Prometheus: A Methodology for Developing Intelligent Agents

Lin Padgham and Michael Winikoff

RMIT University, GPO Box 2476V, Melbourne, AUSTRALIA
Phone: +61 3 9925 2348
{linpa,winikoff}@cs.rmit.edu.au
http://www.cs.rmit.edu.au

Abstract. As agents gain acceptance as a technology there is a growing need for practical methods for developing agent applications. 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 differs from existing methodologies in that it is a detailed and complete (start to end) methodology for developing intelligent agents which has evolved out of industrial and pedagogical experience. This paper describes the process and the products of the methodology illustrated by a running example.

1 Introduction

As agents are gaining acceptance as a technology and are being used, there is a growing need for practical methods for developing agent applications. This paper presents the *Prometheus*[1] methodology for developing intelligent agent systems.

The 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 activities required in developing intelligent agent systems.

Our claim is that Prometheus is developed in sufficient detail to be used by a non-expert. Prometheus has been taught to an undergraduate class of (third year) students who successfully designed and implemented agent systems using JACK. A second year student over the summer vacation 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.

[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*

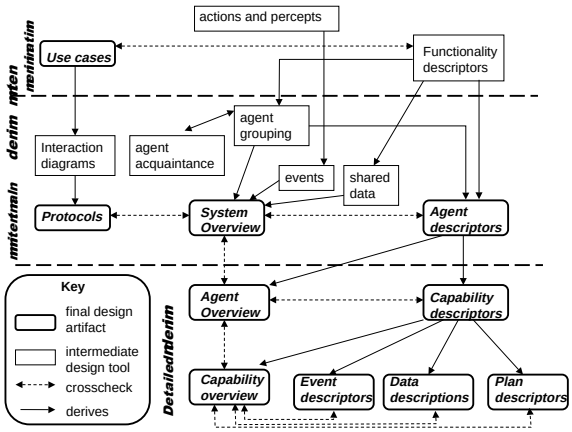
Unfortunately space limitations preclude a detailed comparison with the many existing methodologies. We simply note that Prometheus differs from existing methodologies [1–5, 7–18, 22, 23, 25] in that it:

- Supports the development of *intelligent* agents which use goals, beliefs, plans, and events. By contrast, many other methodologies treat agents as "simple software processes that interact with each other to meet an overall system goal" [6].
- Provides "start-to-end" support (from specification to detailed design and implementation) and a *detailed process*, along with design artifacts constructed and steps for deriving artifacts.
- Evolved out of practical industrial and pedagogical experience, and has been used by both industrial practitioners and by undergraduate students. By contrast, many other methodologies have been used only by their creators and often only on small (and unimplemented) examples.
- Provides hierarchical structuring mechanisms which allow design to be performed at multiple levels of abstraction. Such mechanisms are crucial to the practicality of the methodology on large designs.
- Uses an iterative process over software engineering phases rather than a linear "waterfall" model. Although the phases are described in a sequential fashion in this paper, the intention is *not* to perform them purely in sequence.
- Provides (automatable) cross checking of design artifacts.

Of the properties above, perhaps the most contentious is the first: many existing methodologies intentionally do not support intelligent agents, rather, they aim for generality and treat agents as black boxes. We believe that in this case, generality needs to be sacrificed in favour of usefulness. By specifically supporting the development of BDI-like agents we are able to provide detailed processes and deliverables which are useful to developers. Of course, this makes Prometheus less useful to those developing non-BDI-like agents. However, note that the initial stages of the methodology *are* appropriate for the design of any kind of multi agent system.

Although none of these properties is unique in isolation, their combination is, to the best of our knowledge, unique. We believe that these properties are all essential for a practical methodology that is usable by non-experts and accordingly the design of Prometheus was guided by these properties. Although Prometheus' contribution is the combination of these properties, this combination was achieved through careful design of the methodology. It is not possible to easily construct a methodology which has the above properties by combining methodologies that have some of them. For example, given a methodology that provides automated support but does not support intelligent agents and another methodology that supports intelligent agents but not provide automated cross-checking; it is not at all obvious how a hybrid methodology could be created that supports both features.

