Adding debugging support to the Prometheus methodology

Padgham, Lin; Winikoff, Michael; Poutakidis, David

Padgham, L., Winikoff, M., & Poutakidis, D. (2005). Adding debugging support to the Prometheus methodology. Engineering Applications of Artificial Intelligence, 18(2), 173–190.
https://doi.org/10.1016/j.engappai.2004.11.018
Document Version: Accepted Manuscript

Published Version: https://doi.org/10.1016/j.engappai.2004.11.018

Citation: Padgham, L, Winikoff, M and Poutakidis, D 2005, 'Adding debugging support to the Prometheus methodology', Engineering, vol. 18, no. 2, pp. 173-190.

Adding Debugging Support to the Prometheus Methodology

Lin Padgham, Michael Winikoff, David Poutakidis

School of Computer Science and Information Technology
RMIT University, Melbourne, Australia

Abstract

This paper describes a debugger which uses the design artifacts of the Prometheus agent oriented software engineering methodology to alert the developer testing the system, that a specification has been violated. Detailed information is provided regarding the error which can help the developer in locating its source. Interaction protocols specified during design, are converted to executable Petri net representations. The system can then be monitored at run time to identify situations which do not conform to specified protocols. A process for monitoring aspects of plan selection is also described. The paper then describes the Prometheus Design Tool, developed to support the Prometheus methodology, and presents a vision of an integrated development environment providing full life cycle support for the development of agent systems. The initial part of the paper provides a detailed summary of the Prometheus methodology and the artifacts on which the debugger is based.

1 Introduction

Prometheus is a methodology which has been developed to support the building of intelligent agent systems. An important aspect of this methodology is the focus on covering all phases of development. In particular it has a well developed detailed design phase, leading easily into code, which is absent in a number of agent oriented software engineering methodologies (e.g. GAIA (Wooldridge et al., 2000), Tropos (Bresciani et al., 2002)) The artifacts developed during Prometheus' detailed design phase map directly to concepts provided by a range of agent implementation platforms. In particular, skeleton code for JACK^{TM} agents can be readily produced from a Prometheus

Email address: linpa@cs.rmit.edu.au, winikoff@cs.rmit.edu.au, davpout@cs.rmit.edu.au (Lin Padgham, Michael Winikoff, David Poutakidis).

design by mapping the detailed design into the JACK Development Environment (JDE) provided with JACK. Also, Sudeikat et al. (2004) mention that they have developed a tool for translating design files produced with the Prometheus Design Tool (PDT) into Jadex Agent Definition Files.

This paper discusses work to extend the Prometheus methodology and the Prometheus Design Tool, to cover the testing and debugging activities of the software development lifecycle. Testing and debugging is linked to analysis and design, and to the artifacts developed through these activities. Use of this process can also ensure that implementation and design artifacts are in fact consistent, making code more maintainable over the long term.

2 System Specification

The system specification phase, as with all software engineering methodologies, is about clarifying and developing a clear high level view of the expectations of the system. In Prometheus this phase [1] focuses on:

- Identification of *actors* and their interactions with the system.
- Developing *scenarios* illustrating the system's operation.
- Identification of the *system goals* and sub-goals.
- Specifying the interface between the system and its environment in terms of *actions*, *percepts* and any *external data*.
- Grouping goals and other items into the basic *functionalities* of the system.

These steps are not sequential, but rather, one shifts between them, using one aspect to help develop others. There are especially strong links between goals and scenarios, with each scenario being linked to a goal of the same name. Although the description of these steps follows a stepwise process in developing the system specification, it is important to stress that in fact the process can be started at any step, and always involves shifting back and forth until the overall picture is sufficiently complete.

2.1 Actors and their use-cases

The first step is to identify which actors (these may be people, agents, or other systems) will interact in some direct way with the system to be developed.

[1] This phase as described here has been further refined since publication of *Developing Intelligent Agent Systems: A practical guide* by the authors. We acknowledge and thank Mikhail Perepletchikov for his work on this aspect of the methodology, during his honours project.

Each of these actors is identified, along with *use cases* or interactive situations each will be involved in. This identifies some initial use cases of the system.

