Systematic incremental development of agent systems using Prometheus

Perepletchikov, Mikhail; Padgham, Lin

Perepletchikov, M., & Padgham, L. (2005). Systematic incremental development of agent systems using Prometheus. Proceedings of the Fifth International Conference on Quality Software (QSIC 2005), 413–418. https://doi.org/10.1109/QSIC.2005.60
Document Version: Accepted Manuscript

Published Version: https://doi.org/10.1109/QSIC.2005.60

Citation: Perepletchikov, M and Padgham, L 2005, 'Systematic incremental development of agent systems using Prometheus', in Proceedings of the Fifth International Conference on Quality Software (QSIC 2005), Melbourne, Australia, 19-20 September 2005.

Systematic Incremental Development of Agent Systems, using Prometheus

Mikhail Perepletchikov and Lin Padgham
RMIT University, School of Computer Science and Information Technology
{*mikhailp, linpa*}*@cs.rmit.edu.au*

Abstract

This paper presents a mechanism for dividing an agent oriented application into the three IEEE defined scoping levels of essential, conditional and optional. This mechanism is applied after the initial system specification, and is then used to direct incremental development with three separate releases. The scoping described can be applied at any stage of a project, in order to guide consistent scoping back if such is needed. The three levels of scoping that are used are consistent with the approach used in many companies. The approach to scoping requires that scenarios are prioritised manually on a five point scale. All other aspects are then prioritised automatically, based on this information. The approach used allows a developer to indicate what size partitions - based on number of scenarios - are required for each scoping level. The mechanisms are applied to the Prometheus development methodology and are integrated into the Prometheus Design Tool (PDT).

1. Introduction

It is currently well accepted in Software Engineering (SE) that an iterative and incremental approach to software development is far preferable to a waterfall approach. The Rational Unified Process (RUP) [6, 7] 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 modeling, 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 [7].

Development of multi agent systems is no different than other software development with respect to the advantages of an incremental and iterative process. The Prometheus methodology for developing agent systems [9] suggests that an iterative approach should be applied, but has no specific mechanisms to support such. The style of Prometheus in general is to provide structured support for the development process. This paper proposes some mechanisms for splitting a project into multiple scoping levels, and using these to provide automated support for incremental development within the Prometheus framework.

In the following sections we first briefly introduce the Prometheus methodology, including some modifications that have been made to the System Specification phase of Prometheus to better support a systematic and balanced specification, appropriate as the basis for the scoping activity. Next, we describe the scoping process and algorithms. This leads to a system specification with three levels of scoping, similar to the three prioritisation levels used by many organisations [3, 14], and compatible with the IEEE standards [4]. Finally, we describe how the scoping supports incremental development through the phases of architectural and detailed design, as well as implementation and testing. The (refined) Prometheus Design Tool (www.cs.rmit.edu.au/∼mikhailp/research/PDT.2.0.jar) assists by providing automated support in developing according to the scoping levels. We then briefly review related work in prioritisation of requirements with a view to scoping, outside of Agent-Oriented SE[1].

2. Overview of the Prometheus methodology

Prometheus consists of three main design phases, plus implementation and testing which are not really covered in any detail. The design phases are: (i) System Specification, (ii) Architectural Design, (iii) Detailed Design.

The scoping that we introduce is based on a full system specification, as it is not really possible to scope and prioritise things that have not yet been specified. We have modified the original Prometheus specification phase to provide greater structure, and thus a better basis for scoping.

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

(Revised) System Specification. The revised system specification process consists of the following steps:

1 To the author's knowledge there has not been any work done in this area specifi cally within AOSE.

(i) Identification of *actors* and their interactions with the system; (ii) Developing *scenarios* illustrating the system's operation; (iii) Identification of the *system goals* and subgoals; (iv) Specifying the interface between the system and its environment in terms of *actions*, *percepts* and any *external data*; (v) Grouping goals and other items into the basic *functionalities* of the system.