The *Prometheus* methodology consists of three phases. 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.



The rest of this paper describes 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 full detail all aspects of the Prometheus methodology. However, we hope that an understanding of its structure and some sense of its phases, deliverables, activities, and depth of detail can be gained.

2 System specification

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. In line with [21] we call incoming information from the environment "percepts", and the mechanisms for affecting the environment "actions".

As discussed in [24] 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. Both can be done within the agent system (either in specific agents, or distributed) or outside of the agent part of the system. If done within the agent part of the system then the Prometheus methodology can be applied, otherwise existing methodologies can be used.

The online bookstore has the percepts of customers visiting the website, selecting items, placing orders (using forms), and receiving email from customers, delivery services and book suppliers. Actions are bank transactions, sending email, and placing delivery orders.

In parallel with discovering 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 functionalities[3] of the system. For example, in order to define the book store we may need to define functionalities such as *"the book store will provide a personalised interface to customers"* and *"the book store will maintain its stock"*. These functionalities start to give an understanding of the system - some sense of its purpose.

It is important in defining functionalities that they be kept as narrow as possible, dealing with a single aspect or 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 the information that is required, and the information produced by it. The functionality descriptor produced 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. For example, the following describes the *welcomer* functionality in the online bookstore.

Welcomer: provides a customised response to the user when they log into the site.
Actions: provide link to status of existing orders, welcome by name, welcome as new user, query enjoyment of recent purchases, indicate special offers relevant to interests.
Percepts: Customer accesses site.
Data access: Reads customer information, special offers, and customer interactions data. Writes customer interactions data.
Interactions: No interactions with other functionalities.

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

The central part of a use case scenario in Prometheus is the sequence of steps describing an example of the system in operation. Each step is annotated with the name of the functionality responsible, as well as information used or produced. These annotations allow cross checking for consistency with the functionality descriptors.

The use case templates which we use contain an **identification number**, a brief natural language **overview**, an optional field called **context** which indicates when this scenario would happen, or the start point of the scenario, the **scenario** itself which is a 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 use case scenario.

3 Architectural design

The major decision to be made during the architectural design is which agents should exist. We assign functionalities to agents by analysing the artifacts of the previous phase

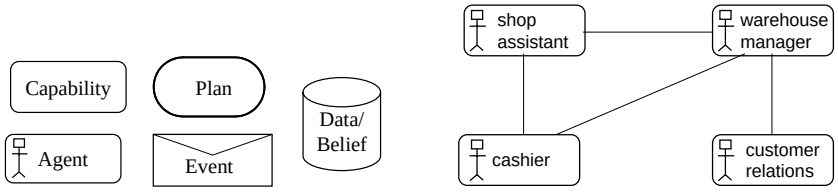
[3] A number of methodologies call these "roles". We prefer to avoid overloading the term since it has a similar, but non-identical, meaning in the context of teams of agents.

to suggest possible assignments of functionalities to agents. These are then evaluated according to the traditional software engineering criteria of coherence and coupling.

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 coherence and loose coupling.

It can be useful at this stage to draw a matrix having all functionalities on one axis and the properties or relationships on the other axis. Specific properties and relationships that are useful in deriving groupings of functionalities are whether two functionalities are *related*, whether they are *clearly unrelated*, the *data used*[4] and *data produced* as well as *information received* from other functionalities and data that is *written by two (or more) functionalities*. The last two columns can be derived from the information in the previous columns.

In order 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. The design for the book store depicted below (on the right) is reasonable, since it indicates low coupling. A design which produced an acquaintance diagram where each agent was linked to every other agent would be highly undesirable. Note that Prometheus uses a consistent notation to depict agents, events, plans, capabilities, etc. This notation is summarised below, on the left.



A simple heuristic for assessing coherence is whether an agent has a simple descriptive name which encompasses all the functionalities without any conjunctions (“and”). For example, the shop assistant agent combines the functionalities of visit manager, client welcomer, query processor, pro-active helper, and customer DB manager; yet it has a simple descriptive name.