Communications between the actors and the multi agent system being developed are identified as inputs (to the system) and outputs (from the system). Following standard practice in agent systems (e.g. (Russell and Norvig, 1995)) the system inputs are referred to as *percepts*, and the system outputs as *actions*. This information can be shown graphically, as in figure 1 in much the same way as actors and use cases are depicted in object oriented analysis.

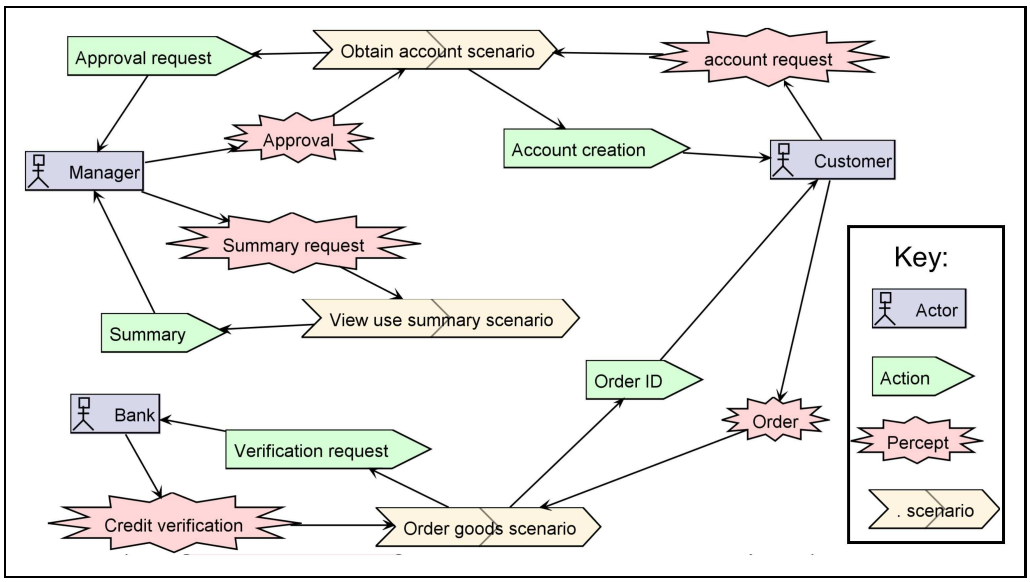

Fig. 1. Actors and use cases, with percepts and actions identified

2.2 Development of scenarios

Each use case defined as part of the actor identification is then developed as a detailed scenario. The aim of scenario development is to identify the steps that describe what may happen in a typical scenario. This provides an accessible and easy to understand view of system operation, that can easily be understood by clients or co-developers. The steps within a Scenario can be of five different types: Goal, Scenario, Action, Percept or Other. (Other can be useful occasionally for capturing such things as a wait). Thus scenarios in Prometheus are more structured than those in object oriented design. This structure supports automated reasoning to ensure consistency between the different views of the system.

Scenarios (like all entities in the design) also have a descriptor which carries additional information regarding the scenario, including information about scenario variations.

Goals provide a useful, succinct, high level description of both the motivation for developing the system, and more operationally, what it is expected to do. An initial set of goals can be obtained by defining a goal for each initial use case identified in the process of identifying actors and interactions. Further initial goals can be extracted from a textual description of the system.

Additional goals can be identified by a process of abstraction and refinement with respect to the initial set of goals. Asking the question "how?" can provide either a sequence of subgoals that can accomplish the given goal, or alternative ways of achieving the goal. Asking "why?" can provide more abstract goals, which with further "why?" questions may facilitate identification of alternative ways of achieving these goals (van Lamsweerde, 2001).

The process of abstraction and refinement leads to goal hierarchies. Some goals arising in different contexts can also then be coalesced, giving a (directed acyclic) graph, rather than a tree. Figure 2 gives an example of a partially developed goal hierarchy.

