Chapter V

Prometheus: A Practical Agent-Oriented Methodology

Lin Padgham
RMIT University, Australia

Michael Winikoff
RMIT University, Australia

Abstract

We present the Prometheus methodology for designing agents and multi-agent systems. The methodology is intended to be practical; in particular, it aims to be complete and detailed, and to be usable by industrial software developers and undergraduate students. We present the methodology using a case study, describe existing tools that support both design and implementation, and report on experiences with using Prometheus, including our experiences in teaching Prometheus to an undergraduate class over the past few years. These experiences provide evidence that Prometheus is usable by its intended target audience.
Introduction

“One of the most fundamental obstacles to the take-up of agent technology is the lack of mature software development methodologies for agent-based systems” (Luck, McBurney, & Preist, 2004, p. 224).

It is widely accepted in the agent research community that a key issue in the transition of agents from research labs to industrial practice is the need for a mature software engineering methodology for specifying and designing agent systems. In this chapter, we describe the Prometheus methodology that aims to address this need.

In developing the Prometheus methodology, there are a number of motivating considerations that have influenced (and continue to influence) the evolution of Prometheus and the choices made in its development.

- First and foremost, Prometheus is intended to be a practical methodology. As such, it needs to be both complete and detailed. Prometheus has to be sufficiently complete in that it must cover a range of activities from requirements specification through to detailed design; and it has to be sufficiently detailed in that it must provide detailed guidance on how to perform the various steps that form the process of Prometheus.
- Prometheus needs to support (but not be limited to) the design of agents that are based on goals and plans. We believe that a significant part of the benefits that can be gained from agent-oriented software engineering comes from the use of goals and plans to realise agents that are flexible and robust.
- The methodology should facilitate tool support, and tool support should be (freely) available.
- Prometheus needs to be usable by industrial software developers and undergraduate students, not researchers and post-graduate students. In particular, it is important that these groups use the methodology and that their experiences and feedback help guide the development and improvement of the methodology.

These features of Prometheus distinguish it from existing methodologies such as those described in the other chapters of this book, as well as, for example, Brazier, Dunin-Keplicz, Jennings, and Treur (1997); Burmeister (1996); Bush, Cranefield, and Purvis (2001); Collinot, Drogoul, and Benhamou (1996); Drogoul
and Zucker (1998); Elammari and Lalone (1999); Glaser (1996); Kendall, Malkoun, and Jiang (1995); Kinny and Georgeff (1996); Kinny, Georgeff, and Rao (1996); Lind (2001); Sturm, Dori, and Shehory (2003); and Varga, Jennings, and Cockburn (1994). We shall not attempt to compare Prometheus to existing methodologies in this chapter; comparisons of Prometheus with other methodologies can be found in Dam (2003); Dam and Winikoff (2003); and Sudeikat, Braubach, Pokahr, and Lamersdorf (2004). Other comparisons of agent-oriented methodologies include Cernuzzi and Rossi (2002); Shehory and Sturm (2001); Sturm and Shehory (2003), and, of course, Chapter 12 of this book.

In the remainder of this chapter, we describe the Prometheus methodology incorporating a case study, and then discuss the strengths and weaknesses of Prometheus before concluding. Our description of Prometheus is necessarily brief – for a full description see Padgham and Winikoff (2004).

The Prometheus Methodology

Before we present Prometheus, it is important to consider the question “what is a methodology?” This is not just an academic exercise: if we view a methodology as consisting purely of notations for describing designs or as consisting only of a high-level process, then we end up with a very different result.

We adopt a pragmatic stance; rather than debating what should and should not be considered part of a methodology, we simply include in Prometheus everything that we think is necessary. In particular, the Prometheus methodology includes a description of concepts for designing agents, a process, a number of notations for capturing designs, as well as many “tips” or techniques that give advice on how to carry out the steps of Prometheus’ process – in accordance with the more generic definitions in Chapter 1.

Since design is a human activity that is inherently about tradeoffs, rather than about finding the single best design (which often does not exist), it is not possible to provide hard and fast rules. However, it is important to provide detailed techniques and guidelines for carrying out steps.

We would like to stress that Prometheus is a general purpose methodology. Although the detailed design phase makes some assumptions about the agent architecture, the rest of the methodology does not make these assumptions. It is not possible to provide a detailed methodology that proceeds to detailed design and towards implementation without making some assumptions. For example, Tropos, like Prometheus, also targets systems that are based on the Belief Desire Intention (BDI) model.
Agent Concepts

Before we proceed to present the process, notations, and techniques that are associated with the Prometheus methodology, we summarize the concepts used here (Winikoff, Padgham, & Harland, 2001) (see also Chapter 1).