The first modification of the original Prometheus process is the addition of *actors* for which the system has relevance. These actors are any persons or roles 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 is not new in AOSE, for example the Tropos methodology also uses actors in its Requirements Phases [2].

Use cases are then developed for each actor that will interact with the system, in much the same way as for 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*. This process results in improved structure for identifying percepts and actions. Also, each use case becomes an initial scenario.

The second modification is that scenarios are linked to goals in a similar way to that of the Goal-Scenario coupling framework (GSCF) [11] which is based around the notion of a Requirement Chunk (RC) (a pair of <Goal, Scenario>). Since a goal can be described as a contextual property of a scenario, and scenarios show concrete steps for achieving a goal, the correlation between goal-oriented and scenario-driven approaches can bring a number of benefits as described in [11]. The linkage used in the revised Prometheus is not as strict as the one used in GSCF[2]. We use a unidirectional coupling where each scenario necessarily has a goal which is linked to it (where we also use the same name), but the more specific goals may not require a scenario.

Each initially identified scenario is then developed, with a number of detailed steps, where each step is a *goal*, *scenario*, *action* or *percept*. With the coupling identified above, any nested scenario identified, automatically introduces a goal. Goals introduced as steps may warrant development of a scenario, in which case the goal step is automatically modified to be a scenario step.

Initial goals are identified via the initial use-cases as described above, and also by examining the initial system description. Further goals are then identified by a process of abstraction and refinement[12]. For each goal, we ask the question *how?* and *why?*, thus identifying new goals, and forming a goal hierarchy.

One further modification that is introduced is the notion of two kinds of goal refinement: *AND-refinement* and *OR-refinement* as described by van Lamsweerde [12]. If a goal is AND-refined, we mean that 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.

Finally, after goals and scenarios are sufficiently developed, goals are grouped into functionalities, where similar goals are grouped together. Actions and percepts are also allocated to functionalities. Scenarios are then annotated with information about which functionality each step belongs to.

A detailed description of modifications that have been made to the System Specification phase of Prometheus can be found in [10].

Architectural Design. In architectural design there are three main steps: (i) Determining agent types; (ii) Determining interaction protocols between agents; (iii) Describing the overall system structure.

Agent types are determined by analysing possible groupings of functionalities, and determining the most preferred grouping. Groupings should have high cohesion within each agent and low coupling between agents. There are various criteria which should be taken into account when determining which functionalities should be grouped together.

Interaction protocols are developed based on an initial transformation from scenarios to interaction diagrams. These interaction diagrams are then generalised to provide the protocols which should be a complete specification of the interaction between agents.

The system overview diagram provides an overview of system structure, showing which agent types are involved in which interaction protocols.

Detailed Design. Detailed design focusses on the internal functioning of each agent type. The main aspects of detailed design are: (i) Identification and description of agent capabilities, resulting in an *agent overview diagram*; (ii) Description of internal agent processing, via *process diagrams*; (iii) Development of *plans* and *events*.

The agent overview diagram is similar in structure to the system overview diagram, but focusses 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 use case/s.

2 In GSCF, every goal must be linked to a scenario, and vice-versa.

3. Scoping levels

The scoping process that we propose uses the use case scenarios as the primary artifact for scoping. In collaboration with clients, users or managers, the developer is required to assign a ranking of one to five to each use case scenario that has been identified. Where a scenario is included as a step in the (parent) use-case scenario it is given the ranking of the parent scenario. Where a scenario has been included in more than one parent scenarios it is given the ranking of the highest ranked parent.

We will eventually arrive at three scoping levels, according to common practice. However we start with five, in order to provide a broader range of options and to assist in avoiding the trap of insufficiently prioritised requirements, where "more than 90% of the requirements are classified as high priority" [14].