Once a decision has been made as to which agents the system should contain it is possible to start working out and describing some of the necessary information about agents. The high level information about each agent is contained in the form of an agent descriptor, 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

[4] Both *read* (in the case of data stores) and *received* (in the case of events and messages).

done? What data does this agent need to keep track of? What events will this agent react to?

Agent descriptors contain the above information plus the **name** and **description** of the agent, a brief description of ways in which it **interacts** with other agents and a list of the **functionalities** which are incorporated within this agent. For example consider the following agent descriptor:
Name: Shop assistant agent
Description: greets customer, follows through site, assists with finding books
Cardinality: 1/customer. Instantiated on customer arrival at site
Functionalities included: visit manager, client welcomer, query processor, pro-active helper, customer DB manager.
Reads data: user profile, client orders
Writes data: user profile
Interacts with: cashier (to pay for purchase); warehouse manager (to get price, availability and to hand over order for shipping)

Consistency checking should be done to ensure that agent descriptors are consistent with the set of functionality descriptors which have been incorporated within the agent. Particular items to check are the information, and the agent interactions. If a functionality is listed as interacting with another functionality, then this should translate into an agent interaction, unless the two functionalities are incorporated within the same agent.

At this stage of the design it is important to identify what **events** (i.e. significant occurrences) will be generated as a result of information from the environment (the percepts), either directly or after processing. These are the things the agents will notice, which will cause them to react in some way. A decision must be made as to which agents will react to which events.

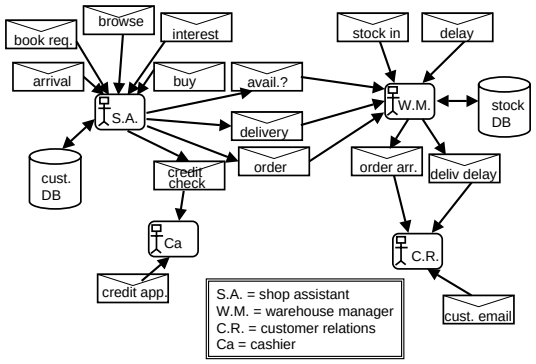
In order to accomplish the various aims of the system agents will also send messages to each other. These must also be identified at this stage. It is also necessary to identify what information fields will be carried in these messages, as this forms the interface definition between the agents.

If the implementation platform does not provide specialised message types, these must be specified precisely at this stage to enable modularity in the development of the detailed design for the individual agents.

Shared data objects (if any) must also be identified at this stage. A good design will minimise these, but there may be situations where it is reasonable to have shared data objects. 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 what at first appears to be a shared data object can be reconceptualised as a data source managed by a single agent, with information provided to other agents as they need it. Alternatively each agent may have its own version of the information, without there being any need for a single centralised data object. Data objects should be specified using traditional object oriented techniques.

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. A system overview diagram for the running example is below. By viewing this diagram we obtain a general

understanding of how the system as a whole will function. Messages between agents can include a reply, although this is not shown explicitly on the diagram. Looking further at agent descriptors provides any additional detail needed to understand the high level functioning of the system.



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, showing interaction between agents rather than objects. One of the main processes for developing interaction diagrams is to take the use cases developed in the specification phase and to build corresponding interaction diagrams. Wherever there is a step in the use case which involves a functionality from a new agent there must be some interaction from a previously involved agent to the newly participating agent. While it is not possible to automatically derive the interaction diagrams from the use cases, substantial consistency checking is possible. Figure 1 (left) shows an interaction diagram for a scenario of buying a book at our electronic bookstore. It depicts the user requesting a particular book from the sales assistant which checks the details with the warehouse then replies. The user decides to buy the book and places an order; the sales assistant checks for delivery options, confirms them with the user, checks the user's credit card details with the cashier, and then places the order and thanks the user. In addition to deriving interaction diagrams from use cases, each of the major environmental events should have an associated interaction diagram.

Interaction diagrams, like use cases, 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** which define precisely which interaction sequences are valid within the system.

Figure 1 (right) shows the protocol for the credit check portion of the interaction diagram shown in figure 1 (left). Because protocols must show all variations they are often larger than the corresponding interaction diagram and may need to be split into smaller chunks. We use the AUML notation [18] to specify protocols,

Consistency checking should be done between protocols and interaction diagrams, the system overview diagram, and use cases. With the appropriate tools, much of this consistency checking can be automated.