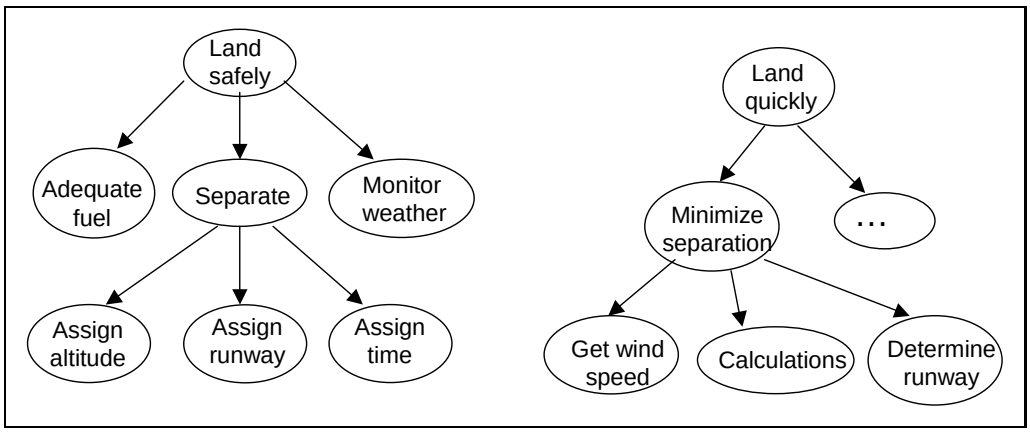

Fig. 2. Example of goal hierarchy

There are no hard rules as to when to stop with goal refinement. While it is important to ensure that one identifies sufficient goals, producing overly low-level goals, such as "obtain preferred hour for meeting", is not useful. Refining below the level needed as steps to explain scenarios is usually not useful at this stage.

2.4 System interface - percepts, actions, data

Agent systems are always situated in an environment. Interaction with a changing environment during runtime is expected, and typically the agent

system also affects that environment. Thus it is critically important early on to identify and obtain clarity on the interface between the agent system and the hardware, software, or people that make up its environment.

As mentioned previously the interface is defined primarily in terms of *actions* (ways that the agents affect the environment), and *percepts* (incoming information from the environment). External data that is used or produced by the agent system is also a part of the interface.

Percepts and actions should be identified as part of the process of identifying actors and their interactions with the system. However sometimes necessary percepts or actions are identified separately from this process, which then may lead back to actor identification to determine whether new actors should be identified as the source or recipient of the action/percept. Scenario development can also lead to identification of the need for a new percept or action which may require revisiting actor identification.

2.5 Grouping into functionalities

Functionalities [2] are chunks of system behaviour which logically belong together. Functionalities allow for grouping entities into modules that have a specific purpose, but which are more comprehensive than a single goal. In identifying functionalities we follow the basic software engineering principle of modularity. Specifically, functionalities should be cohesive and easy to describe fully in two to three sentences [3].

To obtain functionalities we start by grouping goals into natural subgroups. This usually involves taking lower level goals from different places in the goal hierarchy to achieve sensible groupings. Figure 3 illustrates this principle with a simple example. Actions and percepts are then also placed into functionalities.

Functionalities are related back to scenarios by requiring that each step in a scenario is annotated with the functionality it belongs to (this can be automated by a support tool, based on membership of goals, percepts and actions in functionalities). Data used and produced at each step in a scenario is also identified, and this is added into the relevant functionalities.

[2] For those familiar with other agent oriented design methodologies, functionalities are quite similar to *roles*. However it was deemed preferable to reserve the term role for use in the agent team setting, where it has a somewhat different meaning.
[3] Coupling is considered when grouping functionalities into agents (see section 3.1).

