Prometheus: A Pragmatic Methodology for Engineering Intelligent Agents

Lin Padgham Michael Winikoff
RMIT University
{linpa,winikoff}@cs.rmit.edu.au

Abstract

Agents are a powerful technology with many significant applications. A key issue in getting the technology into mainstream software development is the development of appropriate methodologies *for engineering agent-oriented software. This paper presents the* Prometheus *methodology, which has been developed over several years in collaboration with Agent Oriented Software. The methodology has been taught at industry workshops and university courses. It has proven effective in assisting developers to design, document, and build agent systems. Prometheus is a detailed and complete (start to end) methodology for developing intelligent agents which has evolved out of industrial and pedagogical experience, and which is supported by tools. This paper gives a brief overview of the design process as a whole and then discusses envisaged tool support as well as the implemented tool prototypes we have been using.*

1 Position Statement

Agents are a powerful technology with many significant applications, both demonstrated and potential [8, 7]. A key issue in getting the technology into the mainstream of software development is the development of appropriate *methodologies* for engineering agent-oriented software. This paper motivates and presents the *Prometheus*[1] methodology for developing intelligent agent systems.

We consider a methodology to encompass (i) a set of concepts used; (ii) notations for modelling aspects of the software (requirements, designs, implementation); and (iii) a process that is followed in order to produce the software.

The Prometheus methodology has been developed over the last several years in collaboration with Agent Oriented Software[2] (AOS). Our goal in developing Prometheus was to have a process with associated deliverables which can be taught to industry practitioners and undergraduate students who do not have a background in agents and which they can use to develop intelligent agent systems. To this end Prometheus is *detailed* and *complete* in the sense of covering all the stages of software development as applied to agent systems.

We believe that particular strengths of the Prometheus methodology include:

- Provision of "start-to-end" support (from specification to detailed design, implementation and testing), including a *detailed process*, along with design artifacts constructed and steps for deriving artifacts.
- Hierarchical structuring mechanisms which allow design to be performed at multiple levels of abstraction. Such mechanisms are crucial to the practicality of the methodology for large applications.
- The structured nature of the design artifacts facilitates development of tool support, especially automated cross checking of design artifacts, and automated provision of skeleton artifacts at certain stages. Partial tool support already exists.
- Support for detailed design of the internals of *intelligent* agents. This necessarily makes increased assumptions about the implementation platform. The detailed design assumes a plan based system where agents react to events, based on their beliefs about the situation. It is particularly well suited to BDI type systems.
- The fact that the methodology has evolved out of practical industrial and pedagogical experience, and thus addresses issues in designing agent systems that have been experienced by both industrial practitioners and undergraduate students.

Prometheus also uses an iterative process over the phases described in this paper, rather than a linear "waterfall"

[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. (*http://www.greekmythology.com/*)

[2] *http://www.agent-software.com*

model, although the amount of work on the earlier phases decreases as the design progresses, just as the amount of work on later phases increases.

Designs for large systems are almost always developed incrementally with many revisions. When revising any artefact, be it documentation, code, or design, it is easy to introduce inconsistencies and minor errors. We have found the prototype tool we have developed extremely useful for checking and maintaining design consistency across varying levels of detail. The automatic generation of skeleton code from design artifacts in the tool prototype is also extremely useful, and has encouraged students to do design prior to coding.

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 they 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. Over the last summer vacation a second year student was given a description of the methodology and a description of an agent application (in the area of Holonic Manufacturing). With only (intentionally) limited support, the student was able to design and implement an agent system to perform Holonic Manufacturing using a simulator of a manufacturing cell. Although this is only anecdotal evidence that the methodology is helpful, it has had a very significant effect on our ability to teach students to program agent systems.

The methodology is also taught to industry software developers who are starting to use the JACK intelligent agents development environment [2], and has been successful in introducing them to methods to assist them in design of agent applications.

In this paper we give a brief overview of the design process as a whole, focussing in somewhat more detail on the *Architectural Design*, in order to give some flavour of the methodology. We discuss envisaged tool support as well as the implemented tool prototypes we have been using. There is not room in a paper of this length for much comparison with other methodologies, but we mention this briefly in section 4, in particular discussing the relationship to UML.

2 The Prometheus Methodology

The *Prometheus* methodology [15] includes three design phases, where artifacts are produced which are used in both production of skeleton code for implementation, and for debugging and testing. The *system specification phase* focuses on identifying the basic functionalities of the system, along with inputs (percepts), outputs (actions) and any important shared data sources. The *architectural design phase* uses the outputs from the previous phase to determine which agents the system will contain and how they will interact. The *detailed design phase* looks at the internals of each agent and how it will accomplish its tasks within the overall system. Figure 1 indicates the main design artifacts arising from each of these phases as well as some of the intermediary items and relationships between items.

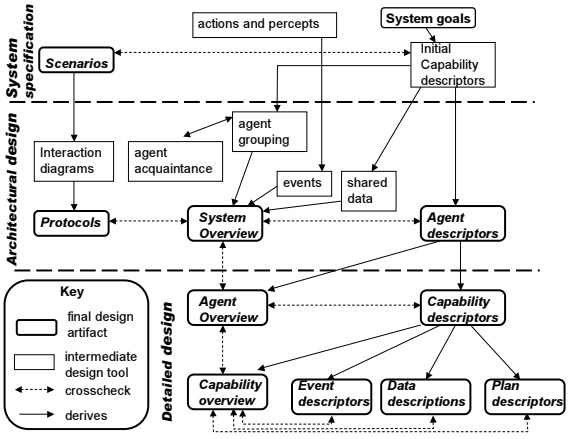


Figure 1. Phases, artifacts and relationships in the design process

Tools from OO analysis and design are adapted in several places as appropriate. For example, Prometheus' scenarios are a variant of the scenario part of UML's use cases. while our interaction diagrams are essentially UML sequence diagrams.

