraints are maintained. Once a partitioning has been determined, the developer can work within a particular partition, and all aspects of the system which are in a lower partition can be deactivated.

Prioritisation can be applied at any stage - and indeed if things are added to the specification, then priorities should be recalculated. The ability to prioritise system artefacts, thus allocating them to some priority partition, at any stage results in a flexible development model since the developer can apply the prioritisation procedure at any point of system design. For example, it is possible to fully design the system, and only then apply the prioritisation mechanism. This will then be propagated through the system, allowing implementation and testing to proceed in incremental iterations.

Also, there is often a "rapid descoping phase" late in a project, when it is necessary to determine features that can be cut down due to the lack of time and resources (Wiegers, 2000). Again, we can apply the proposed procedure to the final design artefacts in order to determine the less important system features.

4 Case Study

In order to illustrate how the prioritisation mechanism works in practice, we have used the case study of an Electronic Bookstore developed in "Developing Intelligent Agent Systems: A practical guide" (Padgham and Winikoff, 2004), and applied our procedures to it. We have augmented the original design with the actor diagram as described in the system specification phase. We have then provided rankings of 1-5 for all the scenarios, choosing those that are most basic/critical as 1, and those that are most advanced as 5, with others in between. While this was subjective and somewhat arbitrary, it seems a reasonable process for then exploring what support the proposed prioritisation mechanisms do actually give.

The initial description of the (desired) Electronic Bookstore system is as follows:

> *We would like to develop a fully online system for worldwide sale of books. This system will offer a broad range of books to customers, and a personalized, friendly user interface. The system must facilitate fast and reliable service at all stages, from locating a desired book, to delivery of the purchase. The store should have competitive prices.*

The resulting system as developed within (Padgham and Winikoff, 2004), can be described with the actor diagram as shown in figure 10.

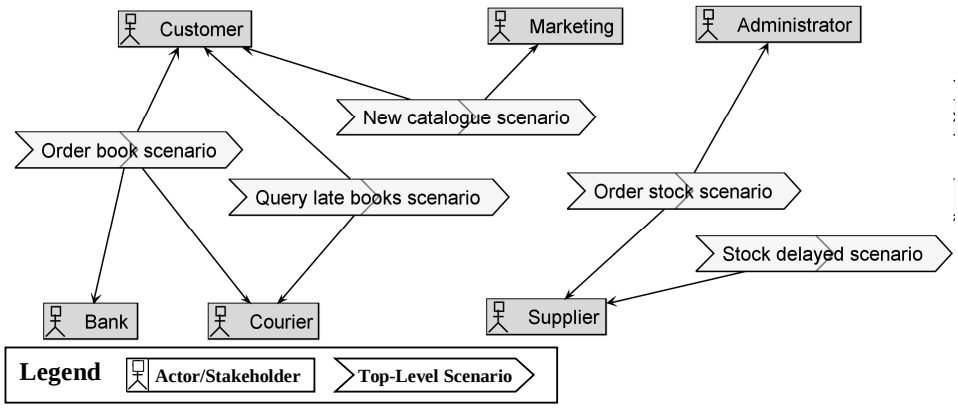

Figure 10 Overview of Actors and top level Scenarios for the Electronic Bookstore

The full list of identified scenarios from (Padgham and Winikoff, 2004) is as follows:

- Query Late Books scenario (5)
- Book Finding scenario (3)
- Order Book scenario (1)
- Pending Order Arrives scenario (2)
- Order Stock scenario (2)
- Stock Arrival scenario (2)
- Stock Delayed scenario (3)
- Missed Stock Arrival scenario (3)
- Order Status Query scenario (4)
- Customer Profile Update scenario (4)
- WWWsite Arrival scenario (3)
- Cheaper Price Notification scenario (4)
- New Catalogue scenario (4)

The numbers in parentheses after each scenario name are the rankings from 1 to 5 that we assigned to each scenario, with 1 being most critical and 5 least critical. The rationale for these rankings were that stocking books (Order Stock), and selling books (Order Book) are the core aspects of the system, and should therefore be priority 1. The scenario Cheaper Price Notification which actively monitors competitor prices to remain under them is something to be added later, as is the ability for customers to interact with the system regarding their orders (Query Late Books), so these received priority 5. Pending Order Arrives was ranked 2, as it is the scenario which allows for a book that was not in stock when ordered, to be sent out immediately it arrives. The rest were ranked 3 and 4 based on a quick decision about relative importance.