Once all use cases have been assigned a priority, we apply a scoping algorithm which attempts to partition the use cases into three suitably sized partitions based on the rankings obtained. By default we assume that the *essential* partition should contain 35-45% of use-case scenarios, the *conditional* partition should contain 20-40%, and the *optional* partition should contain 25-35%. These percentages were chosen arbitrary, and can be modified for particular situations or preferences. The scoping algorithm will attempt to stay as close to these partition sizes as is possible, given the rankings supplied.[3]

The rankings from one to five give us six different possibilities for how to assign ranks to partitions ([1,234,5],[12,34,5],[123,4,5],[1,23,45],[1,2,345],[12,3,45]). 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 1.

If more than one option has all partitions within range, we by default choose the one with the smallest essential partition. However, it is also possible to present the options visually to the developer 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, we could use the actors' rankings[4] as one of the alternatives for deciding a particular scenario scoping option in case we have options that are equally good.

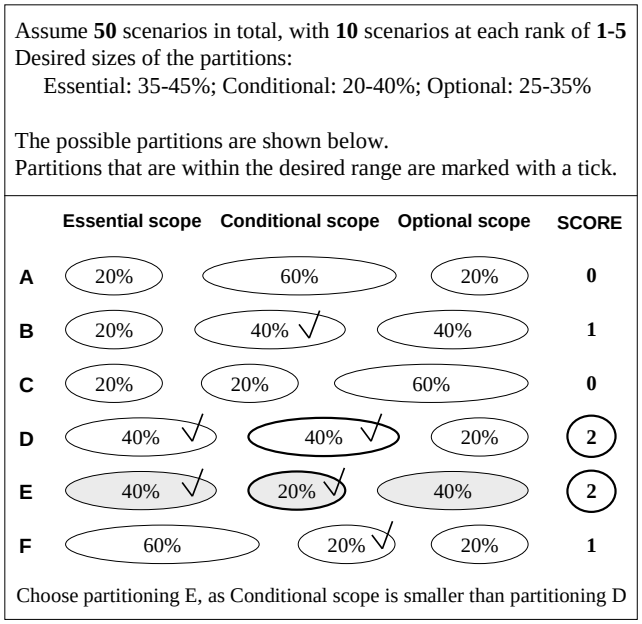

Figure 1. The process of selecting the 'best' scoping option

Although we have not done it, it would be possible to provide support whereby the developer could move particular scenarios between partitions in order to achieve better distributions. The tool could allow scenarios of the same rank to be placed in different partitions, while ensuring that the relative partial ordering obtained from the original ranking is maintained.

Once scenarios are assigned to partitions, other artifacts 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.

Functionalities are assigned to a partition according to the highest priority goal that is included in the functionality. However functionalities will be developed incrementally, according to the scoping.

3 We maintain the constraint that ranking 1 should map to the *essential* partition, so if too many level one priorities are given, it may not be possible to stay within the recommended partition size for the essential partition. Similarly, ranking 5 should always map to the *optional* partition, therefore it may not be possible to stay within the recommended partition size for the optional condition.

4 The refi ned PDT supports the prioritisation of actors.

Scoping of Goals. Many goals can be prioritised directly according to associated use-case scenarios. If the goal is linked to a scenario, the scope of this goal is equal to the scope of the linked scenario. Where a goal is a step in a scenario it is given the scope level of the scenario it is a step in. Where a goal receives different scope levels from different scenarios, the highest scope level is used.

The assignment of remaining goals to a scoping partition involves considering both parent 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 scoping rules we assume that all goals are covered by scenarios.

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

- **All** goals must be in a scoping level 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 scoping level at least as high as their parent.
- **Some** OR-refined subgoal must be assigned to a scoping level 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 filled by allocating all unallocated goals with AND-refined subgoals, and all AND-refined subgoals, as illustrated in Figure 2. If AND-refined subgoals have differing scoping levels, this implies that some of the goals are used in a context separately than as steps within the AND-refined parent. As higher scoping over-rides lower scoping, the lower scope within the set must be the case where all subgoals are required. This should then be the scope of the parent goal. The system will check that all subgoals are used within at least one expanded scenario[5], and if this is not the case a warning will be generated.

