

Chapter 1

THE PROMETHEUS METHODOLOGY*

Michael Winikoff and Lin Padgham

School of Computer Science and Information Technology

RMIT University

GPO Box 2476V

Melbourne, 3001, AUSTRALIA

+61 3 9925 2348

{ winikoff,linpa }@cs.rmit.edu.au

Abstract In this chapter we present the Prometheus methodology for building agent-oriented software systems. Our goal in developing Prometheus was to have a process with associated deliverables which could be used by industry practitioners and undergraduate students without a previous background in agents. As a result, the Prometheus methodology aims to be detailed and complete, as well as being general-purpose and having tool support. Prometheus comprises three phases: system specification, architectural design, and detailed design. The Prometheus methodology has been developed over a number of years as a response to both educational and industrial needs. The methodology has been used by industrial practitioners, taught at workshops at a number of conferences, and has been taught to undergraduate and postgraduate students, as well as having been used in student projects. These experiences have been positive and we have noticed an enormous difference in the ability of our students to develop agent systems. Using Prometheus third year undergraduates are able to build reasonable agent systems in a one semester course, something that previously was challenging for graduate students.

Keywords: Agents, Software Engineering, Methodologies.

*Figures 1.1 and 1.2, the Query Late Books Scenario, and some of the text in the Architectural Design section are reproduced by permission of John Wiley & Sons, Ltd. from Lin Padgham and Michael Winikoff, *Developing Intelligent Agent Systems: A Practical Guide to Design*, ISBN 0-470-86120-7 to be published in 2004.

1. Introduction

*Prometheus*¹ is a methodology for developing agent-oriented software systems. Our goal in developing Prometheus was to have a process with associated deliverables which could be used by industry practitioners and undergraduate students to develop intelligent agents systems, without a previous background in agents. To this end Prometheus aims to be *detailed* and *complete* in the sense of covering all the stages of software development as applied to agent systems.

The Prometheus methodology includes three phases:

- The *system specification phase* focuses on (i) identifying the system's *interface*. Since we are dealing with situated agents the interface consists of *percepts* (information from the environment), and *actions*. (ii) determining the system's goals, functionalities, and use case scenarios; along with any important shared data. The outputs from this phase are a set of functionality descriptions, percept and action descriptions, system goals, and use case scenarios.
- The *architectural design phase* uses the outputs from the previous phase to determine which agents the system will contain, how they will interact, and what significant events occur in the environment. The outputs of this phase are a system overview diagram, agent descriptions, agent interaction protocols and a list of significant events and messages between agents.
- The *detailed design phase* looks at the internals of each agent and how it will accomplish its tasks within the overall system. The outcomes of this phase are detailed diagrams showing the internal functionality of each agent and its capabilities, process diagrams that show the internal processing of the agent, as well as descriptions of data structures used by the agent, plans and subtasks and the details of plan triggers.

Figure 1.1 indicates the main design artifacts arising from each of these phases as well as some of the intermediary items and relationships between items. The figure shows the models and dependencies, but does not show the process (although it does imply it).

The development (and revision) of the various models depicted in figure 1.1 is intended to proceed in an iterative fashion (similar to the Rational Unified Process) where in each iteration the focus of the work gradually shifts further down towards implementation, but where it is expected that most iterations will not be exclusively concerned with a single phase and that many iterations will involve revision of previously developed models.

Figure 1.1 is divided horizontally and vertically. The three horizontal regions form the three *phases* of the methodology discussed above. The left-

most region (consisting of scenarios, interaction diagrams, interaction protocols and process diagrams) deals with descriptions of the *dynamic* behaviour of the system. The middle vertical region (data coupling, acquaintance, system overview, agent overview and capability overview) deal with *overviews* of the system while the remaining models (the right region) give detailed descriptions for each entity in the system. Both the middle and right region deal with the static structure of the system.

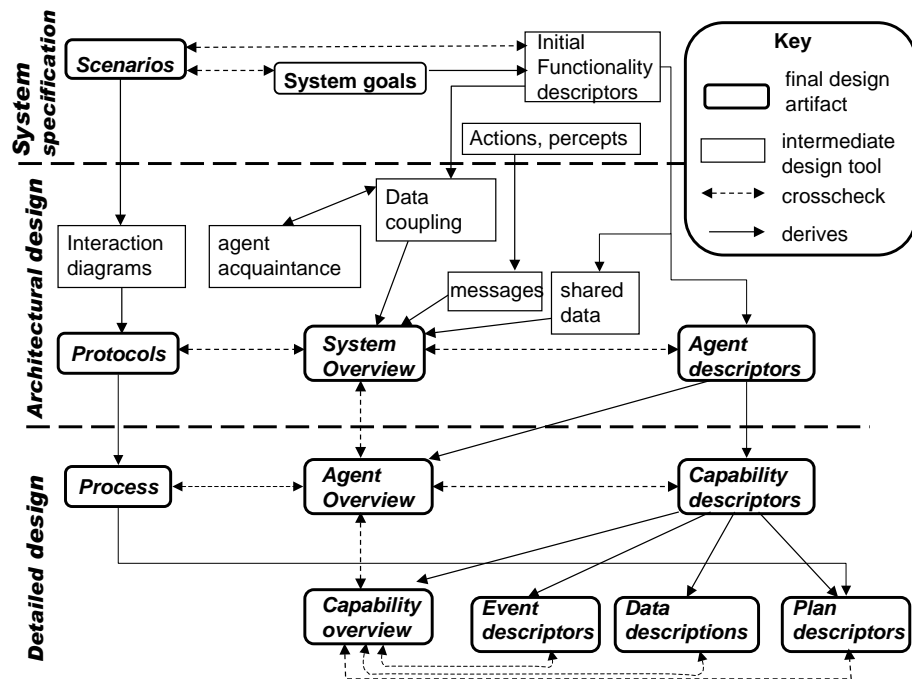


Figure 1.1. Overview of the Prometheus Methodology

Prometheus, like any other methodology, defines a number of system models and notations that are used to describe these models. We describe structural overviews at various levels (system, agent, capability) with a single diagram type. In addition, diagrams are used for showing data coupling and agent acquaintance relationships. Dynamic behaviour is currently described with existing models from UML (Unified Modeling Language) and AUML (Agent UML).