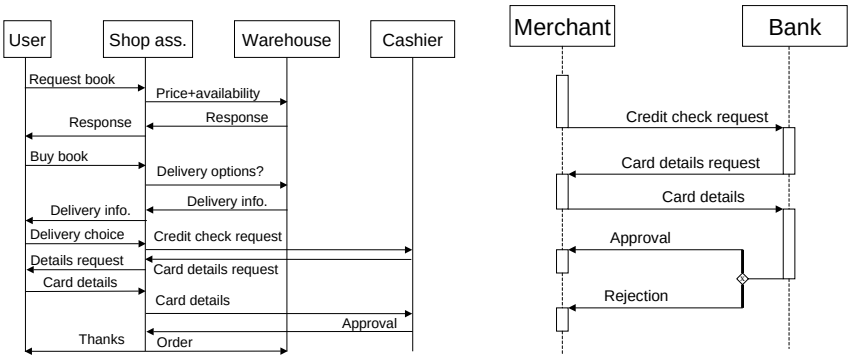


Fig. 1. Example Interaction Diagram (left) and Protocol (right)

4 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. It is at this stage of the design that the methodology becomes specific to agents that use user-defined plans, triggered by goals or events, such as the various implementations of Belief, Desire, Intention (BDI) systems (e.g. PRS, dMARS, JAM, or JACK). A number of details regarding the implementation platform also become evident at this stage when looking at any particular design. However, 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 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.

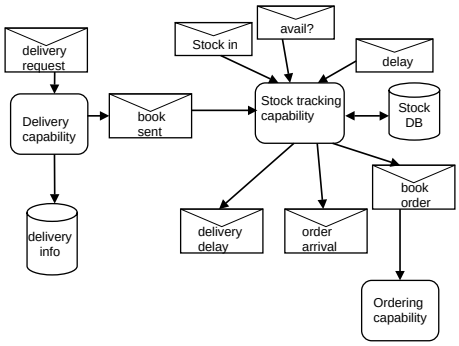
Capabilities are allowed to be nested within other capabilities and thus this model allows for arbitrarily many layers within the detailed design, in order to achieve an understandable complexity at each level.

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** by (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.

The agent overview diagram below is for a warehouse manager agent in the electronic bookstore. This agent has the capabilities of tracking stock, placing orders for new stock and organising delivery of books to clients.



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.

Just as the agent overview diagram should be checked for consistency with the system overview (in terms of incoming and outgoing events), so each capability overview diagram should be checked against its enclosing context - either the agent overview, or another capability overview.

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**, 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 whether it is expected to be *covered* and whether it is expected to be *unambiguous*. An event is *covered* if there is always at least one handling plan which is applicable; that is, for any situation, at least one of the matching plans will have a true context condition. An event is *unambiguous* if there is always at *most* one handling plan which is applicable.

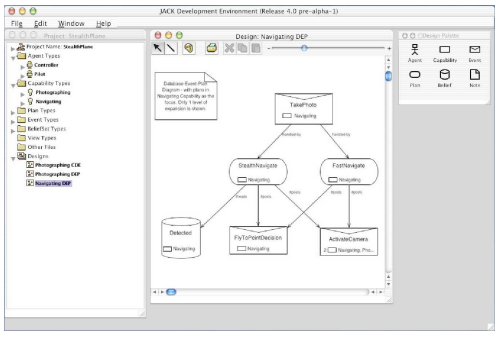
Data descriptors should specify the fields and methods of any classes used for data storage within the system. If specialised data structures are provided for maintaining beliefs, these should also be specified.

An additional artifact that is completed (and checked) at this point 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). One option is to organise the data dictionary into separate sections for agents, capabilities, plans, events and data, organised alphabetically within sections. The other option is to have a flat alphabetical structure. With tool support multiple views (automatically generated) can be provided.