The reason why it is important to consider and discuss concepts is that the concepts are the foundation upon which a software engineering methodology builds. For instance, object-oriented methodologies assume that the designer is familiar with concepts such as objects, classes, and inheritance. The concepts that are appropriate for designing agents are, not surprisingly, different from those that are used for objects. Whereas the concepts of object-oriented programming are well-known, those associated with agent-oriented programming are not, and so we feel that it is useful and important to discuss them.

In considering what concepts are appropriate for designing agent systems, we take as a starting point the definition of an intelligent agent. We take the standard definition of an agent as being software that is situated in an environment, autonomous, reactive to changes in its environment, proactive in its pursuit of goals and social (Wooldridge, 2002). We extend this definition by adding to the list of desired properties of agents being flexible (having multiple ways of achieving goals) and robust (being able to deal with various forms of failure) (Padgham & Winikoff, 2004).

Based on these properties, we identify a number of concepts that are used to design and implement agents. An agent’s interface with its environment is expressed in terms of percepts and actions. Proactive agents pursue goals, and reactive agents respond to events (“significant occurrences”). Agents have beliefs and plans. Finally, social agents use messages to communicate, and these messages are collected in interaction protocols.

Our experience with teaching agent concepts has been that the concept set discussed is sufficient to design BDI style agent systems and also easier to understand than, for example, the standard BDI concepts (particularly intentions).

Prometheus, as a methodology, is intended to be able to support the design of BDI systems, although it is not limited to such; all but the lowest level of design, leading into code, can be used equally well for non-BDI systems. However, the lowest level needs to be modified to accommodate the particular style of implementation platform being targeted. For instance, if building JADE agents, the lowest level would specify behaviours rather than plans, and there would be some changes in details.
Overview of the Prometheus Methodology

We now turn to consideration of the overall structure of the Prometheus methodology. The Prometheus methodology consists of three phases (see Figure 1):

- **System Specification**: where the system is specified using goals and use case scenarios; the system’s interface to its environment is described in terms of actions, percepts, and external data; and functionalities are defined.
- **Architectural design**: where agent types are identified; the system’s overall structure is captured in a system overview diagram; and use case scenarios are developed into interaction protocols.
- **Detailed design**: where the details of each agent’s internals are developed and defined in terms of capabilities, data, events, and plans; process diagrams are used as a stepping stone between interaction protocols and plans.

*Figure 1. The phases of the Prometheus methodology*
Each of these phases includes models that focus on the dynamics of the system, (graphical) models that focus on the structure of the system or its components, and textual descriptor forms that provide the details for individual entities.

In the following sections, we briefly describe the processes and models associated with each of the three phases, as well as discussing some specific techniques. Due to space limitations and the desire to describe all of the methodology, this chapter cannot present all of Prometheus in complete detail along with a full running example. We have done our best to make this chapter a “stand-alone” description of the methodology, but due to space limitations certain aspects of the methodology (such as process diagrams) have not been covered, and the running example that we use is only sketched out. For more information on Prometheus, including a complete description and running example, see Padgham and Winikoff (2004).

Case Study

In order to help explain the Prometheus methodology, we will use a case study of a meeting scheduler system as a running example. This case study was set as the main assignment for our undergraduate class, Agent Oriented Programming and Design, in the second half of 2003. Students were given a high-level description of a calendar system and were required to produce a design using Prometheus and to implement the system using JACK\textsuperscript{2}. The description of the system provided was essentially the following.

*The calendar system supports the scheduling, rescheduling, and cancellation of meetings involving users. When a meeting is requested with certain users the system attempts to find a suitable time. Finding such a time may fail, may require that existing meetings are moved automatically, or may create a double booking for certain users that must be manually resolved.*

*Each user has his or her own application instance with which he or she interacts. These applications can contain multiple agents that interact with the applications of other users to coordinate and schedule meetings.*

*In scheduling meetings, some users may be essential to the meeting and others may not be essential. If an essential user pulls out of a meeting after it is set (or cannot make a proposed time), then the meeting needs to be rescheduled.*

*Setting or rescheduling meetings may be impossible without changing existing meetings. When doing this, care should be taken to...*
avoid creating a “cascade” where, in order to schedule meeting A, meeting B is moved, which creates a clash with meeting C, which is moved creating a clash with D, and so on. Some possibilities for dealing with this situation include: (a) assigning priorities to meetings (and in particular marking some meetings as not movable or only manually movable); (b) using the heuristic that when a clash is created, only a meeting with fewer people (or perhaps with less senior people) will be moved.