We describe the methodology using a running example: an online bookstore that assists customers in finding and choosing books, manages deliveries, stock, customer and supplier information, and does selective advertising based on interests. Space limitations prevent us from describing in detail all aspects of the Prometheus methodology. However, we hope that some understanding of its structure, phases, artifacts and activities can be gained. We are not able to cover testing and debugging in this paper, but some of this is covered in [17].

2.1 System specification

Before starting to design or build any software system there is discussion with clients, managers or other stakeholders regarding the general purpose of the system. We do not cover the preliminary part of this discussion, but start from the point at which it is possible to start specifying some of the details arising out of such discussions.

Agent systems are typically situated in a changing and dynamic environment, which can be affected, though not totally controlled by the agent system. One of the earliest questions which must be answered is how the agent system is going to interact with this environment. We call incoming information from the environment "percepts", and the mechanisms for affecting the environment "actions".

As discussed in [21] it is important to distinguish between percepts and events: an event is a significant occurrence for the agent system, whereas a percept is raw data available to the agent system. Often percepts can require processing in order to identify events that an agent can react to. For example, if a soccer playing robot's camera shows a ball where it is expected to be then this percept is not significant. However, if the ball is seen where it is not expected then this *is* significant.

Actions may also be complex, requiring significant design and development outside the realm of the reasoning system. This is especially true when manipulation of physical effectors is involved. We shall not address percept processing and actions any further in this paper, although identification and specification of these are an important part of the initial and ongoing design process.

The online bookstore's percepts and events include customers visiting the website, selecting items, placing orders using forms, and receiving email from customers, delivery services and book suppliers. Actions include bank transactions, sending email, and placing delivery orders.

In parallel with identifying or specifying (which of these will depend on the situation) the percepts and actions the developer must start to describe what it is the agent system should do in a broader sense - the goals and functionalities[3] of the system. For example, in order to define the book store we may have goals such as *maintain stock levels* and *provide competitive prices* with related functionalities such as *"stock ordering"*, *"stock monitoring"*, *"competitor monitoring"* and *"pricing review"*. These goals and functionalities start to give an understanding of the system.

It is important in defining functionalities that they be kept as narrow as possible, dealing with a single aspect or sub-goal of the system. If functionalities are too broad they are likely to be less adequately specified leading to potential misunderstanding.

In defining a functionality it is important to also define information required and produced by it. The functionality descriptor contains a name, a short natural language description, a list of actions, a list of relevant percepts, data used and produced and a brief description of interactions with other functionalities.[4] For example, the following describes the *welcoming* functionality in the online bookstore.

NAME: Welcoming
Description: Welcomes a new visitor to the world wide web site (with personalised information if possible).
Percepts/events/messages: CustomerArrived (message), CustomerInformation (message)
Messages sent: CustomerInformationRequest (message), CustomisedWWWPage (message),
Actions: DisplayCustomisedWWWPage
Data used: CustomerDB, CustomerOrders
Interactions: CustomerManager (via CustomerInformationRequest, CustomerInformation) OnlineInteraction (via CustomisedWWWPage, CustomerArrived)

Each functionality should be linked to some system goal, while each goal should result in one or more functionality, although there may not be a one to one mapping.

While functionalities focus on particular aspects of the system, *scenarios* give a more holistic view of system processing. The basic idea is borrowed from object oriented design. However, the scenarios are given slightly more structure.

The central part of a scenario in Prometheus is the sequence of steps describing an example of the system in operation. Each step in the scenario is one of the following:

- incoming event/percept (→ receiving functionality)
- message (sender → receiver)
- activity[5] (functionality)
- action (functionality)

These steps can optionally have data read and data written noted.

Our scenario templates contain an identification number, a brief natural language overview, an optional field called context which indicates the situations in which this scenario could occur – i.e. its preconditions, the sequence of steps, a summary of all the information used in the various steps, and a list of small variations. Because a scenario captures only one particular sequence of steps it can be useful to indicate small variations with a brief description. Any major variations should be a separate scenario.

Scenario: Book Order
Overview: The user orders a book. Delivery options are explored and then confirmed (with an OrderRequest). The books are shipped, stock updated, and the user notified.
Context: Assumes the book is in stock.
Steps:

1. EVENT BookOrder (→ Online Interaction)

[3] A number of methodologies call these "roles". We prefer to avoid overloading the term since it has a similar, but non-identical, meaning in the context of teams of agents.

[4] Any of these except name and description may be empty.

[5] We use the term *activity* to denote anything done within the functionality - typically some kind of computation

2. DeliveryOptionQuery (Online Interaction → Transport Information)
3. DeliveryOptions (Transport Information → Online Interaction) Data read: Transport DB
4. Obtain preferred delivery option (Online Interaction)
5. MakePayment (Online Interaction → Sales Transaction)
6. ACTION BankTransaction (Sales Transaction)
7. PlaceOrder (Sales Transaction → Order Handling)
8. Register order (Order Handling) Writes data: CustomerOrders
9. ACTION EmailCourierCompany (Order Handling)
10. DecreaseStock (Order Handling → Stock Manager)
Variations: steps 9 (email courier) and 10 (decrease stock) replaced with notification of delay (Order Handling to Customer Contact) and then placing an order for more stock (Order Handling to Stock Manager).