In addition to graphical notations, we use structured textual descriptors (i.e. forms) for describing individual system entities (e.g. agents, functionalities, plans, . . .). We also maintain a data dictionary which is important in ensuring consistent use of names.

It is important to note that Prometheus is a general purpose methodology. In particular, most of the methodology (specification and architectural design) does *not* assume a particular agent architecture. Although the detailed design phase does target a particular family of agent architectures (namely those that achieve goals using a library of plans), this does not make Prometheus special-purpose. Any methodology that addresses implementation needs to have a target platform. For example, Tropos (Bresciani et al., 2002; Giunchiglia et al., 2002; Giorgini et al., 2004) also targets BDI²-like systems, whereas Gaia (Wooldridge et al., 2000; Jennings et al., 2004) avoids the issue by not addressing implementation.

In the following sections we briefly describe the processes and models associated with each of the three phases. Due to space limitations and the desire to describe all of the methodology this chapter cannot do justice to Prometheus. In particular, we cannot describe a running example in detail, and the detailed techniques, that is *how* particular steps in the process are performed, are not described. The description in this chapter is current as of October 2003. For further or up-to-date information see www.cs.rmit.edu.au/agents/SAC.

2. System Specification

System specification consists of three main activities: determining the system's interface to the environment, determining the system's goals and functionalities, and determining scenarios which capture the usage of the system.

Since agents are situated, one of the key things to be captured in the development process is how the agents interact with their environment. Following standard terminology (Russell and Norvig, 1995) we call incoming information from the environment *percepts* and agents' means of affecting the environment *actions*. As discussed in Winikoff et al., 2001 the raw data from percepts may need to be processed in order to obtain things that are a significant event for the agent system. Prometheus prompts the developer to consider such issues. For example a video frame from a camera on a soccer playing robot, may need processing to extract the symbolic objects, such as ball, goal and players, as well as further processing to determine whether anything significant has actually happened - such as a ball having moved since a previous frame, or a ball not appearing where one was expected.

Determining the system's goals and functionalities is done by iterating over the following steps:

- Identify and refine system goals - main and subsidiary
- Group goals into functionalities.
- Prepare functionality descriptors
- Define use case scenarios (and variations)

- Check that all goals are covered by scenarios

An initial set of goals is identified from the initial requirements. These are refined and elaborated into a hierarchy of goals by asking *how* goals will be achieved, and *why*³ goals are being achieved (van Lamsweerde, 2001). For example, if we are designing an online book store we might have a high-level goal *fully online system*. This goal might have associated with it the subgoals *find books online*, *pay online* and *order online*.

Functionalities are limited “chunks” of system behaviour that describe in a broad sense what the system needs to be able to do. We derive functionalities by grouping related goals. For example, given the goals above, we might also have another high-level goal of *purchase books* with subgoals *find books*, *place order*, *make payment*, and *arrange delivery*. *Pay online* and *make payment* are clearly closely related if not identical goals, and are therefore grouped together in a single functionality.

Functionality descriptors capture the name and description of each functionality as well as what events activate it, what goals it achieves, what actions it performs, what percepts it receives, what messages it sends/receives, and what data it uses and produces.

Use case scenarios are complementary to goals in that they show how processes are composed within the system. In developing goals, we typically already are building up scenarios of how these goals will be part of various processes within the system. Scenarios enable us to specify some of this structure, which in turn may help to identify missing goals.

Use case scenarios are based on ideas from object oriented design but are more structured. This structure allows for automated cross checking, and automatic production of partial information for later design artifacts (e.g. protocols).

The core of the use case scenario is the sequence of steps describing a particular example of the system in operation. Each step can optionally have data read and data written noted as well as the functionality that performs that step. Each step can be a GOAL, ACTION, PERCEPT or SCENARIO, as well as OTHER allowing for additional step types, although these cannot be used in automated processing. The following example illustrates the steps of a use case scenario in Prometheus.

Query Late Books Scenario

Trigger: User enquiry

1. GOAL: Determine delivery status
2. GOAL: Log delivery problem
3. ACTION: Request delivery tracking
4. GOAL: Inform customer
5. OTHER: Delay
6. PERCEPT: Tracking information received

7. GOAL: Arrange delivery
8. GOAL: Log books outgoing
9. GOAL: Inform customer
10. GOAL: Update delivery problem

Functionality descriptors, goals, and use cases give different views of a common underlying system. As a result they should be checked for mutual consistency. For example an interaction between functionalities in a use case scenario should also be evident in the interactions field of a functionality descriptor. Also, each system goal should be represented in at least one scenario; all functionalities should be covered; and use case scenarios should cover the important normal uses of the system as well as some error/unusual situations, in order to give an idea of how these will be handled.

3. Architectural Design

The three aspects that are developed during architectural design are:

- 1 Deciding on the *agent types* used in the application. Agent types are formed by grouping a number of functionalities together. Diagrams which we use to assist in the analysis are *data coupling diagrams* and *agent acquaintance diagrams*.
- 2 Designing the overall system structure (described using a *system overview diagram* along with descriptors).
- 3 Describing the interactions between agents using *interaction diagrams* (developed from scenarios) and *interaction protocols* (developed from interaction diagrams).

Deciding on the agent types

One technique that we use to systematically examine the properties which lead to coupling and cohesion is the *Data Coupling Diagram*. Potential groupings are then evaluated and possibly refined using an *Agent Acquaintance Diagram*.

A data coupling diagram (see figure 1.2) consists of the functionalities and all identified data (not only persistent data, but also data the functionalities require to fulfil their job). Directed links are then inserted between functionalities and data, where an arrow pointing towards the data indicates the data is *produced or written by* that functionality, whereas an arrow pointing towards the functionality indicates the data is *used by* the functionality. A double-headed arrow indicates that the functionality both uses and produces the data. Edges between data and data or between functionality and functionality are incorrect syntax (and cannot be drawn in the tool).

The data coupling diagram is used to identify groupings which are linked by their data use. When assessing the diagram visually we are looking for clusters of functionalities around data. This is one important aspect in the analysis of potential groupings of functionalities. It is also used to guide refinements and changes to achieve a cleaner delineation between agents.