Running the prioritisation algorithm on the above initial rankings resulted in an Essential partition with three level 1 scenarios, a Conditional partition with five level 2 and 3 scenarios, and an Optional partition with five level 4 and 5 scenarios. Checking these against the desired percentages in each partition[j], only the conditional partition was actually in the desired range with respect to number of scenarios. The Essential partition with 15.4% was well under, while the optional with 38.5% was slightly over. However, scenarios are very coarse granularity. When we look at the partitioning with respect to goals, actions and percepts - the entities which are the building blocks of the application, we find that the Essential partition contains 46.5%, the Conditional has 18.6% and the Optional has 34.9%.

The most important aspect of the algorithm is that it does partition the application into *coherent chunks* for further development. With this approach all aspects required for a particular scenario will be developed within a given partition. In the absence of accurate measures of development time[k] a coarse level of granularity seems appropriate.

In this example, although the Optional partition is the only one within the desired bounds, the other two are close. However if the developer wishes to increase or decrease a particular partition size to be closer to the specified range for that partition, it is simply a matter of providing an interface for moving scenarios manually between partitions and rerunning the algorithm.

Figure 11 shows the roles (containing goals, actions and percepts) contained in each of the partitions. Each sub-figure shows the complete application at that level, with the new items for that partition circled. As can be seen, the Essential partition contains only 5 roles, of the total 12, while the Conditional contains 3 new roles, and the Optional 4 new roles[l].

When we come to the architectural overview, we find that all agents are represented, even in the Essential partition. However their functionality is substantially reduced. Of the seven protocols required in the full system, only two are required in the Essential partition, with another two in the Conditional, and the final three in the Optional partition. The Sales Assistant agent has two of its four roles in the essential partition, but all the other agents have only one role. As the main work of developing the agents is in their interactions (the protocols) and the internal capabilities and plans to implement the roles, we consider this quite successful.

As we move into Detailed Design, we note that the Sales Assistant will have two capabilities developed within the Essential partition, and two within the Conditional partition. In this case, each of the capabilities are fully developed within the relevant partition. However it can happen that some aspect of a capability is added when developing the later partition. It can then be implemented as a sub-capability, if desired. In the Stock Manager agent the aspects of the Stock management role having to do with handling delays, are within the Conditional

[j] essential: 35%-45%, conditional: 20%-40%, optional: 25%-35%

[k] This is an area for further research.

[l] The size of the images showing the Essential and Conditional partitions (top part of figure 11) has been decreased in order to fit all three partitions into one figure, for ease of comparison. The system structure (including the artefacts and relationships) shown in the Essential and Conditional partitions in the top part of the figure, is the same as the enlarged Optional partition, though without the extra elements. The reader can refer to the Optional partition for details of entities.