The system should allow users to nominate when they are available (and when they are not available) and should also allow for certain constraints to be specified, such as only scheduling a maximum of 4 hours of meetings on a certain day (e.g., to allow time to prepare a lecture). Or, more generally, only scheduling N hours of meetings in a certain time period (M days).

Parts of the design that we present in this chapter are reproduced with permission from the work of a student (John Sietsma) as part of the course and was developed using an earlier version of the Prometheus Design Tool. Specifically, the functionalities, data coupling diagram (slightly modified), agent grouping, system overview diagram, and agent overview diagram are taken from John’s design.

**System Specification**

System specification begins with a rough idea of the system, which may be simply a few paragraphs of description, and proceeds to define the requirements of the system in terms of:

- The goals of the system;
- Use case scenarios;
- Functionalities; and
- The interface of the system to its environment, defined in terms of actions and percepts.

We would like to stress that these are not considered in sequence. Rather, work on one of these will lead to further ideas on another. For example, the goals of the system are a natural starting point for developing use case scenarios. Conversely, developing the details of the scenarios often suggests additional
subgoals that need to be considered. Thus, system specification is an iterative and non-linear process.

**System Goals**

Since agents are proactive and have goals, it is natural to consider using goals to describe requirements. The process for capturing the goals of the system begins by capturing an initial set of goals from the high-level system description. For example, given the description of the case study in the previous section, we can extract an initial set of goals (and subgoals) such as:

- schedule meeting
- reschedule meeting
- cancel meeting
- find suitable time
  - avoid cascade
  - meet user preferences
- manage user availability
- manage meetings
  - track essential participants
- interact with user
- track contacts

These initial goals are then developed into a more complete set of goals by considering each goal and asking *how* that goal could be achieved (van Lamsweerde, 2001), which identifies additional subgoals. For example, by asking how a suitable meeting time can be found, we may realize that this involves a subgoal of negotiating with other users.

As well as finding additional goals, the set of goals is also revised as common subgoals are identified. For example, since the subgoals associated with scheduling and re-scheduling are identical, we choose to merge these two goals into the goal “(re)schedule meeting.” We also make (re)scheduling meetings and cancelling meetings into subgoals of managing meetings. These changes (and a few others) result in the following (revised) set of goals and subgoals.

- manage meetings
  - (re)schedule meetings
• find meeting time
  • propose time meeting user preferences
  • negotiate with other users
• cancel meeting
  o determine essential participants
  o update essential participants
• manage user availability
  o update available times
  o update preferences
• track contacts
  o update contact
  o add contact
  o retrieve contact details
• communicate with other users
• interact with user
  o remind user of meeting
• learn user preferences

Figure 2. Goal diagram (including additional goals from use case scenario development)
This set of goals is not complete; it is merely a first draft. Indeed, as is often the case, developing use case scenarios (Section “Use Case Scenarios”) suggests additional goals that are included in the goal diagram (see Figure 2).

Functionalities

As we revise the groupings of goals, we are attempting to identify what we term “functionalities” – coherent chunks of behavior that will be provided by the system. A functionality encompasses a number of related goals, percepts that are relevant to it, actions that it performs, and data that it uses. Functionalities can be thought of as “abilities” that the system needs to have in order to meet its design objectives; indeed, often functionalities of the system end up as capabilities of agents in the system.

An initial set of functionalities is identified by considering groupings of goals. The functionalities are often then revised as a result of considering the agent types (done as part of architectural design).

In the case study, the top-level goal of manage meetings was made into the Meeting Manager functionality, but the specific subgoals of proposing a meeting time and negotiating were split off into separate functionalities (Meeting Scheduler and Negotiator, respectively) because the associated ability was relatively complex and could be cleanly separated from other aspects of managing meetings. Each of the other top-level goals corresponds to a single functionality, yielding the following functionalities (taken from John Sietsma’s design):

• **Meeting Manager**: Manages meeting information. Encompasses all goals associated with the top-level goal of manage meetings except for propose time meeting user preferences and negotiate with other users.
• **Meeting Scheduler**: Schedules meetings subject to provided constraints and user habits/preferences. Encompasses the goal propose time meeting user preferences.
• **Negotiator**: Negotiates with other users to determine meeting times. This differs from Meeting Scheduler in that Negotiator is about inter-user constraints, whereas Meeting Scheduler is concerned only with the constraints of a single user. Encompasses the goal negotiate with other users.
• **Contact Manager**: In charge of managing contact information for other users. Encompasses the goal track contacts and its subgoals.
• **Contact Notify**: Provides a means of communicating with other users. Encompasses the goal communicate with other users and its subgoals.
• **User Interaction**: Interacts with the user (as opposed to interacting with other users, which is done by Contact Notify). Encompasses the goal *interact with user* and its subgoals except for *remind user of a meeting*.