Some reasons for grouping functionalities into a single agent are if the functionalities seem to be related or if they share a lot of information. Some reasons for *not* grouping functionalities are if the functionalities are clearly unrelated, or if they exist on different hardware platforms.

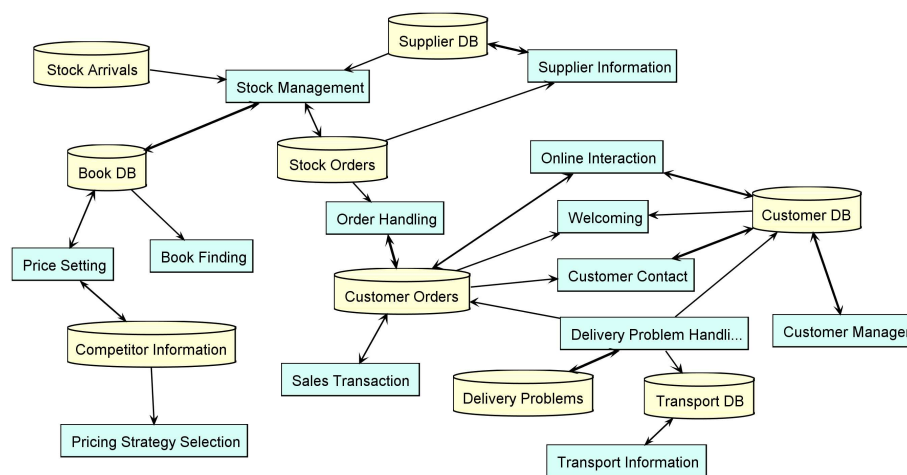


Figure 1.2. Data Coupling Diagram

In order to evaluate a potential grouping of functionalities into agents with respect to agent coupling we use an *agent acquaintance diagram* (see figure 1.3). This diagram represents each of the agent types in the system. Information about agent interaction is extracted from the functionality descriptors and each agent type is linked with the other agent types it interacts with. Links can be decorated with the cardinality of the relationship if desired (e.g. one warehouse agent interacts with many sales agents).

We then analyse the resulting diagram in two ways. One is simply an analysis of the density of the links within the diagram. It is a measure of the ratio of the actual coupling to the maximal possible coupling. If the system has four agents, then each agent could potentially be linked to a maximum of three other agents, giving a total number of $3 + 2 + 1$ possible links. To get the link density we simply count the links and divide by this number. This measure is only one aspect of the analysis. We also consider bottlenecks and other issues.

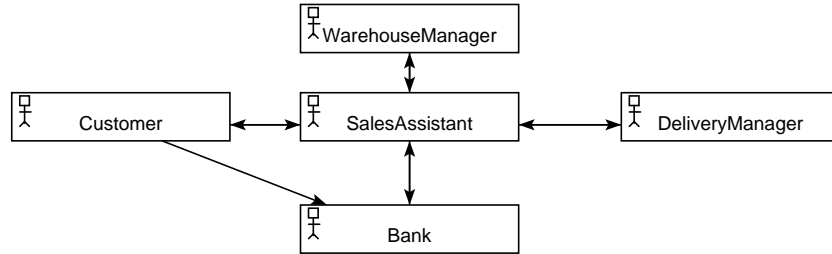


Figure 1.3. Agent Acquaintance Diagram

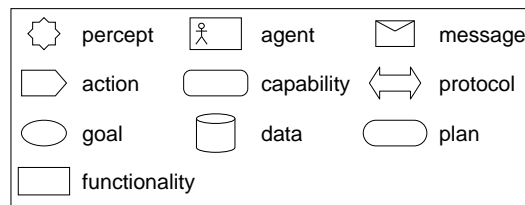


Figure 1.4. Notation used in Overview Diagrams

Designing the Overall System Structure

The system overview diagram is arguably the single most important artifact of the entire design process, although of course it cannot really be understood fully in isolation. The various descriptors provide the more detailed information that may be required.

The notation used in the system overview diagram (and in agent and capability overview diagrams) is a directed graph where nodes represent design entities and directed arcs represent relationships. Figure 1.4 depicts the nodes that are currently used, these correspond directly to the concepts used in the Prometheus methodology.

A syntactically valid overview diagram consists of a set of nodes (excluding goals and functionalities), each labelled with a name, with links between them. We distinguish between “active” nodes (entities that do things - agents, capabilities, and plans) and “passive” nodes (anything else - percepts, actions, messages, protocols, data). A link is valid from an active node to a passive node or from a passive node to an active node. A link is *not* valid from an active to an active node or from a passive to a passive node. An additional constraint is that there cannot be links *to* a percept and there cannot be links *from* an action.

The meaning of links is as follows:

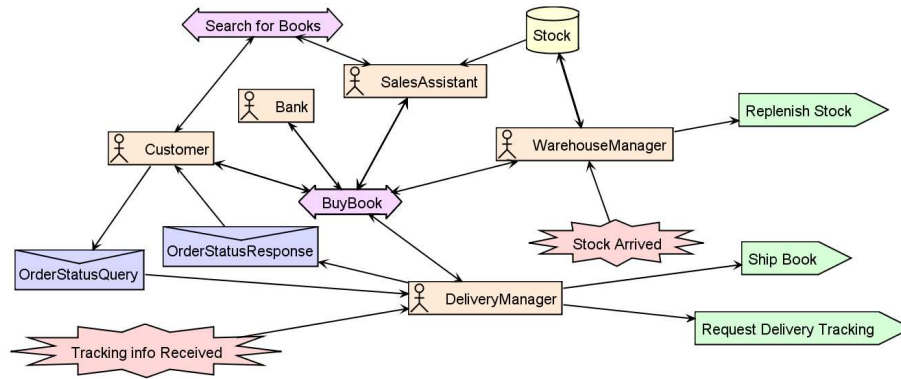


Figure 1.5. System Overview Diagram (excerpt)

- A link *to* a message indicates that the agent type (or capability or plan) sends that message.
- A link *to* a protocol indicates that the entity communicates using the protocol in question.
- A link *to* an action indicates that the entity performs the action.
- A link *to* a data node indicates that the entity writes to it.
- A link *from* a message indicates that the agent type (or capability or plan) receives that message.
- A link *from* a percept indicates that the entity receives the percept.
- A link *from* a data node indicates that the entity reads the data.