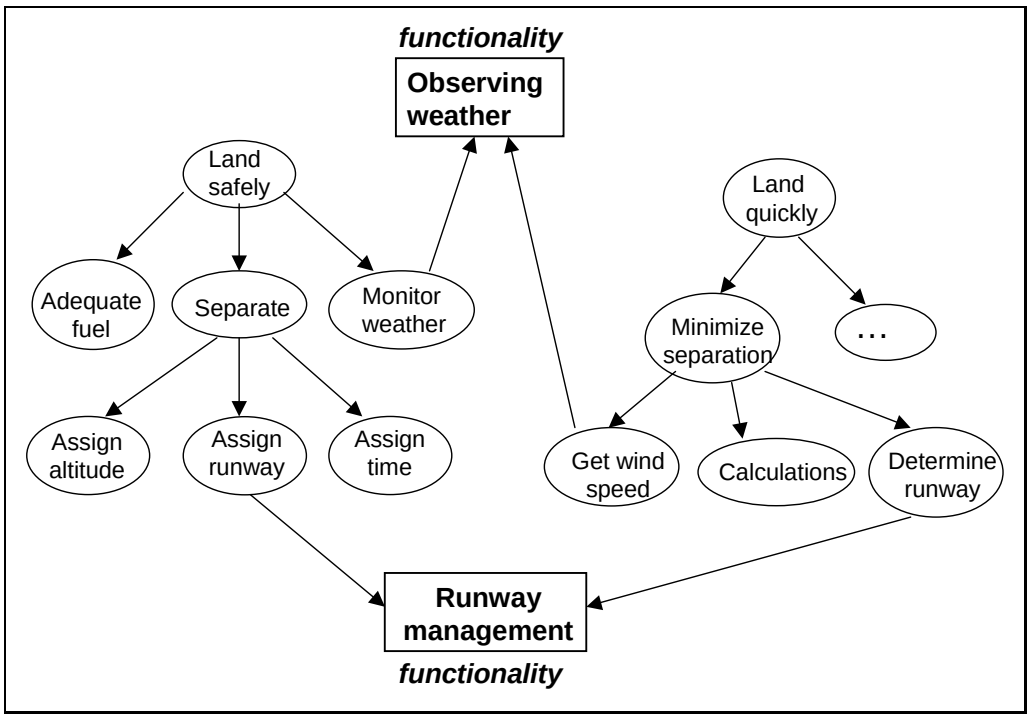

Fig. 3. Grouping goals into functionalities

3 Architectural Design

The architectural design phase refines the system specification to determine the agent types within the system, and fully specifies the interactions between these agent types. The main steps in this phase are as follows:

- Deciding what *agent types* will be implemented and developing the *agent descriptors*
- Describing the dynamic behaviour of the system using *interaction diagrams* and *interaction protocols*.
- Capturing the system's overall (static) structure using the *system overview diagram*.

As with the system specification phase, although we present these steps in a workable sequence, the reality is an iterative process where work on one aspect affects other aspects. It is also very common that at the architectural design phase some modifications may be made, or issues identified, which lead back to further work or modifications in the system specification phase.

The major decision to be made during the architectural design is which agent types should exist. This is done by grouping functionalities into agent types, with each type consisting of one or more functionalities.

Given a set of functionalities there is a large number of possible groupings. Deciding on a reasonable grouping is guided primarily by considerations of data coupling and cohesion, although in particular application areas other issues may also be important, such as the coupling arising due to participation in decision making in control systems (Bussmann et al., 2004).

Prometheus provides the concept of a *data coupling diagram* to help guide the decision process regarding groupings of functionalities, and an *agent acquaintance diagram* to help in comparing different possibilities.

The data coupling diagram represents the functionalities, with links showing the use and production of groups of data. This diagram can help in finding groupings which are relatively loosely data coupled. Various design decisions can be made to eliminate some data sharing, and in particular functionalities which write to the same data store should be in the same agent. Figure 4 shows a simple data coupling diagram and two possible groupings.

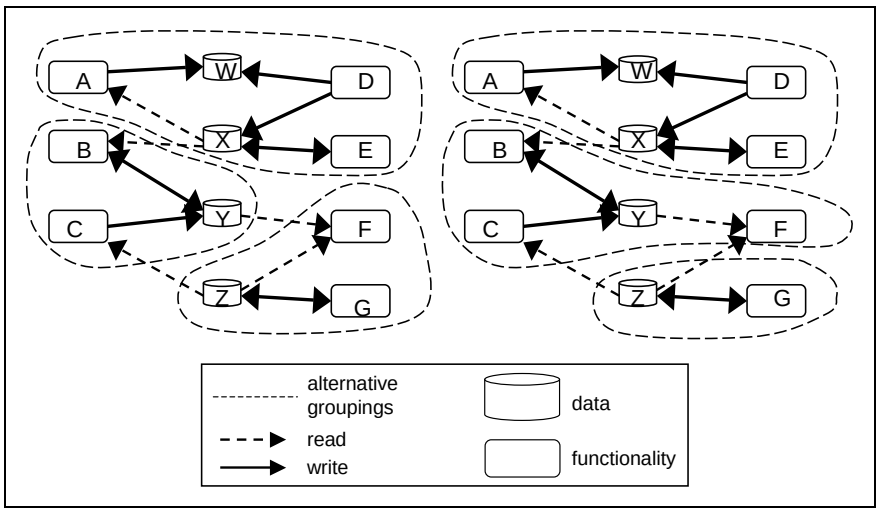

Fig. 4. Data coupling diagram showing different groupings

For example functionalities A, D and E seem to group together, as do B and C. Functionalities F and G could be grouped with each other, or F could be combined with B and C, with G standing alone. This gives two possible groupings which can then be compared using an agent acquaintance diagram. The cohesiveness of the groupings is also a critical (and in fact an overriding) factor in deciding whether a given grouping is appropriate.