• **User Notify**: Reminds user of events. Differs from User Interaction in that (a) it uses a range of communication media such as SMS, e-mail and so on, but also (b) waits for conditions to arise and reminds the user, rather than receiving input from the user. Encompasses the goal *remind user of a meeting*.

• **User Information Manager**: Manages information about the user such as his/her preferences. Encompasses the goal *manage user availability* and its subgoals.

• **User Monitor**: Observes user, attempting to learn his habits. Encompasses the goal *learn user preferences*.

Functionalities are described using *descriptors*. These are just textual forms that capture necessary information. In addition to a (brief) natural language description, the descriptor form for a functionality includes the goals that are related to it, the actions that it may perform, and “triggers” – situations that will trigger some response from the functionality. Triggers may include percepts but more generally will include events as well. Finally, the descriptor form also includes notes on the information used and produced by the functionality.

**Use Case Scenarios**

The third aspect of system specification is *use case scenarios*. Use case scenarios (sometimes abbreviated to “scenarios”) are a detailed description of one particular example sequence of events associated with achieving a particular goal or with responding to a particular event.

Scenarios are described using a name, description, and a triggering event. However, the core of the scenario is a sequence of steps. Each step consists of the functionality that performs that step, the name of the step, its type (one of ACTION, PERCEPT, GOAL, SCENARIO or OTHER) and, optionally, the information used and produced by that step.

In addition, scenarios often briefly indicate variations. For example, when scheduling a meeting, a scenario may include a step that selects a preferred time from a list of possible times. A variation of this scenario might be where there is only a single time when all participants are available. In this case, the selection step is omitted.

An example of a scenario descriptor for scheduling a meeting is given in the following.
In developing this use case scenario the additional goals *get user preferences, update user’s diary, inform others of meeting* and *remind others of meeting* were identified.

**System Interface: Actions and Percepts**

Finally, the environment within which the agent system will be situated is defined. This is done by describing the percepts available to the system, the actions that it will be able to perform, as well as any external data that are available and any external bodies of code.

When specifying percepts, we also consider *percept processing*. Often percepts will need to be processed in some way to extract useful information. For example, raw image data indicating that a fire exists at a certain location may not be significant if the agent is already aware of this fire. When agents are situated

<table>
<thead>
<tr>
<th>#</th>
<th>Type</th>
<th>Name</th>
<th>Functionality</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>PERCEPT</td>
<td>Request meeting</td>
<td>1) User Interaction</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>GOAL</td>
<td>Propose time meeting user preferences</td>
<td>2) Meeting Scheduler, MeetDB(R), Prefs(R)</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>GOAL</td>
<td>negotiate with other users</td>
<td>3) Negotiator, MeetDB(R), Prefs(R)</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>GOAL</td>
<td>update user's diary</td>
<td>4) Meeting Manager, MeetDB(W)</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>GOAL</td>
<td>inform others of meeting</td>
<td>5) Contact Notify</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>OTHER</td>
<td>wait for day of meeting</td>
<td>6)</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>GOAL</td>
<td>remind user of meeting</td>
<td>7) User Notify, MeetDB(R), Prefs(R)</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>GOAL</td>
<td>remind others of meeting</td>
<td>8) Contact Notify, ContactInfo(R)</td>
<td></td>
</tr>
</tbody>
</table>

Variations:
- Steps 2-3 may be repeated in order to obtain agreement.
- If agreement on a meeting time is not reached then steps 4-8 are replaced with notifying the user that the meeting could not be scheduled.
- The meeting can be rescheduled or cancelled during step 6 (waiting).

Key:
- MeetDB(R) = Meetings Database read
- Prefs(R) = User Preferences read
- MeetDB(W) = Meetings Database written
- ContactInfo(R) = Contact Information read
in physical environments, then percept processing can be quite involved, for example, extracting features of interest from camera images. Similarly, actions may also be complex and require design.

In the case study, the system interacts with the user (and with other users) and so the percepts correspond to different requests from the user. Actions correspond to sending information back to the user (or other users) and reminding users of meetings.

**Architectural Design**

In the architectural design phase, the focus is on:

- Deciding on the *agent types* in the system: where agent types are identified by grouping functionalities based on considerations of coupling; these are explored using a coupling diagram and an agent acquaintance diagram. Once a grouping is chosen, the resulting agents are described using agent descriptors.
- Describing the interactions between agents using *interaction diagrams* and *interaction protocols*: where interaction diagrams are derived from use case scenarios; these are then revised and generalised to produce interaction protocols.
- Designing the overall system structure: where the overall structure of the agent system is defined and documented using a system overview diagram. This diagram captures the agent types in the system, the boundaries of the system and its interfaces in terms of actions and percepts, but also in terms of data and code that are external to the system.