When drawing the system overview diagram (see figure 1.5) we start by creating a named agent symbol for each agent type. We also add the percepts and actions at this point.

A data store icon is placed for each persistent data store, with an incoming link from each agent that writes to the data store and an outgoing link from the data store to each agent that directly accesses the data. Double headed links (arrows at both ends) indicate both read and write.

Once interaction protocols have been defined they are added into the diagram and we indicate which agents participate in these protocols.

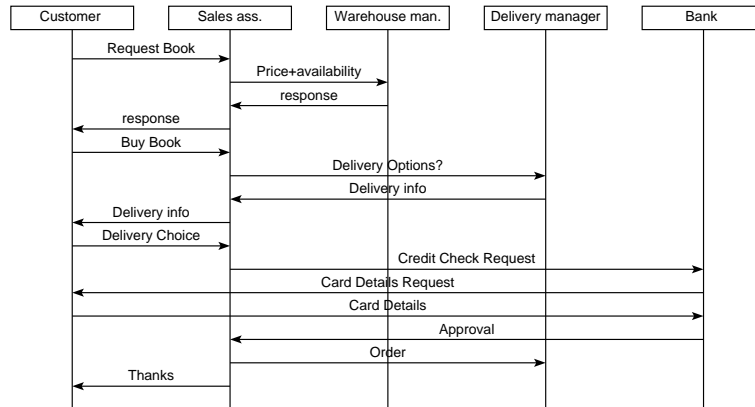


Figure 1.6. Interaction Diagram

Describing the Interactions between Agents

This sub-phase focuses on the system's *dynamic* behaviour by fully specifying the interaction between agents. *Interaction diagrams* borrowed from UML sequence diagrams, are used as an initial representation of agent interaction. Fully specified *interaction protocols* (borrowed from the revised version of AUML currently under development) are the final design artifact.

Interaction diagrams are the same as sequence diagrams of UML except that they show interaction between agents rather than objects. One of the main processes for developing interaction diagrams is to take the use case scenarios developed in the specification phase and to build corresponding interaction diagrams, showing the interaction between agents in a scenario.

As with scenarios, we would expect only to have a representative set of interaction diagrams, not a complete set. In order to have complete and precisely defined interactions we progress from interaction diagrams to protocols which define exactly which interaction sequences are valid within the system.

Developing protocols is done by considering alternatives. For each message (or percept) that an agent receives we ask "what are the possible messages that the agent could send as a response?" We then repeat the process for these messages. Because protocols must show all variations they are often larger than the corresponding interaction diagram and may need to be split into smaller chunks.

An example interaction diagram can be found in figure 1.6 and an example interaction protocol (using the new (draft) AUML notation) can be found in figure 1.7.

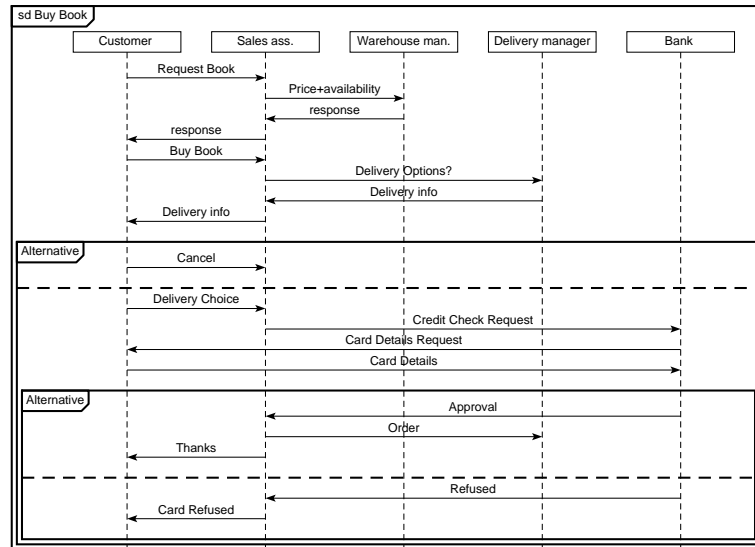


Figure 1.7. Interaction Protocol

4. Detailed Design

This phase deals with the internals of each agent, rather than the system as a whole. We use a hierarchical model so that each agent is broken up into capabilities. Capabilities may be included in more than one agent.

The steps within detailed design are:

- 1 Develop agent overviews (showing interactions between capabilities) and capability descriptors.
- 2 Develop the internal process of an agent from the interaction protocols, described using a variant of UML activity diagrams⁴.
- 3 Develop the internal design of each capability in terms of plans, events, beliefs, and (possibly) sub-capabilities.

The process followed is essentially iterative refinement. We begin by considering for each agent what the agent needs to be able to do. Often, the functionalities that were grouped to form the agent type will be a good starting point for defining the capabilities of that agent type.

We then “connect” up the capabilities. As depicted in the system overview diagram, each agent has incoming and outgoing messages, percepts that it received, actions that it performs, and data that is read and/or written. Each of these connections to an agent is mirrored in the *agent overview diagram* for

that agent type. The agent overview diagram for a given agent type is quite similar to the system overview diagram, but shows interactions between capabilities *within* an agent, rather than between agents within a system. Any messages or percepts that are incoming to an agent in the system overview, must be incoming to some capability (or plan) within that agent in the capability overview diagram. Similarly any actions or messages that are outgoing from an agent in the system overview, must be outgoing from some capability (or plan) within that agent in the capability overview diagram.

Once capabilities (and plans) within an agent have been defined we consider each capability and refine its internals. This process continues until the internal operation of each agent and each capability is defined in terms of plans, messages, data, and other capabilities.

5. Tool Support

Designs for large systems are almost always developed incrementally with many revisions. When revising any artifact, be it documentation, code, or design, it is easy to introduce inconsistencies and minor errors. We have found tool support to be extremely useful for checking and maintaining design consistency across varying levels of detail.