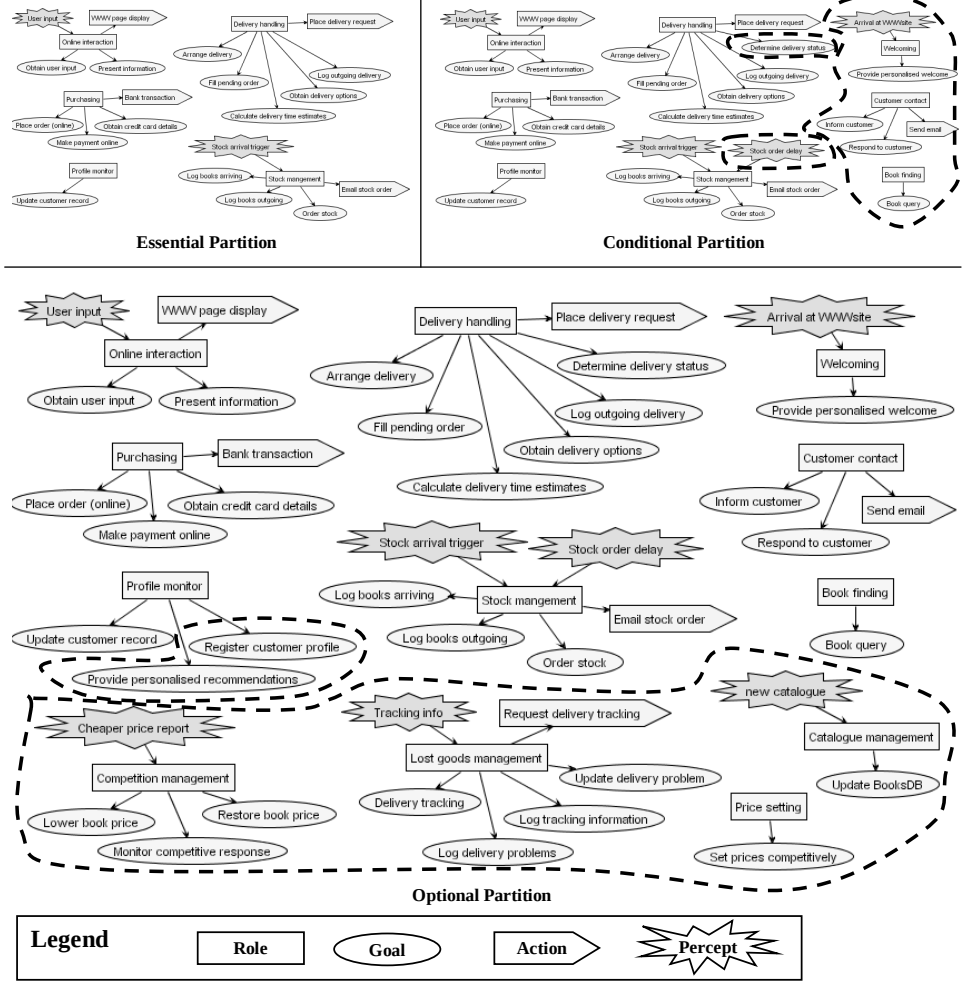

Figure 11 Prioritisation of roles, goals, actions and percepts: each partition includes the entities from the previous level, and the new entities are circled.

partition, although other aspects of this role are Essential.

The process diagrams within the detailed design are taken directly from the protocols, and thus these are also successfully limited within each of the agents, and developed incrementally within the different partitions.

4.2 Discussion

The case study indicates a very successful partitioning of the system, even though the initial partitioning of scenarios mostly did not result in the partition sizes specified. Using the scenarios as the basic prioritisation entity, ensures that the system pieces developed enable a coherent subset of the full functionality. The protocols were partitioned cleanly in the different levels, thus avoiding difficulties

of incrementally developing them. While this will not always happen, using the scenarios as the basic entity for partitioning does assist in this.

Prioritising and partitioning the system based on scenarios is quite a coarse granularity approach and may not provide a very accurate view of the relative sizes of different partitions. Therefore, there are a number of ways in which the current approach could be modified, and perhaps improved. In particular a more complex algorithm could be developed in order to obtain better initial partition sizes, that considered not just the numbers of scenarios, but also the numbers of resulting goals, percepts and actions. Also, the algorithm could be modified to incorporate a scenario measure based on number of goals new to the priority level, with some adjustment for goals that were shared amongst scenarios. However it is not clear that this would lead to substantial improvement. We feel that it is better to first gain some substantial experience with the simpler approach, and to extend it based on experienced difficulties. It is particularly pleasing, that taking a previously developed example, and then applying the prioritisation algorithms to it led to such a clean separation which would clearly have facilitated incremental development. We plan to obtain further experience with this approach, in collaboration with our industry partners, and to then modify aspects of it as needed. We note that the basic approach easily allows for a range of refinements.

5 Related work

Requirements prioritisation is recognised as an important, but difficult activity in software development process since it lays the foundation for release planning/system partitioning (Lehtola et al., 2004). Having a systematic requirements prioritisation process is a challenge because such process involves decision making, domain knowledge, and estimation skills (Wiegers, 1999).