**Deciding on Agent Types**

Selecting the agent types that will exist in the system is perhaps the most important decision that is made in this phase. Ideally, each agent type should be cohesive and coupling between agents should be low.

In Prometheus, an agent type is formed by combining one or more functionalities. Different groupings of functionalities give alternative designs that are evaluated based on the cohesiveness of the agent types and the degree of coupling between agents. For example, when considering a grouping of functionalities, if two functionalities are clearly related, then it might make sense to group them together in the same agent type. Conversely, if two functionalities are clearly not related, then they should perhaps not be grouped in the same agent type.
Similarly, if two functionalities need the same data, then they should perhaps be grouped together.

A useful tool for suggesting groupings of functionalities is the data coupling diagram. This depicts each functionality and each data repository showing where functionalities read and write data. An arrow from functionality to data denotes reading. It is often fairly easy to extract some constraints on the design by visually examining a data coupling diagram.

Figure 3 shows a data coupling diagram for the functionalities in the meeting scheduler. In this case, the student chose to create four agent types:

- **UserInterface**: combining the functionalities of User Interaction and User Notify. This grouping made sense because both functionalities are concerned with interaction with the user.
- **ContactTracker**: based on the Contact Manager functionality and including the Contact Information database. Although the Contact Notify functionality could have been included in this agent type, it interacts more with the functionalities concerned with scheduling meetings and so was grouped with them in the Meetings agent.
- **Meetings**: combining the functionalities of Meeting Scheduler, Meeting Manager, Negotiator, and Contact Notify, and including the Meetings Database. These functionalities are both related and interact with each other.

Figure 3. Data coupling diagram
• **UserManager**: combining the *User Monitor* and *User Information Manager* functionalities; and including the *User Habits* and *User Preferences* databases.

The process of deriving agent types by grouping functionalities, with the aid of a data coupling diagram, can often suggest possible changes to the functionalities. For example, suppose that the design includes two functionalities that are unrelated, and we would like to put them in two different agent types. However, the two functionalities both read a particular data source; for instance, in the grouping given, the *User Interaction* functionality within the *UserInterface* agent reads the *Meetings Database*. We could change one of the functionalities so that rather than read the data source directly, it sends a message to another agent requesting the information.

The result of this process is a number of possible designs, each design consisting of a grouping of functionalities into agent types. We now need to select a design. One technique that is useful in comparing the coupling of different alternatives is the use of agent acquaintance diagrams. An agent acquaintance diagram shows the agent types and the communication pathways between them. Agent acquaintance diagrams provide a convenient visualization of the coupling between the agent types—the higher the link density, the higher the coupling.

For example, the agent acquaintance diagram in Figure 4 shows the agent types and the communication pathways between them. This is believed to be a reasonable degree of coupling; the *UserInterface* agent needs to communicate with all of the other agent types, but most of the other agent types don’t communicate.

Once agent types have been decided upon, they are documented using an agent descriptor. In addition to capturing the interface of the agent, what goals it achieves, what functionalities were combined to form it, and with what protocols the agent is involved, the descriptor prompts the designer to think about lifecycle issues: When are instances of this agent type created? When are they destroyed? What needs to be done when agents are created/destroyed?

*Figure 4. Agent acquaintance diagram*
Agent Interactions

The next step in the architectural design is to work on the interactions between agents. These are developed using interaction diagrams and interaction protocols. Specifically, the notations used are a simplified variant of UML sequence diagrams for interaction diagrams and the revised version of Agent-UML (AUML) (Huget et al., 2003) for interaction protocols.

Interaction diagrams are derived from use case scenarios using a fairly mechanical process (although not completely mechanical). In essence, if step \( N \) is performed by an agent \( A \), and this is followed by step \( N+1 \) performed by a different agent \( B \), then a message needs to be sent from \( A \) to \( B \). For example, in the use case scenario given earlier, step 1 is a percept that is received by the User Interaction functionality that has been grouped into the UserInterface agent type, and step 2 is a goal of the Meeting Scheduler functionality that has been grouped into the Meetings agent type. Therefore, assuming a single agent of each of the two types, in between step 1 and step 2, there is a message from the UserInterface agent to the Meetings agent.

Like use case scenarios, interaction diagrams show example interactions rather than all possible interactions. In order to define plans that will handle all necessary interactions, we use interaction protocols to capture all possible sequences of messages.