The *Prometheus Design Tool* (PDT) allows a user to enter and edit a design, in terms of Prometheus concepts; check the design for a range of possible inconsistencies; and automatically generate a design report that includes descriptors for each design entity, a design dictionary, and the various diagrams. It also provides descriptor forms which prompt for the various aspects which should be considered. When any aspect of the design is modified, the change is propagated to all levels, although in some cases user input is still required for finalisation.

PDT supports the Prometheus methodology in a number of ways. It supports the process of deriving agent types from functionalities by deriving part of each agent's interface, by cross checking the declared interface of an agent against the functionalities that make up the agent type, and by generating coupling and acquaintance diagrams. It supports the process of developing the internals of agents in the detailed design phase by cross checking an agent's internals against the agent's declared interface, checking the consistency of a plan with its context, and by supporting views of design diagrams at different levels (system overview, agent overview, and capability overview). For more details on tool support for the Prometheus methodology see Padgham and Winikoff, 2002 (this paper discusses an early prototype tool which was also, somewhat confusingly, called PDT).

The Prometheus Design Tool is currently available⁵ and further functionality is under development.

Debugging with Design Artifacts

The Prometheus methodology aims to support the full life cycle, including testing and debugging. David Poutakidis, a research student of the authors, has been working on debugging multi-agent systems, using design artifacts such as those produced by the Prometheus methodology. The central claim in his work is that:

“... the design documents and systems model developed when following an agent-based software engineering methodology [such as Prometheus] can be incorporated in an agent and used at run-time to provide for run-time error detection and debugging.” (Poutakidis et al., 2002)

Specifically, the work described in Poutakidis et al., 2002; Poutakidis et al., 2003 uses *interaction protocols* expressed in AUML (Odell et al., 2000). These are translated into Petri nets and a debugging agent uses these to monitor agent interactions and alert the programmer when a protocol is not followed correctly.

Code Generation

The latest version of the JACK⁶ Development Environment (JDE) includes a design tool that allows Prometheus overview diagrams (based on a slightly older version of the methodology) to be drawn. The JDE also includes a graphical user interface that allows the structure of an agent system to be built by drag-and-drop and by filling in forms.

The JDE supports the Prometheus methodology in that the concepts provided by JACK correspond to the artifacts developed in Prometheus' detailed design phase. It is important to realise that the agent structure described in the JDE generates JACK code that can be compiled and run. This automatic generation of skeleton code from design artifacts is extremely useful, and has encouraged students to do design prior to coding.

6. Experiences with using Prometheus

The Prometheus methodology has been developed over a number of years, as a response to both educational and industrial needs. During its development it has been used by industrial practitioners, taught at workshops at a number of conferences, and has been taught to undergraduate and postgraduate students, as well as having been used in student projects.

We have worked with development of agent software for eight years and have during this time had a wealth of experience in trying to teach students to build such systems. The *Prometheus* methodology has partially grown out of this experience and we have noticed an enormous difference in the last few years, in the ability of our students to develop agent systems. Previously, without a methodology, graduate students would flounder and end up building a

system which made little real use of agents. Using Prometheus, third year undergraduates are now able to build reasonable agent systems in a one semester course.

We have worked with companies that sell agent development platforms (for BDI agents) and they have experienced similar difficulties with teaching their customers how to develop agent based systems, as we have experienced with students. We have worked with Agent Oriented Software⁷ (AOS) both in developing the methodology and also in producing materials for training professional software developers in development of agent systems. The methodology has formed the basis for a course on agent-oriented design that is offered by AOS to industry software developers who are starting to use the JACK intelligent agents development environment (Busetta et al., 1998), and has been successful in introducing them to methods to assist them in design of agent applications. For example, a prototype weather alerting system (Mathieson et al., 2004) developed for the Australian Bureau of Meteorology by AOS used Prometheus overview diagrams (produced using the JDE) to capture the design. The Prometheus overview diagram notation (as implemented in the JDE) is also used within AOS on a range of projects.

The methodology has also been taught to undergraduate students as a class. The class spends roughly half of the semester covering the methodology and the other half introducing the JACK agent programming language and platform. The students were able to design and implement reasonable agent systems in a single semester.

Finally, we have on two occasions given undergraduate students materials on Prometheus (tutorial notes and papers) and, with intentionally limited guidance, had them design (and in one case also implement) an agent system. During the Christmas 2002/2003 vacation a second year student was given a description of the methodology and a description of an agent application (in the area of tourism) and asked to design and build a system. Although there was not enough time to build the system (a considerable amount of time was spent in developing requirements based on an available database of tourist information), the student did produce a detailed design in 8 weeks. During the Christmas 2001/2002 vacation a (different) second year student was given a description of the methodology and a description of an agent application (in the area of Holonic Manufacturing) and asked to build a system. With only (intentionally) limited support, the student was able to design and implement an agent system to perform a Holonic Manufacturing simulation in a period of 8 weeks. This was in marked contrast to projects in the late 1990s where students struggled and required large amounts of help, usually ending up with poorly designed agent systems. The feedback from these undergraduate students was valuable in improving the methodology. The two primary issues identified by the students were the need for tool support and the need to simplify the con-

cepts (Prometheus previously had percepts, events, incidents, messages and triggers). Both issues have since been addressed.

7. Related Work

Naturally Prometheus has some similarities to other agent-oriented methodologies as well as to object-oriented approaches, in particular UML. We review briefly some of these similarities and differences.

Agent-Oriented Software Engineering

There is currently a large amount of work being done in agent-oriented software engineering methodologies (e.g. Brazier et al., 1997; Bresciani et al., 2002; Burmeister, 1996; Burrafato and Cossentino, 2002; Bush et al., 2001; Caire et al., 2001; Collinot et al., 1996; Cossentino and Potts, 2002; Debenham and Henderson-Sellers, 2002; DeLoach et al., 2001; Drogoul and Zucker, 1998; Elammari and Lalonde, 1999; Giunchiglia et al., 2002; Glaser, 1996; Iglesias et al., 1999; Iglesias et al., 1997; Kendall et al., 1995; Kinny and Georgeff, 1996; Kinny et al., 1996; Lind, 2000; Odell et al., 2000; Shehory and Sturm, 2001; Varga et al., 1994; Wooldridge et al., 2000 and, of course, the other chapters in this book).