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

5 An expanded scenario is one where all scenario steps are expanded out.

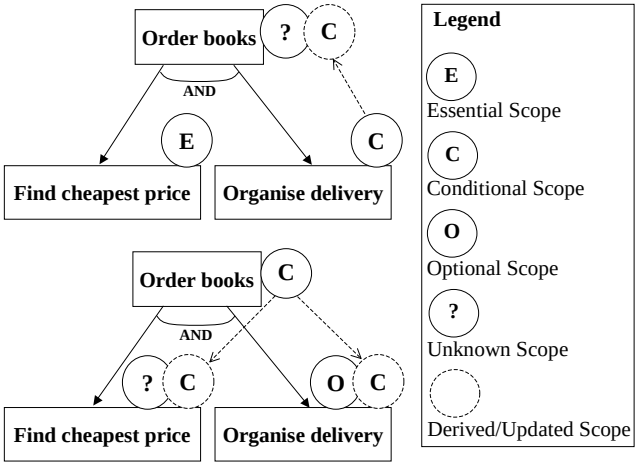

Figure 2. Allocations involving AND-refinement

When an unallocated top level goal is OR-refined we assign it to the scoping of its highest scoped 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 functionality is receiving its scoping 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 scoping fulfills the third constraint above. Consequently, if a sibling subgoal is already assigned to a scope at least as high as the parent, then the goal under consideration is assigned to the lowest scope. Otherwise it is assigned to the scope of the parent goal. Figure 3 illustrates these allocations.

Scoping of Functionalities. The priority of a functionality is determined by the highest priority goal included in this functionality. For example, if the Functionality X contains three goals with the scope levels *essential*, *conditional*, and *optional* respectively, the scope level of Functionality X will be *essential*. However, the functionality will be only developed to the extent required within each scoping level. The one exception to this is the case where the scope of the functionality 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 scope was justified.

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

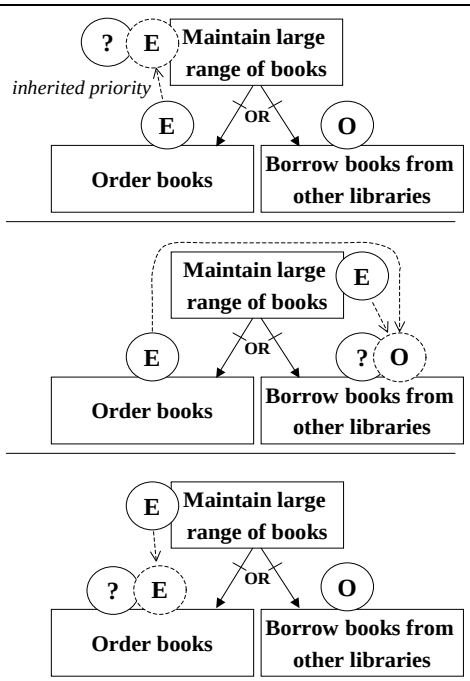

Figure 3. Allocations involving OR-refinement

4. Iterative development of scoping levels

Dividing the system into three clearly separated levels of scope, results in the identification of three separate releases. In the first incremental iteration (release) we develop the full system specification of an agent system. We then apply the scoping 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. The third release covers the *optional* features.

In moving to architectural design, it is necessary to decide the agent types within the system, based on the full set of functionalities. Once this is done, agents can themselves be scoped. The agent type itself needs to be placed within the scoping partition of its highest scoped functionality, as such, there is a potential risk of having too many agent types in the essential scope. However, the agents are also scoped internally, so that only those aspects which relate to goals and scenarios within the essential scope are developed during the first release.

At the architectural design phase, it is the development of protocols which is most affected by the scoping. Interaction diagrams should be developed only for scenarios in the prioritised level of scope. In generalising the interaction diagrams to protocols, it will be necessary to generalise only with respect to the goals within scope. Following iterations will then need to revise and expand the protocols. This will often involve adding alternative options, optional constructs or references to additional protocols.