Often an interaction protocol will combine a number of interaction diagrams. For example, if there are three interaction diagrams corresponding to different cases of scheduling a meeting, then there will be an interaction protocol that covers all cases and that subsumes the interaction diagrams. When looking at the use case scenarios, we also consider the documented variations of these scenarios.

Another useful technique for developing interaction protocols is to consider each point in the interaction sequence and ask “What else could happen here?” If the interaction diagram shows an example sequence where a request for possible meeting times is replied to with a number of possible times, then an alternative possibility is that there won’t be any meeting times available.

The interaction protocols are accompanied with descriptors for both the protocols and for the messages. These descriptors capture additional information such as the information carried by a message.

Overall System Structure

Finally, the overall architecture of the system is captured using a system overview diagram. The system overview diagram is one of the most important design artifacts produced in Prometheus and is often a good starting point when
trying to understand the structure of a system. The system overview diagram shows agents, percepts, actions, messages, and external data as nodes (see Figure 5). Directed arrows between nodes indicate messages being sent and received by agents, actions being performed by agents, percepts being received by agents, and data being read and written by agents. The system overview diagram collects information from the various descriptors (especially the agent descriptors) and presents it in a more easily understood form.

Figure 5 shows the overall design of the system. It depicts the four agent types identified and also shows the messages between them, percepts received by the UserInterface agent, and data read and written.

**Detailed Design**

Detailed design consists of:

- Developing the internals of agents, in terms of capabilities (and, in some cases directly in terms of events, plans, and data). This is done using agent overview diagrams and capability descriptors.
• Developing process diagrams from interaction protocols.
• Developing the details of capabilities in terms of other capabilities as well as in terms of events, plans, and data. This is done using capability overview diagrams and various descriptors. A key focus is developing plan sets to achieve goals and ensuring appropriate coverage.

Capabilities (Busetta, Rönnquist, Hodgson, & Lucas, 2000; Padgham & Lambrix, 2005) are a structuring mechanism akin to modules. A capability can contain plans, data, and events. It can also contain other capabilities allowing for a hierarchical structure. In identifying the capabilities that each agent type contains, one usually starts by considering a capability for each functionality that was grouped in the agent type. This initial detailed design is then refined by merging capabilities that are similar and small, splitting capabilities that are too large, and adding capabilities that correspond to common “library” code.

The structure of each agent is depicted by an agent overview diagram. This is similar to the system overview diagram except that it does not contain agent nodes and does not (usually*) contain protocol nodes. However, the agent overview diagram does (usually) contain capability nodes and (sometimes) plan nodes.

Figure 6. Example of an agent overview diagram: Meeting agent

*Copyright © 2005, Idea Group Inc. Copying or distributing in print or electronic forms without written permission of Idea Group Inc. is prohibited.
Developing the internal structure of an agent (or capability) is done using a refinement process that begins with the *interface* of the agent: the data that it reads and writes, the incoming and outgoing messages, the percepts it receives, and actions it performs. Then, for each part of the agent’s interface, we consider to what capability or plan it connects. In addition to connecting the interface of the agent to internal plans and/or capabilities, additional internal messages and data are introduced as needed.

For example, in developing the internals of the *Meeting* agent (see Figure 6) we begin by considering its interface: the messages that it sends and receives, and the data that it reads and writes. Some of the messages are concerned with adding, deleting, or searching for meetings. Based on the original functionalities, we introduce the capability *ManageMeetings* that receives these messages and that reads and writes the *MeetingsDB*. Other messages concern proposing a meeting time based on the user preferences. These are handled by the *CreateMeetingSchedule* capability. The two capabilities communicate using internal messages (*RequestScheduleMeeting* and *MeetingSchedule*), which are now introduced.

During the architectural design phase, the system’s dynamics were described using interaction protocols. These are global in that they depict the interaction between the agents from a “bird’s-eye view.” In the detailed design phase, we develop process diagrams based on the interaction protocols. The process diagrams depict *local* views for each agent. Typically, each interaction protocol will have multiple process diagrams corresponding to the viewpoints of different agents. The notation that we use for process diagrams is an extension of UML activity diagrams (OMG, 2001); for more details see Padgham and Winikoff (2004, Chapter 8).

The design of each agent is, usually, in terms of capabilities. These capabilities are then refined in turn. Eventually the design of how each agent achieves its goals is expressed in terms of plans, events, and data. At this point, the design process needs to make certain assumptions about the implementation platform. Specifically, we assume that the agents are implemented using a platform that supports plans that are triggered by goals. Such platforms include PRS (Georgeff & Lansky, 1986), JAM (Huber, 1999), JACK (Busetta et al., 1998), Jadex³, and Jason⁴.