2.2 Architectural design

A major decision to be made during architectural design is which agent types the system should have. We assign functionalities to agents by analysing the artifacts of the previous phase to suggest possible assignments of functionalities to agents. These are then evaluated according to the traditional software engineering criteria of cohesion and coupling.

Once we have decided upon the agents in the system we identify which agents react to which percepts or environmental events, as well as which agents perform particular actions on the external environment. In addition we specify the messages that are sent between agents, and determine the major data repositories. These items form the overall design of the system and are depicted in the *system overview diagram*. The system overview diagram is perhaps the single most important product of the design process. It ties together agents, data, external input and output, and shows the communication between agents.

The system overview diagram shows the pathways of communication - which agents talk to which other agents - but not the *timing* of communication – which messages are followed by which other messages. An indication of the timing of communication is captured initially in the scenarios which describe processing. This can then be described in the form of agent interaction diagrams which show message passing between agents. Like scenarios, these depict one possible sequence of messages between agents. In order to describe all possible interactions we develop interaction protocols, depicted using the Agent UML (AUML) notation [14].

The process of identifying agents by grouping functionalities involves analysing the reasons for and against groupings of particular functionalities. If functionalities use the same data it is an indication for grouping them, as is significant interaction between them. Reasons against groupings may be clearly unrelated functionality or existence on different hardware platforms. More generally, we seek to have agents which have strong internal cohesion and loose coupling to other agents.

The main reasons for combining functionalities into a single agent are:

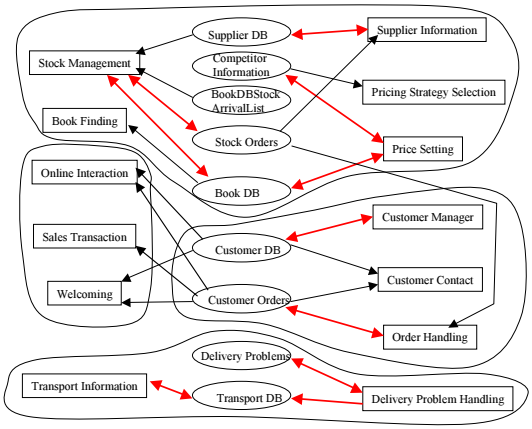
- The functionalities seem related - it "makes sense" to group them. For example, Competitor Monitoring and Pricing Review functionalities in the electronic bookstore example.
- The functionalities require a lot of the same information. If grouped into a single agent this can then be represented in internal agent data structures. If separate agents are used the information must be passed via messages unless a shared data store is used which is not usually a good design decision.

Reasons for *not* grouping functionalities include:

- The functionalities are clearly unrelated
- The functionalities exist on different hardware platforms
- Security and privacy - if data associated with one functionality should not be available to another functionality.
- Modifiability - if a functionality will change, or will be modified by different people.

One technique that we use to systematically examine the properties which lead to coupling is the *data coupling diagram*. A data coupling diagram 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. An example data coupling diagram[6] for the bookstore showing one possible grouping of functionalities into agents is shown below.

[6]This particular diagram is actually reached after some analysis and refinement from the diagram first produced.



The diagram can be checked to ensure that all data is produced somewhere, unless it has been determined that it is provided externally on system start-up and is static. Also, the existence of data that is produced but not used usually indicates an omission elsewhere in the design.

The diagram can be assessed visually for groupings which are linked by their data use. Each such grouping must also be assessed for cohesion and with respect to whether there is some reason to keep the functionalities separated. It is one mechanism to assist in analysis of how functionalities may be combined.

To evaluate a potential grouping for coupling we use an agent acquaintance diagram. This diagram simply links each agent with each other agent with which it interacts. A design with fewer linkages is less highly coupled and therefore preferable.

A simple heuristic for assessing cohesion is whether an agent has a simple descriptive name which encompasses all the functionalities without any conjunctions ("and"). For example, the SalesAssistant agent combines the functionalities of OnlineInteraction, SalesTransaction and Welcoming; yet it has a simple descriptive name. This rule of thumb is obviously not universally applicable, however, we have found it to be a useful (and quick) way of assessing cohesion in proposed designs.

Once a decision has been made regarding agent types, high level information can be recorded in agent descriptors, similar to functionality descriptors. Questions which need to be resolved about agents at this stage include:

- How many agents of this type will there be (singleton, a set number, or multiple based on dynamics of the system, e.g. one sales assistant agent per customer)?
- What is the lifetime of the agent? If they are created or destroyed during system operation (other than at start-up and shut-down), what triggers this?
- Agent initialisation - what needs to be done?
- Agent demise - what needs to be done?
- What data does this agent need to keep track of?
- What events will this agent react to?

In addition we extract from the functionality descriptors, information about goals, percepts and events, actions, data and messages and add these to the agent descriptor.

For example consider the following agent descriptor from our electronic bookstore example:

Name: Sales Assistant agent
Description: greets customer, follows through site, assists with finding books
Cardinality: one/customer.
Lifetime: Instantiated on customer arrival at site. Demise when customer logs out or after inactivity period.
Initialisation: Obtains cookie. Reads Customer DB.
Demise: Closes open DB connections.
Functionalities included: Online Interaction, Sales Transaction, Welcomer, Book Finder.
Uses data: Customer DB, Customer Orders, Book DB.
Produces data: Customer preferences, orders, queries
Goals: Welcome customer; Update customer details; Respond to queries; Facilitate purchases;
Events responded to: new arrival; customer query; customer purchase; credit check response customer response;
Actions: Display information to customer (greetings, book info, info requests, Display customised WWW page, RequestCreditCheck messages
Interacts with: Warehouse Manager (book request protocol), Delivery Manager (order protocol, order query protocol), Customer Manager (customer information query protocol, customer information update protocol)

Having identified the agent types we proceed to capture the top level structure of the system using a system overview diagram. We identify *events* (i.e. significant occurrences) that will be generated as a result of information from the environment (the percepts), and will be noticed by the agents, and *shared data objects*.