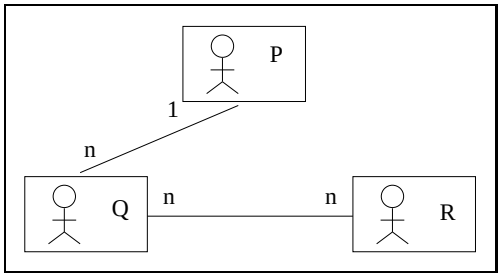

Fig. 5. Agent acquaintance diagram

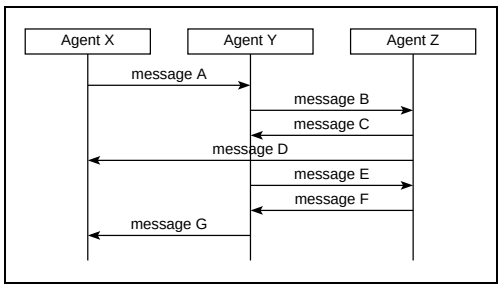

Fig. 6. Interaction diagram

The agent acquaintance diagram simply represents each grouping as an agent, places a link where there is communication between the agents [4], and then (optionally) annotates the links with information regarding the number of instances of each type of agent. For example figure 5 shows an agent acquaintance diagram based on one of the suggested groupings from figure 4, where it is assumed that there is one agent of type P in the system, and multiple agents of type Q and R.

All else being equal, designs with fewer links in the agent acquaintance diagram are preferred, as are ones where the links are not one to many, due to potential bottleneck problems. However there are no hard and fast rules and these are only tools to help in the analysis regarding suitability of possible groupings.

Once the agent types are decided a descriptor should also be filled in for each agent type, covering a range of information. Some of this is automatically filled in if using the PDT support tool, while other things need to be considered as part of the design. Items in the latter category include such things as agent initialisation and demise.

3.2 Agent interactions

Once the agent types have been decided it is possible to start to define the interactions between them. The scenarios developed earlier assist in this process. The first step is to convert each scenario to an agent interaction diagram. This is not something which can be automated, but there are heuristics which can be used to assist in the process.

Taking the scenario steps, annotation with functionalities is replaced by annotation with agent types, based on the groupings determined. At each transition between steps it can then be asked whether any message is necessary between agents for that step to be carried out. If a transition between steps involves

[4] Which is implied by a link existing on the data coupling diagram, since an agent can only access data in another agent by communicating with it.

also a transition from one agent to another, this is an indication that a message is likely to be required. Certainly each agent must receive some message within the interaction before it is involved in a scenario step.

This process results in a number of interaction diagrams, which are similar to UML sequence diagrams but with agents rather than objects. Like scenarios, interaction diagrams show only particular instances of system behaviour. They do not capture the full system behaviour, nor do they show alternatives. An example interaction diagram is shown in figure 6.

The developed interaction diagrams are generalised to interaction protocols which fully define the interactions between agents. This is done by considering at each point in the interaction diagram, what else could occur at that point. AUML-2 (Huget and Odell, 2004) is the current notation used to specify interaction protocols as it appears to be an emerging standard. However any similar notation would be suitable.

Figure 7 shows an example AUML-2 protocol [5] . AUML-2 shows roles, although Prometheus uses agents for tagging the *lifelines*, which are the dashed vertical lines. Messages are directed arrows between lifelines. Time increases down the page, although various *boxes* modify this, showing such things as alternatives and parallel statements (such as the alternative box labelled alt in figure 7). Padgham and Winikoff (2004) Appendix C, provides an introduction to the AUML-2 constructs used in Prometheus, while Huget and Odell (2004) provides full details.