It can be helpful in developing the protocols, to annotate with goals on each agent lifeline, related to the activity the agent is performing which results in the following message. If this method is used then all goals should be in the relevant scope. Messages are then also defined in line with the reduced, appropriately scoped protocols.

In developing the detailed design, only the aspects of the agent that have to do with the goals that are in scope 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 scope. Process diagrams follow directly from the reduced protocols, and thus are also developed according to the relevant scoping partition.

Including scoping within the PDT allows the scoping to be fully integrated, and to support incremental development in a structured manner. In the process of developing the scoping partitions, the developer 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 scoping partition, and all aspects of the system which are in a lower partition can be deactivated.

Scoping can be applied at any stage - and indeed if things are added to the specification, then scoping should be recalculated. The ability to do scoping at any stage results in a flexible development model since the developer can apply the scoping procedure at any point of system design. For example, it is possible to fully design the system, and only then apply scoping. This will then be propagated through the system, allowing implementation and testing to proceed in incremental iterations.

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 [14]. Again, we can apply the scoping procedure to the final design artifacts in order to determine the less important features.

5. Related work

Requirements prioritisation is recognised as an important, but difficult activity in software development process since it lays foundation for release planning/system scoping [8]. Having a systematic requirements prioritisation process is a challenge because such process involves decision making, domain knowledge, and estimation skills [13].

Unfortunately, there is little agreement within industry as to how, when, and why requirements should be prioritised. Requirements can be prioritised along many different di-

mensions, such as: stakeholders preference, business value, risk avoidance, cost, difficulty, and frequency of use [3]. In addition, various techniques can be used to determine, and develop a consensus regarding the priorities of the requirements. The following prioritisation techniques are the most widely used as stated by Firesmith [3]: pair-wise comparisons, scale of 1-to-10 rankings, and voting schemes.

Another important issue is the granularity at which the requirements prioritisation occurs. A large-sized projects can have thousands of functional requirements, hence we need to choose an appropriate level of abstraction for the prioritisation procedure. This could be at the use case, feature, or individual functional requirement levels [13].

We decided that the use case level is most suited for Prometheus. This decision was influenced by the practices prescribed by the Unified Software Development Process (UP) [5]. 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 [6, 7].

We have adopted a similar strategy to UP/RUP where we prioritise use cases rather than actual requirements, 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 lesser effort than the existing manual approaches (such as the one described in [1]).

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 scoping 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 scoping to be applied at any stage after the initial system specification. There are plans to verify this work by using it on real systems and evaluating the usefulness of the proposed mechanism via empirical studies.

A revised and more structured version of the Prometheus system specification phase is also briefly presented. This has been shown experimentally to result in more consistent and well developed system specifications, thus providing a better base for the scoping activity.

Scope management is a crucial part of managing large software systems. The presented approach provides structured and automated support for incremental development according to scoping priorities. To the author's knowledge there is currently no other work on cost estimations, scoping, or other mechanisms for supporting incremental development in an automated and structured fashion in agent systems. 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 artifacts.

References

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

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

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

[4] IEEE-Std830. *IEEE Std 830-1998, Recommended practice for software requirements specifi cations*. IEEE, 1998.

[5] I. Jackobson, G. Booch, and J. Rumbaugh. *The Unifi ed Software Development Process*. Addison-Wesley, Reading, USA, 1999.

[6] P. Kroll and P. Kruchten. *The Rational Unifi ed Process Made Easy*. Addison-Wesley, Reading, USA, 2003.

[7] P. Kruchten. *The Rational Unifi ed Process: An Introduction, Third Edition*. Addison-Wesley, Boston, USA, 2004.

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

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

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

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

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

[13] K. E. Wiegers. First things fi rst: Prioritizing requirements. *Software Development*, 7(9), 1999.

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