A good design will minimise shared data objects, although there may be situations where this makes sense. If multiple agents will be writing to shared data objects this will require significant additional care for synchronisation (as agents operate concurrently with each other). Often shared data objects can be avoided by having the data source managed by a single agent, with information provided to other agents as needed. Sometimes each agent can have its own version of the information, without there being any need for a single centralised data object. Data objects could be specified using traditional object oriented techniques or database design techniques as appropriate.

The *system overview diagram* ties together the agents, events and shared data objects. It is arguably the single most important artifact of the entire design process, although of course it cannot really be understood fully in isolation. By

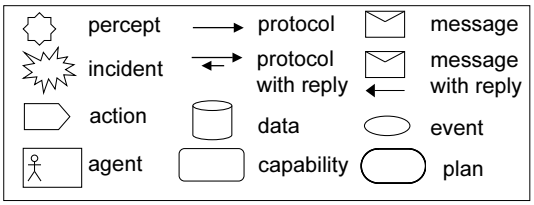
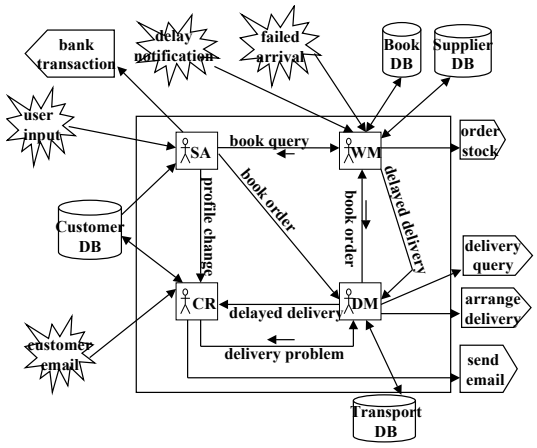


Figure 2. Notation for System Overview and other diagrams

viewing a system overview diagram we obtain a general understanding of how the system as a whole will function, including interactions between agents. Agent descriptors and protocols provide additional detail needed to understand the high level functioning of the system. Similar diagrams are used at lower levels of the system as it is broken down in a hierarchical fashion. A (partial) system overview diagram for the electronic bookstore is shown below; the notation used is summarised in figure 2.



The final aspect of the architectural design is to specify fully the *interaction* between agents. Interaction diagrams are used as an initial tool for doing this, while fully specified interaction protocols are the final design artifact.

Interaction diagrams are borrowed directly from object oriented design (UML's sequence diagrams), showing interaction between agents rather than objects. An advantage of the highly structured scenarios developed in the specification phase is that they can be used directly for producing interaction diagrams. Often at this stage there is some modification and refinement of scenarios to produce simpler interaction diagrams.

Interaction diagrams, like scenarios, give only a partial picture of the system's behaviour. In order to have a precisely defined system we progress from interaction diagrams to *interaction protocols*, specified in AUML, which define precisely which interaction sequences are valid within the system. These are generalisations of the interaction diagrams. At each point that a message is sent, the question must be asked as to whether there are other choices that could be made - no message sent or a different message. Similarly the question must be asked for each message, whether it is necessarily sequenced, or whether some parallelism is allowed.

2.3 Detailed design

Detailed design focuses on developing the internal structure of each of the agents and how it will achieve its tasks within the system.

Detailed design is, by its nature, specific (at least to some degree) on the model of agents used; specifically on the internals of agents. In order to develop a detailed design process and associated models we need to select a particular agent model. We have chosen to focus on *plan-based* agents that use a library of user-defined plans. Thus, the results of the detailed design phase include a collection of event-triggered plans. These can be mapped directly to the implementation constructs that are provided by certain plan-based implementations, including the various implementations of Belief, Desire, Intention (BDI) systems (e.g. PRS, dMARS, JAM, or JACK). Thus the detailed design phase is particularly well suited for such systems.

Although our detailed design caters well for BDI agents, it is not limited to them. The principles are easily adapted to the specifics of whichever development platform has been chosen, as long as it is within the broad general category of agents which use plans and react to events. The earlier phases of the methodology are *not* specific to a given agent model.

The focus of the detailed design phase is on defining capabilities (modules within the agent), internal events, plans and detailed data structures. A progressive refinement process is used which begins by describing agents' internals in terms of capabilities. The internal structure of each capability is then described, optionally using or introducing further capabilities. These are refined in turn until all capabilities have been defined. At the bottom level capabilities are defined in terms of plans, events, and data.

The functionalities from the specification phase provide a good initial set of capabilities, which can be further refined if desired. Sometimes there is also functionality akin to "library routines" which is required in multiple places - either within multiple agents, or within multiple capabilities within a single agent. Such functionality should also be extracted into a capability which can then be included into other capabilities or agents as required.

Capabilities are allowed to be nested within other capabilities and thus this model allows for arbitrarily many lay-

ers within the detailed design, in order to achieve an understandable complexity at each level.