The Gaia methodology (Wooldridge et al., 2000; Jennings et al., 2004) has, like Prometheus, been developed over a number of years by people experienced in building agent systems. However, we found that the lack of a detailed design process - intentionally absent due to a desire for generality - meant that it did not provide sufficient support for the needs of those we were working with. There are similarities between Prometheus and Gaia for specification and architectural design. Our agent acquaintance diagrams are essentially the same as those used by Gaia, and the roles of Gaia are similar in concept to functionalities in Prometheus, although there are slightly different things which are considered.

The Tropos methodology (Bresciani et al., 2002; Giunchiglia et al., 2002; Giorgini et al., 2004) covers early requirements to detailed design. Its detailed design is oriented very specifically towards JACK as an implementation platform. Compared with Prometheus, Tropos provides an early requirements phase, which Prometheus doesn't (although, it would certainly be possible to adapt Tropos' early requirements phase for use in Prometheus). Prometheus provides a more detailed process - particularly in the architectural design phase. Prometheus also provides tool support and cross checking; tool support for Tropos is only in the form of a diagram editor⁸, rather than the consistency checking and automatic generation of some parts of the design that is part of PDT.

The MaSE methodology (DeLoach et al., 2001; DeLoach, 2004) is one of the few methodologies that has significant tool support. However, MaSE is unsuitable for our purposes since it views agents “. . . merely as a convenient abstraction, which may or may not possess intelligence” (DeLoach et al., 2001, p232). Thus, MaSE (intentionally) does not support the construction of plan-based agents that are able to provide a flexible mix of reactive and pro-active behaviour.

The MESSAGE methodology (Caire et al., 2001; Coulier et al., 2004) extends UML to provide rich models for analysis and design. However, there is less detail on detailed design and implementation.

In addition to general purpose methodologies there are also methodologies such as Adelfe (Picard and Gleizes, 2004) and SADDE (Sierra et al., 2004) which focus on particular application domains or aspects of design. For example, Adelfe extends RUP and UML with activities that support developing complex systems with emergent behaviour. Adelfe has some elements in common with Prometheus: for example characterising the environment, and identifying agent types. However there are also differences: for example goals do not appear to play a significant role in Adelfe.

Because the field is still young, none of the available methodologies can claim extensive use well beyond the group which has developed them. However the widespread development and use of these many new methodologies clearly indicates a need that is starting to be met, to provide more specific design methodologies for building agent systems.

Given the large number of agent-oriented methodologies that have been proposed, there is a growing need for comparisons between methodologies. Dam and Winikoff, 2003 compare MaSE, Prometheus and Tropos using a feature-based approach where the assessment of each methodology against the criteria is validated using a survey of the developers of the methodology (and of students). Dam, 2003 extends this to include MESSAGE and Gaia and also performs a comparative analysis of the models and processes of each of the methodologies. Other comparisons between agent-oriented methodologies include Shehory and Sturm, 2001; Sturm and Shehory, 2003; Cernuzzi and Rossi, 2002 and Sturm and Shehory, 2004.

Object-Oriented Software Engineering

Some approaches to developing agent-oriented systems are based on taking UML⁹ and extending or modifying it, as is done by Odell et al., 2000 as well as others with a slightly different approach (e.g. Papasimeon and Heinze, 2001). This approach is sometimes justified by the argument that agents are a special case of active objects.

Although agents can in some ways be seen as a specialised type of object, we believe that it is important to focus on such concepts as goals, plans and descriptions of situations. This is better supported by a more specialised methodology, borrowing and drawing from UML as appropriate. In our experience, just extending UML does not provide sufficient assistance to start thinking in a different paradigm.

There are significant differences between agent oriented design in Prometheus and in Object-Oriented (OO) methodologies. These differences include the provision of a process for determining the agent types in the system; treating messages as entities in their own right, not just as labels on arcs; distinguishing percepts and actions from messages, and looking explicitly at percept processing; distinguishing beliefs from agents (in OO both are (passive) objects); the identification of agent life-cycle issues; the use of protocols to capture the dynamics of agent interaction; and the use of goals.

Although there are clear differences between Prometheus and OO methodologies, there are also commonalities. Although we do not believe that current OO methodologies are sufficient, we certainly *do* believe that they are relevant – agents are software (Wooldridge and Jennings, 1998), and indeed, many aspects of the Prometheus methodology have been based on OO methods and notations. For example, the scenarios are adapted from OO use-case scenarios; interaction diagrams are used as-is; AUML (itself an extension of UML) is used as-is, and Prometheus follows the RUP approach to applying an iterative process over clearly delineated phases.

In the longer term we see integrating agent methodologies with OO methodologies (and specifically with UML, since it is the de-facto standard notation) as important steps in making agent methodologies accessible to developers. In this respect the work of Papasimeon and Heinze, 2001 and Wagner, 2002 is valuable. However, we believe that given the current state-of-the-art, where the concepts and notations for designing agent systems are not agreed upon, it is best to consider possibilities without the world-view suggested by OO.

8. Future Work

Currently we are working on defining in more detail the processes and techniques associated with goals (in the system specification phase), and how these are systematically propagated through the design to the individual plans of an agent. We are also working on processes and techniques that are used to proceed from use case scenarios to interaction protocols. Further tool development is also ongoing.

In the longer term we plan to extend Prometheus with better support for early requirements as well as for implementation, testing and debugging. Secondly, we would like to enhance the methodology to provide specific support

for the design of *team* based systems (in the sense of Cohen and Levesque, 1991) and systems that are *open*. The work of Collinot et al., 1996; Drogoul and Zucker, 1998; Huget, 2002 will no doubt be relevant to this latter enhancement.

Finally, as the methodology continues to be used by a broader range of software developers we will continue to respond to evaluations and feedback in an effort to support the process of building multi-agent systems.

Acknowledgments

We would like to thank Agent Oriented Software and the Australian Research Council. We would also like to thank James Harland, John Thangarajah, David Poutakidis, Anna Edberg and Christian Andersson of RMIT University, and Ralph Rönquist, Andrew Lucas, Andrew Hodgson, Paul Maisano, and Jamie Curmi of Agent Oriented Software, as well as the many students and workshop participants who have provided comments, examples and feedback.