The final part of detailed design develops data, events, and plans. This is similar to developing the internals of agents in terms of capabilities but with a few additional considerations.

For events, we identify the information that is carried by the event. For example, a request to schedule a meeting would include information on who should attend the meeting, the priority of the meeting and perhaps what date is desired. For each plan we identify its trigger (what event causes the plan to run) and the plan’s
context condition and then develop the plan’s body. The plan’s context condition specifies the situations in which it is appropriate to use the plan. The plan’s body describes how the plan operates and is often a sequence of steps which include subgoals.

A given event can trigger multiple plans, and an important consideration relates to this: for a given event, will there always be at least one plan that handles it and that will be applicable? If so, then we say that the event is covered. This is important because an uncovered event will fail to be handled. Where there is a (typically single) plan with a context condition that is always true, then the event is trivially covered. In other situations, coverage can be checked by considering the context conditions of the set of plans that handles the event type in question. A related notion is that of overlap: if a given event might have more than one applicable plan, then there is overlap.

Plans are described using a descriptor, which also prompts the designer to consider whether the plan can fail and, if so, whether the failure will be detected, as well as what should be done to clean up upon failure.

**Tool Support**

We have attempted to stress the iterative nature of design, both across the phases of Prometheus and within phases. One consequence of the iterative nature is that the design is often modified. As the design becomes larger, it becomes more difficult to ensure that the consequences of each change are propagated and that the design remains consistent\(^1\).

Perhaps the simplest example of introduced inconsistency is renaming an entity and failing to rename it everywhere it is mentioned. Other forms of inconsistency that can be easily introduced when making changes to a design include adding a message to an agent in the system overview diagram but failing to ensure that the message appears in the agent overview diagram of that agent type, or adding an incoming message to an agent but forgetting to add a plan to handle the message.

Our experience, and the experience of students who used the Prometheus methodology in its earlier days, was that developing designs by hand (using only standard tools such as word processors and generic diagram editors) is quite error prone and that tool support is invaluable. As a result, the *Prometheus Design Tool* (PDT) was developed.

The Prometheus Design Tool (see Figure 7) allows users to create and modify Prometheus designs. It ensures that certain inconsistencies cannot be introduced and provides cross checking that detects other forms of inconsistency. The tool
can also export individual design diagrams and generate a report that contains the complete design. For more details on tool support for Prometheus, see Padgham and Winikoff (2002). The Prometheus Design Tool is freely available and further functionality is under development.

Another tool that supports the Prometheus methodology is the JACK Development Environment (JDE) (see Figure 8), which provides a design tool that allows Prometheus-style overview diagrams to be drawn. The JDE can then generate skeleton code from these diagrams and ensures that changes made to the code are reflected in the design diagrams and vice versa. This facility has proven quite useful.

Finally, Sudeikat et al. (2004) mention that they have developed a tool that takes a PDT design and generates from it a Jadex agent definition file. This tool is notable in that it was developed completely independently from the Prometheus and PDT developers.

Figure 7. The Prometheus design tool (PDT)
Strengths and Weaknesses

As discussed in the introduction, Prometheus aims to be a practical methodology that can be used by undergraduate students and industry practitioners. The (anecdotal) evidence supports our belief that we have succeeded in this regard, and this is a key strength of Prometheus.

Prometheus has been developed over a number of years. During this time, it has been taught to undergraduate students, used by students doing summer projects, and taught at industry workshops. These activities yielded feedback that has been valuable in refining and improving the methodology.

On the industrial side, a prototype weather-alerting system developed by Agent Oriented Software for the Australian Bureau of Meteorology (Mathieson, Dance, Padgham, Gorman, & Winikoff, 2004) used Prometheus overview diagrams to capture and document the design. These diagrams were produced using the JACK Development Environment (JDE).

In order to obtain a slightly more reliable assessment of Prometheus, we have on two occasions set summer projects where a student was given a written description of the Prometheus methodology, intentionally limited support from staff members, and was instructed to design and build an agent system. During the 2001/2002 Christmas vacation, a second-year student produced a design and
implementation for a Holonic manufacturing system. During the 2002/2003 Christmas vacation, a different student produced a detailed design for a tourism application. These experiences contrast with pre-Prometheus experiences where graduate students struggled to design agent systems and required considerable support from staff.

In our teaching, we have found that the Prometheus methodology has made a significant difference. Before the methodology had been developed, graduate students struggled to design and implement reasonable agent systems, whereas now we are successfully teaching Prometheus and JACK in a one-semester course. This course, which is taught to undergraduate students, sees most of them successfully design and build reasonable agent systems within the period of a semester.