Each capability should be described by a capability descriptor which contains information about the external interface to the capability - which events are **inputs** and which events are **produced** (as inputs to other capabilities). It also contains a natural language **description** of the functionality, a unique descriptive **name**, information regarding **interactions** with other capabilities, or **inclusions** of other capabilities, and a reference to data **read** and **written** by the capability. We use structured capability descriptor forms containing the above fields.

The agent overview diagram provides the top level view of the agent internals. It is very similar in style to the system overview diagram, but instead of agents within a system, it shows capabilities within an agent. This diagram shows the top level capabilities of the agent and the event or task flow between these capabilities, as well as data internal to the agent. By reading the relevant capability descriptors, together with the diagram, it is possible to obtain a clear high level view of how the modules within the agent will interact to achieve the overall tasks of the agent as described in the agent descriptor from the architectural design.

A further level of detail is provided by capability diagrams which take a single capability and describe its internals. At the bottom level these will contain plans, with events providing the connections between plans, just as they do between capabilities and between agents. At intermediate levels they may contain nested capabilities or a mixture of capabilities and plans. These diagrams are similar in style to the system overview and agent overview diagram, although plans are constrained to have a single incoming (triggering) event.

The final design artifacts required are the individual plan, event and data descriptors. These descriptions provide the details necessary to move into implementation. Exactly what are the appropriate details for these descriptors will depend on aspects of the implementation platform. For example if the context in which a plan type is to be used is split into two separate checks within the system being used (as is the case in JACK) then it is appropriate to specify these separately in the descriptor. Fields regarding what information an event carries assumes that events are composite objects able to carry information, and so on.

The plan descriptors we use provide an **identifier**, the **triggering event type**, **events, messages and actions**, the **plan steps** as well as a short natural language **description**, a **context** specification indicating when this plan should be used and a list of data **read** and **written**.

Event descriptors are used to fully specify all events, including those identified earlier. The event descriptor should identify the **purpose** of the event and any **data** that the event carries. We also indicate for each event its *coverage*: how many plans should be applicable to handle it? This is generally a range with the interesting values being zero, one, or many. Interesting cases include $1-1$ (always a single applicable plan), $1-N$ (there will always be a plan to handle the event and there may be more than one possible plan), $0-1$ (there will never be more than one applicable plan in a given situation, however it is possible for there to not be any applicable plan), and $0-N$. The coverage information is useful for debugging: a common programming error involves incorrect context conditions leading to the wrong plan being used, or to an event not being handled because no plan has is applicable.

Data descriptors should specify the fields and methods of any classes used for data storage within the system. If specialised data structures are provided for maintaining beliefs, these should also be specified.

An additional artifact that needs to be maintained as the design evolves is the **data dictionary**. The data dictionary should be started at the beginning of the project and developed further at each stage. The data dictionary is important in ensuring consistent use of names (for example, what is called "delivery info" in one place in the design should not be called "deliv. information" elsewhere).

3 Tool Support

In this section we describe the tool support for Prometheus that Agent Oriented Software and ourselves have developed. We begin by describing the support for the Prometheus methodology that is currently provided by the Jack Development Environment, including the recently-added design tool. We then describe further support that we have developed in a prototype tool and some ideas for supporting the specification phase that have not yet been implemented.

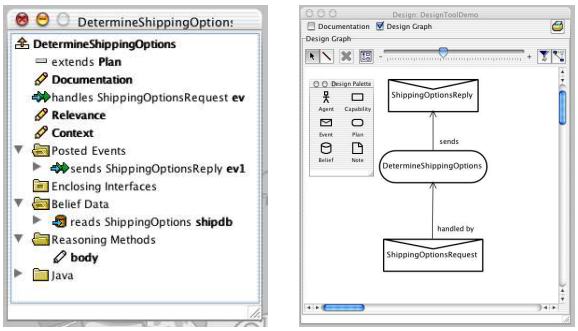
Our experience in using these tools has been positive. In particular, the cross checking provided by the prototype tool has been useful in finding inconsistencies in designs and in helping ensure that a design remains consistent as it is evolved.

3.1 JACK Support for Prometheus

The Jack Development Environment (JDE) provides a graphical user interface that allows the structure[7] of an agent system to be built by drag-and-drop and by filling in forms (see below, left). The JDE supports the Prometheus methodology in that the concepts provided by JACK correspond to the artifacts developed in Prometheus' detailed

[7] Basically, everything except the bodies of plans. However, a graphical editor for plan bodies is under development.

design phase. It is important to realise that the agent structure described in the JDE generates JACK code that can be compiled and run.



A recent addition to the JDE is the *design tool*. This is a graphical canvas that allows overview diagrams to be drawn. However, unlike a drawing program, the diagrams are structured and are linked to the underlying agent system model. For example, the diagram above (right) was constructed by dragging two events and a plan onto the canvas. The links between the nodes are inserted (and labelled) automatically. If the diagram is changed (for example by adding a link between two nodes) then the entities involved are automatically updated. Likewise, if the entities are updated (for example, the plan is modified so that it sends a message rather than posts an event) then the diagram is automatically updated. This automatic insertion (and maintenance) of links ensures consistency between diagrams where possible.

Although consistency is maintained by the JDE in some areas, for example the JDE does not allow type errors to be made such as declaring a plan as handling a posted capability, there are still a number of places where an agent system design may become inconsistent. For example, since nodes are added to diagrams by hand, it is possible to create an agent overview diagram that accidently omits an event. It is possible for an agent or capability to declare that it handles a certain event without the agent/capability actually having a plan to handle the event.