Notes

1. Prometheus was the wisest Titan. His name means “forethought” and he was able to foretell the future. Prometheus is known as the protector and benefactor of man. He gave mankind a number of gifts including fire. (from www.greekmythology.com)
2. Belief Desire Intention
3. At the moment we do not use this in Prometheus – asking “why” can help derive early requirements that capture why the system is being built.
4. This is not discussed further in this chapter.
5. PDT can be downloaded from www.cs.rmit.edu.au/agents/pdt
6. JACK is a commercial agent development platform developed by Agent Oriented Software. It includes an agent-oriented programming language that is a superset of Java.
7. A commercial company that produces the JACKTM agent development platform, www.agent-software.com
8. Conversation with Anna Perini at AAMAS in July, 2002.
9. Strictly speaking UML is not a methodology but rather a notation. However it is often coupled, either explicitly or implicitly with a methodology such as the Rational Unified Process (RUP).

References

- Bergenti, Federico, Gleizes, Marie-Pierre, and Zambonelli, Franco, editors (2004). *Methodologies and Software Engineering for Agent Systems*. Kluwer Academic Publishing (New York).
- Brazier, F. M. T., Dunin-Keplicz, B. M., Jennings, N. R., and Treur, J. (1997). DESIRE: Modelling multi-agent systems in a compositional formal framework. *Int Journal of Cooperative Information Systems*, 6(1):67–94.
- Bresciani, Paolo, Giorgini, Paolo, Giunchiglia, Fausto, Mylopoulos, John, and Perini, Anna (2002). Tropos: An agent-oriented software development methodology. Technical Report DIT-02-0015, University of Trento, Department of Information and Communication Technology.

- Burmeister, B. (1996). Models and methodology for agent-oriented analysis and design. Working Notes of the KI'96 Workshop on Agent Oriented Programming and Distributed Systems.
- Burrafato, P. and Cossentino, M. (2002). Designing a multi-agent solution for a bookstore with the PASSI methodology. In *Proceedings of the Fourth International Bi-Conference Workshop on Agent-Oriented Information Systems (AOIS-2002)*, Toronto. Available from <http://mozart.csai.unipa.it/passi/>.
- Busetta, Paolo, Rönquist, Ralph, Hodgson, Andrew, and Lucas, Andrew (1998). JACK Intelligent Agents - Components for Intelligent Agents in Java. Technical report, Agent Oriented Software Pty. Ltd, Melbourne, Australia. Available from <http://www.agent-software.com>.
- Bush, Geoff, Cranefield, Stephen, and Purvis, Martin (2001). The Styx agent methodology. The Information Science Discussion Paper Series 2001/02, Department of Information Science, University of Otago, New Zealand. Available from <http://divcom.otago.ac.nz/infosci>.
- Caire, Giovanni, Leal, Francisco, Chainho, Paulo, Evans, Richard, Garijo, Francisco, Gomez, Jorge, Pavon, Juan, Kearney, Paul, Stark, Jamie, and Massonet, Philippe (2001). Agent oriented analysis using MESSAGE/UML. In Wooldridge, Michael, Ciancarini, Paolo, and Weiss, Gerhard, editors, *Second International Workshop on Agent-Oriented Software Engineering (AOSE-2001)*, pages 101–108.
- Cernuzzi, L. and Rossi, G. (2002). On the evaluation of agent oriented modeling methods. In *Proceedings of the OOPSLA 2002 Workshop on Agent-Oriented Methodologies*, pages 21–30, Seattle.
- Cohen, P. R. and Levesque, H. J. (1991). Teamwork. *Nous*, 25(4):487–512.
- Collinot, Anne, Drogoul, Alexis, and Benhamou, Philippe (1996). Agent oriented design of a soccer robot team. In *Proceedings of ICMAS'96*.
- Cossentino, M. and Potts, C. (2002). A CASE tool supported methodology for the design of multi-agent systems. In *Proceedings of the International Conference on Software Engineering Research and Practice (SERP'02)*, Las Vegas. Available from <http://mozart.csai.unipa.it/passi/>.
- Coulier, Wim, Garijo, Francisco, Gomez, Jorge, Pavon, Juan, Kearney, Paul, and Massonet, Philip (2004). MESSAGE: a methodology for the development of agent-based applications. In Bergenti et al., 2004, chapter 9.
- Dam, Khanh Hoa (2003). Evaluating agent-oriented software engineering methodologies. Master's thesis, School of Computer Science and Information Technology, RMIT University, Melbourne, Australia. (supervisors: Michael Winikoff and Lin Padgham).
- Dam, Khanh Hoa and Winikoff, Michael (2003). Comparing agent-oriented methodologies. In Giorgini, Paolo and Winikoff, Michael, editors, *Proceedings of the Fifth International Bi-Conference Workshop on Agent-Oriented Information Systems*, pages 52–59, Melbourne, Australia.