3.3 System overview

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. It is obtained by linking interface entities (percepts, actions and external data) to specific agent types, and by showing the interaction protocols connecting agent types. Shared internal data can also be shown.

Figure 8 shows an example system overview diagram. It should be noted that all information for the system overview diagram exists within the design specified so far, and can be automatically assembled via a support tool. The system overview diagram simply brings information together in an easy to visualise summary.

[5] It is the AUML-2 version of the Request protocol approved as standard by FIPA (Foundation for Intelligent Physical Agents), a standards body for agents, (http://www.fipa.org/specs/fipa00026/)

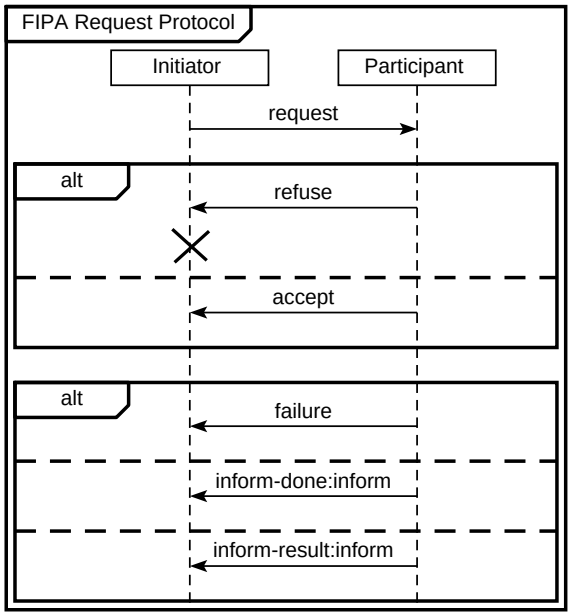

Fig. 7. FIPA Request Protocol using AUML-2 notation

4 Detailed Design

The focus of detailed design is on developing the internal structure of each agent and how it will achieve its functioning within the system. The details of agent functioning are specified using *plans*, which are essentially recipes for agent acting. Plans may be abstract, referring to subgoals, or subtasks. The process allows for progressive refinement, first defining *capabilities* (modules within the agent), and then *plans* along with internal *messages* or *events*, and detailed *data structures*. We also use the protocols to define process diagrams showing the internal processing within each agent. Process diagrams in turn guide the development of plans. The various aspects of the detailed design process are as follows:

- Identifying and developing *capabilities* and their inter-relationships.
- Development of process diagrams showing the *internal processing* of each agent related to the protocol specifications.
- Development of *plans, events* and *data* and their inter-relationships.
- Finalising details for all entities.

4.1 Identifying Capabilities

The detailed design process begins by describing agents' internals in terms of capabilities. The internal structure of each capability is then described,