Finally, the JDE and design tool do not (yet) support the earlier phases of the Prometheus methodology. For example, the process of deciding on agent types by combining functionalities, determining system functionality by developing use case scenarios and functionalities, and proceeding from scenarios via interaction diagrams to interaction protocols are not supported by the current tool. Some of these aspects are supported and addressed by the prototype tool described below.

Our experience has been that the design tool allows for the production of overview diagrams and that these are useful in understanding the structure and "big picture" of an agent system.

3.2 Prometheus Design Tool

The *Prometheus Design Tool* (PDT) is a prototype that we have developed. It allows a user to enter and edit a design, in terms of Prometheus concepts; check the design for a range of possible inconsistencies; automatically generate a range of diagrams according to the methodology; and automatically generate a design description (in LaTeX format) that includes descriptors for each design entity, a design dictionary, and the diagrams generated earlier.

PDT provides a graphical user interface (below) that allows a user to enter design elements (capabilities[8], events[9], agents, databases, and plans). Each of these has associated information that matches the Prometheus descriptors. For example, events have a coverage field, a sub-type field (e.g. action, percept, message, event) as well as a name and a description.

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[10], 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 generating design diagrams at different levels (system overview, agent overview, and module overview).

We have found the cross checking to be very useful in maintaining the consistency of an evolving design. We have recently developed the design of the book store application in detail, working part-time over a period of weeks. When the design was first entered into PDT and cross-checked, a number of inconsistencies were detected. These included inconsistent naming of entities, and inconsistencies in sending and receiving messages (for example, the descriptor for functionality A would say it receives message m_1 from functionality B, whereas m_1 was actually being sent by functionality C).

[8] We use this to represent both functionalities and capabilities.
[9] Encompassing percepts, actions, messages, and events.
[10] At the time of writing this has not yet been implemented.

As we have continued to develop the design the cross checking has proved to be valuable in ensuring that changes are made consistently across the design, for example, when we change a functionality by removing a message that it sends, the cross checking tells us where the message is expected and reminds us that we need to either modify the recipients by removing the expected message, or have another functionality post the message in question.

The cross-checking procedure consists of the following steps:

1. Check for undefined references and unused (unreachable) components.
2. Check for correct type and sub-type usage.
3. Check scenarios for consistency.
4. Check interface consistency of composite (see below) components.

We begin with a few definitions before discussing each of these in turn. The first two steps are straightforward and so we focus on the third and fourth.

Plans, events, and databases are *atomic* design components: they do not have children components. Agents and capabilities are *composite*: they have *children* that are either plans or capabilities. For the purposes of checking we also create a single implicit (composite) component of type *system* that has all of the defined agents as children.

Components of type system, agents, capabilities, and plans all have an *interface*: a definition of the events that are incoming and outgoing, as well as a list of databases that are read or written. For the purposes of cross-checking designs reading data is treated identically to an incoming event and writing data is treated identically to an outgoing event. We thus associate with each design component C the following attributes:

- A set of children components *children*(C), defined to be empty for atomic components.
- An interface, defined by two sets: the set of incoming events and read databases I_C, and the set of outgoing events and written databases O_C. The interface set is only defined for plans and for composite components.

Check for undefined references and unused (unreachable) components: It is clearly an error for a component to refer to an undefined component and this is checked in the obvious way. Although not an error, it is unusual for an component to be defined but not used. All components that are reachable from the system component by following children or interfaces are considered reachable. Warnings are issued for any unreachable components.

Check for correct type and sub-type usage: The following simple constraints are checked:

- Actions cannot appear in an incoming interface.
- Percepts cannot appear in an outgoing interface.
- The interface of a component (where defined) must contain only events and databases.
- The children of a (composite) component can only be capabilities or plans.
- The interface of the system component can only include actions (outgoing) or percepts (incoming).

Check scenarios for consistency:[11] As discussed in section 2.1 we use a fairly structured format for the steps in a scenario. This allows scenarios to be checked for internal consistency and for scenarios to be checked for consistency against functionalities (and to partially derive them).

We perform simple consistency checking on scenarios using the notions of *initiative* and *active participants*. Once an event is received, the functionality that has received it has the initiative. It can then perform actions/activities or send messages. Sending a message transfers the initiative to the recipient functionality. A warning is issued if a functionality acts in some way without having the initiative. The notion of active participants is similar to the notion of initiative, except that the active participant is a set that is extended by message sending and does not contract. It is an error for a functionality to act without it being in the set of active participants.

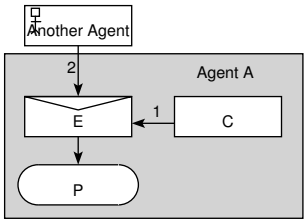
For example, if step 2 of the example scenario in section 2.1 was accidently omitted, then at step 3 Transport Information would be sending a message without being in the active participant set. This would be reported by the cross checking algorithm as an error. If, instead, step 3 was deleted, then at step 4 Online Interaction would be performing an activity without having the initiative, (although it is in the active participant set). This would be reported as a warning.

The interface of a functionality should be a superset of what is implied by its usage in scenarios. For example, where a scenario contains a step where an event is received by a functionality, then that event should be in the incoming interface of the functionality. Likewise, where a message is sent from functionality C_1 to C_2, then the message type should be in C_1's outgoing interface and in C_2's incoming interface. This constraint allows for functionalities to be partly derived from scenarios, and for them to be checked against scenarios.