These experiences provide evidence that Prometheus is useful as well as usable by its intended target audience. Another strength of the Prometheus methodology is the possibility and existence of tool support.

However, Prometheus is not without weaknesses. Its support for the social aspect of agents is currently focused on the lowest common denominator: messages and protocols. Extending the methodology to support more specific types of agent interaction and relationships, such as teams of agents (Cohen & Levesque, 1991) and open societies of agents, is one of the areas on which we are currently focusing. The area of software methodologies for designing open agent systems is quite new and exciting. Existing work that we intend to build on includes Juan, Pearce, and Sterling (2002); Huget (2002); and Mathieu, Routier, and Secq (2003).

Prometheus also does not deal at all with mobile agents. This has not been a priority as we do not see mobility as central for intelligent agent systems. However, if a developer is designing a system where mobility is a significant aspect, then Prometheus is likely to be inadequate as a design methodology.

Prometheus covers the system specification, high-level design, and detailed design activities with some discussion of implementation issues. There has also been some work on using design models to help in debugging agent systems (Poutakidis, Padgham, & Winikoff, 2002, 2003; Padgham, Winikoff, & Poutakidis, 2005). However, the support for implementation, testing, and debugging is limited at the moment. Also, Prometheus currently has less focus on early requirements and analysis of business processes than a methodology such as Tropos. These are, however, all areas in which Prometheus is undergoing development and can be expected to evolve.

Finally, Prometheus is not based on UML. This can be regarded as being either a strength or a weakness. From the point of view of developing a methodology that is well-suited to designing agents, we feel that not starting with a perspective that is very object-centric has been a good decision. On the other hand, UML is
clearly the standard notation, with which most developers are familiar. We have tried to build on aspects of UML and object-oriented design where appropriate, although we note that there are other approaches that do this to a greater extent (e.g., Papasimeon & Heinze, 2001; Wagner, 2002).

Conclusions

We have briefly presented the Prometheus methodology for designing intelligent software agents and agent systems. The methodology provides detailed guidance in terms of processes as well as notations. It is not intended to be prescriptive, but is rather an approach that has evolved out of experience and that the authors expect to be further adapted, refined, and developed to suit the needs of agent software developers.

Recent years have seen a substantial growth of activity in the area of software engineering methodologies suited to an agent programming paradigm. As these mature and develop and are increasingly used beyond the immediate sphere of the developers, we expect them to bear fruit in terms of increased use of agent technology and more widespread familiarity with building of agent systems.

Acknowledgments

We would like to acknowledge the support of the Australian Research Council (ARC) under Grant CO0106934 and its continuation, Grant LP0453486. We would also like to thank John Sietsma, James Harland, John Thangarajah, and David Poutakidis of RMIT University; Ralph Rönquist, Andrew Lucas, Andrew Hodgson, Paul Maisano, and Jamie Curmi of Agent Oriented Software; and the many students and workshop participants who have provided comments, examples, and feedback. The Prometheus Design Tool was initially developed by Anna Edberg and Christian Andersson. Further development has been by Claire Hennekam and Jason Khallouf.

References


Endnotes

1 Named after the Titan who was, amongst other things, the protector of mankind. Prometheus, according to Greek mythology, stole fire from Zeus and gave it as a gift to humanity; an act for which he was punished. (http://www.greekmythology.com)

2 JACK Intelligent Agents™ is a commercial agent development platform developed by Agent Oriented Software.

3 In fact, there are also other reasons for considering goal-oriented requirements (van Lamsweerde, 2001).

4 The terminology used is not consistent with UML. Since Prometheus uses scenario instances (“scenarios” in UML) and does not have anything corresponding to UML’s Use Case Diagram, there is no need to distinguish between two concepts.

5 The diagram is slightly modified from the one that was prepared by the student.

6 These are renamed in the final version of the design to UserHabitsDB and UserPreferencesDB, respectively.

7 Other notations that could be used for this purpose include the original version of AUML and Petri nets.

8 Although it may make sense to allow protocol nodes in agent overview diagrams, the current version of the Prometheus Design Tool does not support this.

9 http://vsis-www.informatik.uni-hamburg.de/projects/jadex

10 http://jason.sourceforge.net

11 In the general sense, not in the formal sense of a logical theory being consistent.


13 Summer projects are done by undergraduate students over eight weeks, full time during the summer non-teaching period.

14 It is important to note that UML is only a notation, not a methodology.

15 *Simplifying the Development of Agent-Oriented Systems*, ARC SPIRT Grant, 2001-2003.