- Debenham, J. and Henderson-Sellers, B. (2002). Full lifecycle methodologies for agent-oriented systems – the extended OPEN process framework. In *Proceedings of Agent-Oriented Information Systems (AOIS-2002) at CAiSE'02*, Toronto.
- Deloach, Scott (2004). The MaSE methodology. In Bergenti et al., 2004, chapter 6.
- DeLoach, Scott A., Wood, Mark F., and Sparkman, Clint H. (2001). Multi-agent systems engineering. *International Journal of Software Engineering and Knowledge Engineering*, 11(3):231–258.
- Drogoul, A. and Zucker, J. (1998). Methodological issues for designing multi-agent systems with machine learning techniques: Capitalizing experiences from the robocup challenge. Technical Report LIP6 1998/041, Laboratoire d'Informatique de Paris 6.
- Elammari, M. and Lalonde, W. (1999). An agent-oriented methodology: High-level and intermediate models. In G. Wagner and E. Yu, editors, Proc. of the 1st Int. Workshop. on Agent-Oriented Information Systems.
- Giorgini, Paolo, Kolp, Manuel, Mylopoulos, John, and Pistore, Marco (2004). The TROPOS methodology: an overview. In Bergenti et al., 2004, chapter 5.
- Giunchiglia, Fausto, Mylopoulos, John, and Perini, Anna (2002). The Tropos software development methodology: Processes, models and diagrams. In *Third International Workshop on Agent-Oriented Software Engineering*.
- Glaser, Norbert (1996). The CoMoMAS methodology and environment for multi-agent system development. In Zhang, Chengqi and Lukose, Dickson, editors, *Multi-Agent Systems Methodologies and Applications*, pages 1–16. Springer LNAI 1286. Second Australian Workshop on Distributed Artificial Intelligence.
- Huget, Marc-Philippe (2002). Nemo: an agent-oriented software engineering methodology. In *Proceedings of the OOPSLA 2002 Workshop on Agent-Oriented Methodologies*, pages 41–53, Seattle.
- Iglesias, Carlos, Garijo, Mercedes, and González, José (1999). A survey of agent-oriented methodologies. In Müller, Jörg, Singh, Munindar P., and Rao, Anand S., editors, *ATAL-98*, pages 317–330. Springer-Verlag: Heidelberg, Germany.
- Iglesias, Carlos Argel, Garijo, Mercedes, González, José C., and Velasco, Juan R. (1997). Analysis and design of multiagent systems using MAS-commonKADS. In *Agent Theories, Architectures, and Languages*, pages 313–327.
- Jennings, N., Kinny, D., Wooldridge, M., and Zambonelli, F. (2004). The gaia methodology. In Bergenti et al., 2004, chapter 4.
- Kendall, E. A., Malkoun, M. T., and Jiang, C. H. (1995). A methodology for developing agent based systems. In Zhang, Chengqi and Lukose, Dickson, editors, *First Australian Workshop on Distributed Artificial Intelligence*.

- Kinny, David and Georgeff, Michael (1996). Modelling and design of multi-agent systems. In *Intelligent Agents III: Proceedings of the Third International Workshop on Agent Theories, Architectures, and Languages (ATAL-96)*. LNAI 1193. Springer-Verlag.
- Kinny, David, Georgeff, Michael, and Rao, Anand (1996). A methodology and modelling technique for systems of BDI agents. In *Seventh European Workshop on Modelling Autonomous Agents in a Multi-Agent World*.
- Lind, Jürgen (2000). A development method for multiagent systems. In *Cybernetics and Systems: Proceedings of the 15th European Meeting on Cybernetics and Systems Research, Symposium "From Agent Theory to Agent Implementation"*.
- Mathieson, Ian, Dance, Sandy, Padgham, Lin, Gorman, Malcolm, and Winikoff, Michael (2004). An open meteorological alerting system: Issues and solutions. In *Proceedings of the 27th Australasian Computer Science Conference (to appear)*, Dunedin, New Zealand.
- Odell, J., Parunak, H., and Bauer, B. (2000). Extending UML for agents. In *Proceedings of the Agent-Oriented Information Systems Workshop at the 17th National conference on Artificial Intelligence*.
- Padgham, Lin and Winikoff, Michael (2002). Prometheus: A pragmatic methodology for engineering intelligent agents. In *Proceedings of the OOPSLA 2002 Workshop on Agent-Oriented Methodologies*, pages 97–108, Seattle.
- Papasimeon, M. and Heinze, C. (2001). Extending the UML for designing JACK agents. In *Proceedings of the Australian Software Engineering Conference (ASWEC 01)*.
- Picard, Gauthier and Gleizes, Marie-Pierre (2004). The ADELFE methodology: Designing adaptive cooperative multi-agent systems. In Bergenti et al., 2004, chapter 8.
- Poutakidis, David, Padgham, Lin, and Winikoff, Michael (2002). Debugging multi-agent systems using design artifacts: The case of interaction protocols. In *Proceedings of the First International Joint Conference on Autonomous Agents and Multi Agent Systems (AAMAS'02)*.
- Poutakidis, David, Padgham, Lin, and Winikoff, Michael (2003). An exploration of bugs and debugging in multi-agent systems. In *Proceedings of the 14th International Symposium on Methodologies for Intelligent Systems (IS-MIS)*, Maebashi City, Japan.
- Russell, Stuart and Norvig, Peter (1995). *Artificial Intelligence: A Modern Approach*. Prentice Hall.
- Shehory, Onn and Sturm, Arnon (2001). Evaluation of modeling techniques for agent-based systems. In *Proceedings of the Fifth International Conference on Autonomous Agents*, pages 624–631. ACM Press.

- Sierra, Carles, Sabater, Jordi, Augusti, Jaume, and Garcia, Pere (2004). SADDE: Social agents design driven by equations. In Bergenti et al., 2004, chapter 10.
- Sturm, Arnon and Shehory, Onn (2003). A framework for evaluating agent-oriented methodologies. In Giorgini, Paolo and Winikoff, Michael, editors, *Proceedings of the Fifth International Bi-Conference Workshop on Agent-Oriented Information Systems*, pages 60–67, Melbourne, Australia.
- Sturm, Arnon and Shehory, Onn (2004). A comparative evaluation of agent-oriented methodologies. In Bergenti et al., 2004, chapter 7.
- van Lamsweerde, A. (2001). Goal-oriented requirements engineering: A guided tour. In *Proceedings of the 5th IEEE International Symposium on Requirements Engineering (RE'01)*, pages 249–263, Toronto.
- Varga, L. Z., Jennings, N. R., and Cockburn, D. (1994). Integrating intelligent systems into a cooperating community for electricity distribution management. *Int Journal of Expert Systems with Applications*, 7(4):563–579.
- Wagner, Gerd (2002). A UML profile for external AOR models. In *Third International Workshop on Agent-Oriented Software Engineering*.
- Winikoff, Michael, Padgham, Lin, and Harland, James (2001). Simplifying the development of intelligent agents. In *AI2001: Advances in Artificial Intelligence. 14th Australian Joint Conference on Artificial Intelligence*, pages 555–568. Springer, LNAI 2256.
- Wooldridge, M., Jennings, N.R., and Kinny, D. (2000). The Gaia methodology for agent-oriented analysis and design. *Autonomous Agents and Multi-Agent Systems*, 3(3).
- Wooldridge, Michael and Jennings, Nicholas R. (1998). Pitfalls of agent-oriented development. In Sycara, K. P. and Wooldridge, M., editors, *Agents'98: Proceedings of the Second International Conference on Autonomous Agents*. ACM Press.