Check interface consistency of composite components: Consider a composite component, say an agent A. Agent A has children (plans and capabilities) and each child has a defined interface. Any incoming event that appears in a child's interface must come from somewhere. It can either

[11] At the time of writing this has not been implemented.

come from outside the agent, or from another[12] component that is internal to the agent (or both). For example, if A has as children a plan P and a capability C and P is triggered by the incoming event E, then either A has to allow for E to come in (from another agent), or E must be an *outgoing* event of C (or P). This situation is depicted below: one (or both) of the edges labelled 1 and 2 must be present, otherwise there is no source for the event E.



This condition can be defined formally as follows. Let C be a composite component and let I_C and O_C be its interface. We define the *combined incoming interface* of C's children $I_C^+ = \{i \mid ch \in \textit{children}(C) \wedge i \in I_{ch}\}$ and the outgoing interface of C's children $O_C^+ = \{o \mid ch \in \textit{children}(C) \wedge o \in O_{ch}\}$. Then the condition just says that any interface component (event or database) in a child's incoming interface (I_C^+) that is not in some child's outgoing interface (O_C^+) must be in the parent component's incoming interface (I_C). Hence I_C must include anything in I_C^+ that is not in O_C^+, thus $I_C^+ \setminus O_C^+ \subseteq I_C$ and conversely $O_C^+ \setminus I_C^+ \subseteq O_C$.

Another, simpler, condition is that any component that is in C's interface (where C is a composite component – that is, excluding plans) must be realised[13] in a plan contained in (a descendent of) C. This condition can be defined as $I_C \subseteq I_C^+$ and $O_C \subseteq O_C^+$. Putting these two conditions together we have that:

$$(I_C^+ \setminus O_C^+) \subseteq I_C \subseteq I_C^+$$

$$(O_C^+ \setminus I_C^+) \subseteq O_C \subseteq O_C^+$$

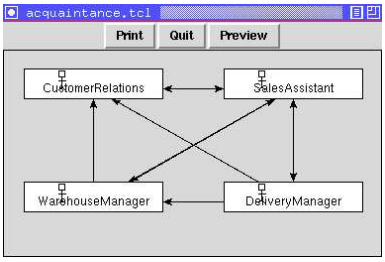
These do not determine the interface of a component, but only constrain it. The "looseness" corresponds to the case where an event can come from both the outside *and* from another child.

A consequence of the definitions above is that an agent or capability *must* export any action or percept (since a percept/action cannot appear in both I_C^+ and O_C^+).

Once cross-checking has been performed the tool generates a number of design diagrams: the *system overview* (all agents and their interfaces), a collection of *agent/capability* diagrams (all capabilities/plans of the agent/capability and their interfaces), a *coupling* diagram (capabilities and exported data), and an *acquaintance* diagram (all agents and links between those agents that interact).

The diagram generation creates the diagram elements and links automatically. This is important for two reasons: firstly, it saves the designer having to manually create multiple diagrams; secondly, and more importantly, it means that mistakes such as leaving out edges or nodes cannot be made.

The placement of nodes (i.e. determining their (x, y) coordinates) is done manually by dragging them – links are automatically repositioned (see below). The diagrams positions are saved and encapsulated postscript is generated.



This encapsulated postscript is used in the generated design report. The report (in LaTeX format) includes a descriptor for each design element including automatically generated cross-referencing. Generated cross-referencing includes for agents and capabilities: what other components they interact with, and via which event/database; and for events and databases: what components read/post/write/receive them.

4 Related Work

There is currently a large amount of work being done in agent-oriented software engineering methodologies (e.g. [1, 3, 4, 5, 6, 9, 10, 11, 12, 13, 14, 18, 19, 22], and see the references in [15]). It is not possible in a brief conference paper to do justice to the significant amount of work being done.

The GAIA methodology [22] 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.

[12] Actually it can also come from the same component. For example, a plan can post an event that will trigger another instance of itself. This is just the agent equivalent of a recursive call.

[13] Only plans contain code that is executed, so if, for example, a capability reads a database or posts an event, then it must contain a plan that does these things.

The Tropos methodology 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 currently only in the form of a diagram editor[14], rather than the consistency checking and automatic generation of some parts of the design that is part of PDT.

The MaSE methodology [5] is one of the few methodologies that appears to have 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" [5, p232]. Thus, MaSE (intentionally) does not support the construction of plan-based agents that are able to provide a flexible mix of reactive and proactive behaviour.

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.

Some approaches are based on taking UML[15] and extending or modifying it, as is done by Odell et. al. [14] as well as others with a slightly different approach (e.g. [16]). This approach is sometimes justified by the observation that agents are just a special case of active objects.

In our experience, just extending UML did not provide sufficient assistance to start thinking in a different paradigm. Although agents can in some ways be seen as a specialised type of object, 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 the current state-of-the-art, where the concepts and notations for designing agent systems are still not agreed, we believe it is best to consider possibilities without the world-view suggested by OO. Our students who just used modified OO design, had enormous difficulty in conceptualising agent applications and ultimately in building good agent systems.

Some of the significant differences between agent oriented design in Prometheus and OO methodologies are:

- The provision of a process for determining the agents in the system.
- Treating messages as components 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.
- 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, 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 the agent methodology accessible to developers. In this respect the work of [16] and [20] is valuable.

Certainly further work to compare and categorise the growing number of proposed agent-oriented software engineering methodologies extending the work of [18] would be a valuable contribution.

5 Discussion and conclusions

We have briefly described the key aspects of the Prometheus methodology. The methodology has been in use for several years as a teaching tool, and has also been taught in industry workshops (most recently at Net.ObjectDays in Germany, October 2002). The feedback we have received indicates that it provides substantial guidance for the process of developing the design and for communicating the design within a work group. With student projects it is abundantly clear that the existence of the methodology is an enormous help in thinking about and deciding on the design issues, as well as conveying the design decisions.