5 Discussion and conclusions

We have briefly described the key aspects of the Prometheus methodology. The methodology has been in use for several years and has been taught in industry workshops (most recently at the Australian AI conference, 2001). The methodology has been in use for several years as a teaching tool. 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.

One of the advantages of this methodology is the number of places where automated tools can be used for consistency checking across the various artifacts of the design process. For example, the input and output events for an agent must be the same on the system overview diagram and on the agent overview diagram. Agent Oriented Software has constructed a support tool for the methodology



that allows design diagrams to be drawn and generates corresponding skeleton code (in JACK).

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 [20]. Having a design methodology which can be used through to testing and debugging is clearly advantageous in terms of an integrated and complete methodology.

Other areas for future work include: clearer integration of goals as a first class concept (currently goals are implicit in functionalities), extensions to the graphical notation to allow percepts, actions, goals, and (some) sequencing information to be specified; introduction of social concepts (teams, roles) and a clarification of the distinction between functionalities and roles; and investigating the commonalities and differences with various extensions of UML to agents [18, 19]. Additionally, we intend to integrate Prometheus with the agent concepts we have identified [24].

Acknowledgements: We would like to acknowledge the support of Agent Oriented Software Pty. Ltd. and of the Australian Research Council (ARC) under grant CO0106934. We would also like to thank James Harland and Jamie Curmi for comments on drafts of this paper.

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. B. Burmeister. Models and methodology for agent-oriented analysis and design. Working Notes of the KI'96 Workshop on AgentOriented Programming and Distributed Systems, 1996.
3. G. Bush, S. Cranefield, and M. Purvis. The Styx agent methodology. The Information Science Discussion Paper Series 2001/02, Department of Information Science, University of Otago, New Zealand., Jan. 2001. Available from *http://divcom.otago.ac.nz/infosci*.
4. 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.
5. A. Collinot, A. Drogoul, and P. Benhamou. Agent oriented design of a soccer robot team. In *Proceedings of ICMAS'96*, 1996.
6. S. A. DeLoach. Analysis and design using MaSE and agentTool. In *Proceedings of the 12th Midwest Artificial Intelligence and Cognitive Science Conference (MAICS 2001)*, 2001.
7. 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.
8. A. Drogoul and J. Zucker. Methodological issues for designing multi-agent systems with machine learning techniques: Capitalizing experiences from the robocup challenge. Technical Report LIP6 1998/041, Laboratoire d'Informatique de Paris 6, 1998.
9. M. Elammari and W. Lalonde. An agent-oriented methodology: High-level and intermediate models. In G. Wagner and E. Yu, editors, Proc. of the 1st Int. Workshop. on Agent-Oriented Information Systems., 1999.

10. N. Glaser. The CoMoMAS methodology and environment for multi-agent system development. In C. Zhang and D. Lukose, editors, *Multi-Agent Systems Methodologies and Applications*, pages 1–16. Springer LNAI 1286, Aug. 1996. Second Australian Workshop on Distributed Artificial Intelligence.
11. C. Iglesias, M. Garijo, and J. González. A survey of agent-oriented methodologies. In J. Müller, M. P. Singh, and A. S. Rao, editors, *Proceedings of the 5th International Workshop on Intelligent Agents V : Agent Theories, Architectures, and Languages (ATAL-98)*, volume 1555, pages 317–330. Springer-Verlag: Heidelberg, Germany, 1999.
12. C. A. Iglesias, M. Garijo, J. C. González, and J. R. Velasco. Analysis and design of multiagent systems using MAS-commonKADS. In *Agent Theories, Architectures, and Languages*, pages 313–327, 1997.
13. 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.
14. 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.
15. 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.
16. 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.
17. 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.
18. 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.
19. M. Papasimeon and C. Heinze. Extending the UML for designing JACK agents. In *Proceedings of the Australian Software Engineering Conference (ASWEC 01)*, Aug. 2001.
20. D. Poutakidis, L. Padgham, and M. Winikoff. Debugging multi-agent systems using design artifacts: The case of interaction protocols. To appear in the proceedings of the First International Joint Conference on Autonomous Agents and Multi Agent Systems, 2002.
21. S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall, 1995.
22. 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.
23. 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.
24. 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.
25. 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.