Unfortunately, there is little agreement within industry as to how, when, and why requirements should be prioritised. Requirements can be prioritised along many different dimensions, such as: *stakeholders preference, business value, risk avoidance, cost, difficulty, and frequency of use* (Firesmith, 2004). Also, there are a number of prioritisation dimensions that target a particular application types (such as systems where *reliability* is one of the major concerns e.g. military systems). For example, Coit et al. defined a reliability-prediction prioritization index (RPPI) in order to provide a relative initial requirements rankings based on their potential for improving the system-level reliability (Coit and Jin, 2001).

In addition, various techniques can be used to determine, and develop a consensus regarding the priorities of the requirements. According to Firesmith (Firesmith, 2004) the most widely used prioritisation techniques are pair-wise comparisons, scale of 1-to-10 rankings, and voting schemes. Additionally, there are a number of more advanced techniques, for example Avesani (Avesani et al., 2005) used various machine learning techniques to induce requirements ranking approximation for the available data (initial ranks). Also, Xiaoqing Liu et al. (Liu et al., 2004) developed a framework that assigns initial priorities using an inter-perspective relationship matrix. This matrix facilitates the prioritisation process by assigning priorities to the requirements based on their relationships captured by multiple stakeholder perspectives. Such techniques can be used to deal with very large sets

of requirements and associated priority rankings.

Note that any of the above mentioned prioritisation approaches can be readily added to our partitioning mechanism in order to provide for a more systematic allocation of initial ranking. The only constraint is that the priority rankings must be integers ranging from 1 to 5 with value 1 indicating the highest priority and 5 indicating the lowest.

Another important issue is the granularity at which the requirements prioritisation occurs. A large-sized project can have thousands of functional requirements, hence we need to choose an appropriate level of abstraction for the prioritisation procedure. Traditionally this can be at the use case, feature, or individual functional requirement levels (Wiegers, 1999).

We decided that the scenario level is most suited for Prometheus. This decision was influenced by the practices prescribed by the Unified Software Development Process (UP) (Jackobson et al., 1999). One of the activities in the Requirements workflow of UP is 'Prioritize Use Cases'. The purpose of this activity is to determine which of the use cases should be developed in early iterations, and which can be developed in later iterations. The Rational Unified Process (RUP) which is a specific and detailed instance of UP also regards use case prioritisation activity as one of the most important in the Requirements discipline (Kroll and Kruchten, 2003; Kruchten, 2004).

We have adopted a similar strategy to UP/RUP where we prioritise scenarios rather than goals, but in our case, the process of deciding on what to be developed within a given iteration is automated based on the algorithms described in the previous sections. As such, the presented approach requires less effort than the existing manual approaches such as (Blahunka, 1999). Also note that there are commercial products, such as the ReleasePlanner (http://www.releaseplanner.com/), that assist in release planning by considering factors not covered in this research (e.g. consideration of different resource types, budget constraints, regulatory considerations). Such products mainly target business-level operations (such as strategic product and release planning, road-mapping, project portfolio management, etc.) and are not suited for the task of automated system partitioning at the lower level.

None of the existing agent development methodologies support the prioritisation of system specification components, such as goals or scenarios.

6 Conclusions

This paper has described a prioritisation process which supports incremental development of an agent based system. If the Prometheus design process is used, the scoping can be automatically supported, and consistency enforced, within the Prometheus Design Tool. The process is flexible in that it allows the developer to specify the desired size ranges of the three partitions, and it also allows prioritisation mechanism to be applied at any stage after the initial system specification. There are plans to verify this work by using it in collaboration with our industry partner to obtain experience in practice, with substantial applications. We also hope that by making it available within the Prometheus Design Tool which is used by external groups, that we will obtain feedback on its usefulness as well as on modifications which may improve estimations of size.

The process of partitioning system functionality into manageable, coherent pieces is a crucial part of managing large software development projects. The presented approach provides structured and automated support for incremental development according to the proposed partitions. To the authors' knowledge there is currently no other work on system partitioning, or other mechanisms for supporting incremental development in an automated and structured fashion in agent systems. In future work it will be important to develop mechanisms for estimations of development time that can be used to more accurately determine partition sizes if desired. However detailed cost estimation is usually a time consuming procedure, so the coarser granularity approach to partitioning may well be a useful approximation, even if more accurate costing is available. While the described mechanisms are developed for Prometheus and integrated within the Prometheus Design Tool, the basic principles could readily be adjusted to suit any agent development methodology which has suitable structured relationships between design artefacts.