[14] Conversation with Anna Perini at AAMAS'02 (July, 2002)

[15] 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).

There are a number of areas where the Prometheus methodology and PDT are currently being extended. These include extending PDT to support the specification phase and moving from scenarios to interaction protocols, representing and using life-cycle information and cardinality information (e.g. in debugging), testing techniques, and evaluating the methodology. Evaluating a software engineering methodology is difficult to do well, but is important.

We are also investigating how some of the design artifacts, such as the protocol definitions, and the capability diagrams, can be used for providing debugging and tracing support within the implemented system [17]. Having a design methodology which can be used through to testing and debugging is clearly advantageous in terms of an integrated and complete methodology.

Acknowledgements: We would like to acknowledge the support of Agent Oriented Software Pty. Ltd. and of the Australian Research Council (ARC) under grant CO0106934.

References

[1] F. M. T. Brazier, B. M. Dunin-Keplicz, N. R. Jennings, and J. Treur. DESIRE: Modelling multi-agent systems in a compositional formal framework. *Int Journal of Cooperative Information Systems*, 6(1):67–94, 1997.

[2] P. Busetta, R. Rönnquist, A. Hodgson, and A. Lucas. JACK Intelligent Agents - Components for Intelligent Agents in Java. Technical report, Agent Oriented Software Pty. Ltd, Melbourne, Australia, 1998.

[3] G. Caire, F. Leal, P. Chainho, R. Evans, F. Garijo, J. Gomez, J. Pavon, P. Kearney, J. Stark, and P. Massonet. Agent oriented analysis using MESSAGE/UML. In M. Wooldridge, P. Ciancarini, and G. Weiss, editors, *Second International Workshop on Agent-Oriented Software Engineering (AOSE-2001)*, pages 101–108, 2001.

[4] J. Debenham and B. Henderson-Sellers. 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, May 2002.

[5] S. A. DeLoach, M. F. Wood, and C. H. Sparkman. Multiagent systems engineering. *International Journal of Software Engineering and Knowledge Engineering*, 11(3):231–258, 2001.

[6] C. Iglesias, M. Garijo, and J. González. A survey of agent-oriented methodologies. In J. Müller, M. P. Singh, and A. S. Rao, editors, *ATAL-98*, pages 317–330. Springer-Verlag: Heidelberg, Germany, 1999.

[7] N. Jennings and M. Wooldridge. Applications of intelligent agents. In N. R. Jennings and M. J. Wooldridge, editors, *Agent Technology: Foundations, Applications, and Markets*, chapter 1, pages 3–28. Springer, 1998.

[8] N. R. Jennings. An agent-based approach for building complex software systems. *Communications of the ACM*, 44(4):35–41, 2001.

[9] E. A. Kendall, M. T. Malkoun, and C. H. Jiang. A methodology for developing agent based systems. In C. Zhang and D. Lukose, editors, *First Australian Workshop on Distributed Artificial Intelligence*, 1995.

[10] D. Kinny and M. Georgeff. 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, 1996.

[11] D. Kinny, M. Georgeff, and A. Rao. A methodology and modelling technique for systems of BDI agents. In R. van Hoe, editor, *Seventh European Workshop on Modelling Autonomous Agents in a Multi-Agent World*, 1996.

[12] J. Lind. 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"*, 2000.

[13] J. Mylopoulos, J. Castro, and M. Kolp. Tropos: Toward agent-oriented information systems engineering. In *Second International Bi-Conference Workshop on Agent-Oriented Information Systems (AOIS2000)*, June 2000.

[14] J. Odell, H. Parunak, and B. Bauer. Extending UML for agents. In *Proceedings of the Agent-Oriented Information Systems Workshop at the 17th National conference on Artificial Intelligence.*, 2000.

[15] L. Padgham and M. Winikoff. Prometheus: A methodology for developing intelligent agents. In *Third International Workshop on Agent-Oriented Software Engineering*, July 2002.

[16] M. Papasimeon and C. Heinze. Extending the UML for designing JACK agents. In *Proceedings of the Australian Software Engineering Conference (ASWEC 01)*, Aug. 2001.

[17] D. Poutakidis, L. Padgham, and M. Winikoff. 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)*, 2002.

[18] O. Shehory and A. Sturm. Evaluation of modeling techniques for agent-based systems. In J. P. Müller, E. Andre, S. Sen, and C. Frasson, editors, *Proceedings of the Fifth International Conference on Autonomous Agents*, pages 624–631. ACM Press, May 2001.

[19] L. Z. Varga, N. R. Jennings, and D. Cockburn. Integrating intelligent systems into a cooperating community for electricity distribution management. *Int Journal of Expert Systems with Applications*, 7(4):563–579, 1994.

[20] G. Wagner. A UML profile for external AOR models. In *Third International Workshop on Agent-Oriented Software Engineering*, July 2002.

[21] M. Winikoff, L. Padgham, and J. Harland. Simplifying the development of intelligent agents. In M. Stumptner, D. Corbett, and M. Brooks, editors, *AI2001: Advances in Artificial Intelligence. 14th Australian Joint Conference on Artificial Intelligence*, pages 555–568. Springer, LNAI 2256, Dec. 2001.

[22] M. Wooldridge, N. Jennings, and D. Kinny. The Gaia methodology for agent-oriented analysis and design. *Autonomous Agents and Multi-Agent Systems*, 3(3), 2000.