Acknowledgements

We would like to acknowledge the support of the Australian Research Council (ARC) and Agent-Oriented Software, under grant LP0453486 "Advanced Software Engineering Support for Intelligent Agent Systems", ARC Linkage Grant, 2004-2006.

References

Avesani, P., Bazzanell, C., Perini, A., and Susi, A. (2005). Facing scalability issues in requirements prioritization with machine learning techniques. In *Proceedings of the 13th IEEE Conference on Requirements Engineering (RE'05)*, Paris, France.

Blahunka, R. (1999). Iterative project scoping: An approach to sensibly selecting business requirements for iterative dss deployment. *DM Review Magazine*, 3(6).

Bresciani, P., Giorgini, P., Giunchiglia, F., Mylopoulos, J., and Perini, A. (2004). TROPOS: An agent-oriented software development methodology. *Autonomous Agents and Multi-Agent Systems*, 8(3):203–236.

Bussmann, S., Jennings, N. R., and Wooldridge, M. (2004). *Multiagent Systems for Manufacturing Control*. Springer-Verlag.

Coit, D. and Jin, T. (2001). Prioritizing system-reliability prediction improvements. *IEEE Transactions on Reliability*, 50(1):17–25.

Firesmith, D. (2004). Prioritizing requirements. *Journal of Object Technology*, 3(8):35–47.

Huget, M.-P. and Odell, J. (2004). Representing agent interaction protocols with agent uml. *Proceedings of the AAMAS04 Agent-oriented software engineering (AOSE) workshop*.

IEEE-Std830 (1998). *IEEE Std 830-1998, IEEE Recommended practice for software requirements specifications*. IEEE.

Jackobson, I., Booch, G., and Rumbaugh, J. (1999). *The Unified Software Development Process*. Addison-Wesley, Reading, USA.

Kroll, P. and Kruchten, P. (2003). *The Rational Unified Process Made Easy*. Addison-Wesley, Reading, USA.

Kruchten, P. (2004). *The Rational Unified Process: An Introduction, Third Edition*. Addison-Wesley, Boston, USA.

Lehtola, L., Kauppinen, M., and Kujala, S. (2004). Requirements prioritization challenges in practice. In *Proceedings of the 5th International Conference on Product Focused Software Process Improvement*, pages 497–508, Kansai Science City, Japan.

Liu, X., Veera, C., Sun, Y., Noguchi, K., and Kyoya, Y. (2004). Priority assessment of software requirements from multiple perspectives. In *Proceedings of the 28th Annual International Computer Software and Applications Conference*, Honk Kong.

Padgham, L. and Winikoff, M. (2004). *Developing Intelligent Agent Systems: A Practical Guide*. John Wiley And Sons Ltd, West Sussex, England.

Padgham, L., Winikoff, M., and Poutakidis, D. (2005). Adding debugging support to the Prometheus methodology. *Journal of Engineering Applications in Artificial Intelligence*, 18(2).

Perepletchikov, M. and Padgham, L. (2005a). Systematic incremental development of agent systems, using Prometheus. In *Proceedings of the 1st International Workshop on Integration of Software Engineering and Agent Technology (ISEAT05)*, Melbourne, Australia.

Perepletchikov, M. and Padgham, L. (2005b). Use case and Actor driven Requirements Engineering: An evaluation of modifications to Prometheus. In *Proceedings of the Fourth International Central and Eastern European Conference on Multi-Agent Systems (CEEMAS'05)*, Budapest, Hungary.

Rolland, C., Souveyet, C., and Achour, B. (1998). Guiding goal modelling using scenarios. *IEEE Transactions on Software Engineering, Special Issue on Scenario Management*, 24(12):1055–1071.

van Lamsweerde, A. (2001). Goal-oriented requirements engineering: A guided tour. In *Proceedings of the 5th IEEE International Symposium on Requirements Engineering*, pages 249–263, Toronto, Canada.

Wiegers, K. E. (1999). First things first: Prioritizing requirements. *Software Development*, 7(9).

Wiegers, K. E. (2000). Karl Wiegers describes 10 requirements traps to avoid. *Software Testing and Quality Engineering*, 2(1).