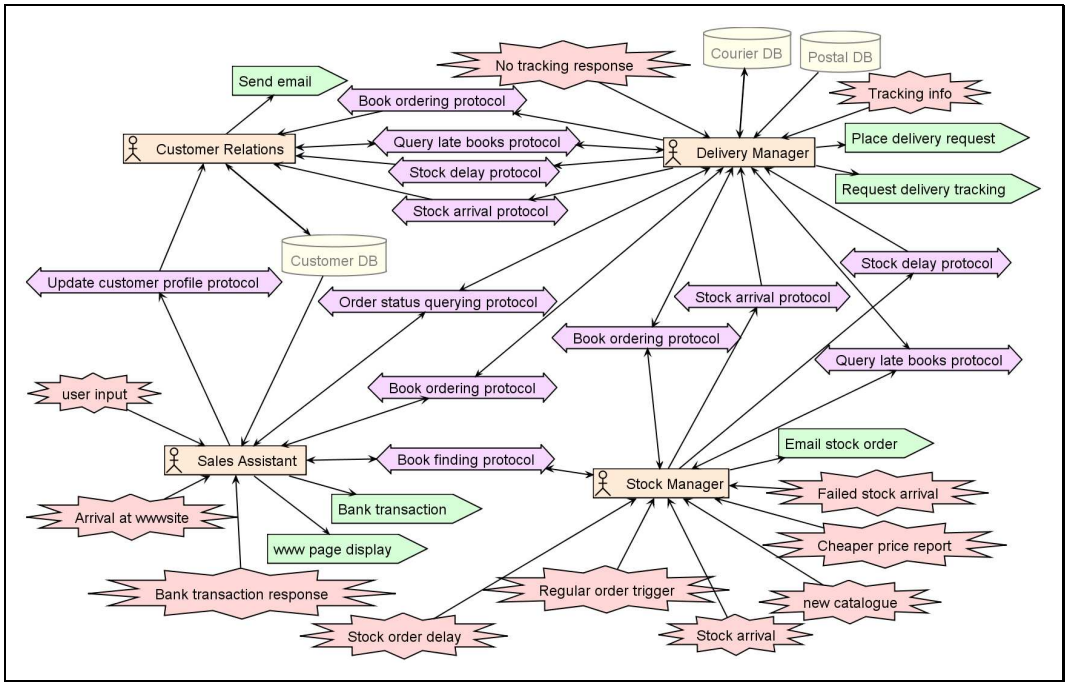

Fig. 8. System overview diagram for electronic bookstore

optionally using or introducing further capabilities [6] . 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. Capabilities are essentially an encapsulation mechanism and non-encapsulated plans, events and data can also be included alongside capabilities at any level.

These specifications are developed using *agent overview diagrams* and *capability overview diagrams*, which are similar to the system overview diagram in form, except that they contain individual messages rather than protocols, and capabilities and plans rather than agents. Figure 9 shows an example of an agent overview diagram.

The incoming and outgoing entities in each agent/capability overview diagram must always be the same as the inputs and outputs to that entity within its parent diagram. These nested overviews with increasingly narrower focus and greater detail allow for a systematic development of detailed design. They also provide a way of partitioning the system, allowing different people to work on different agents/capabilities.

[6] Capabilities are allowed to be nested within other capabilities and thus this model allows for as many layers within the detailed design as are needed in order to achieve an understandable level of complexity at each level.

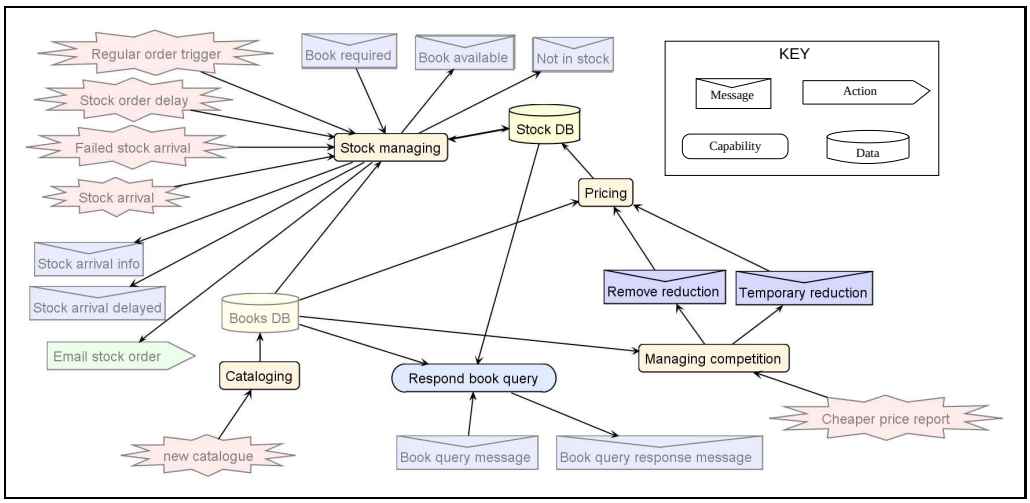

Fig. 9. Agent overview diagram: stock manager

4.2 Agent Processes Using Activity Diagrams

The protocols produced during the architectural design give a "global" view of the interaction showing all agents and the interactions between them. In proceeding towards implementation *process specifications* are derived which are local to a given agent. Each global protocol will have a number of corresponding local views that define the process from the point of view of different agents.

For describing the process a slight variant of UML activity diagrams is used [7]. Figure 10 illustrates the concepts and the notation of the extended variant of activity diagrams. Rather than using swimlanes to separate activities of different agents, as would perhaps be the most obvious modification of UML, the focus is only on the activity within a single agent, indicating interaction with other agents via the inclusion of messages within the diagram. This avoids diagrams being overly cluttered, and perhaps more importantly, it allows for modular development of agents, with shared knowledge only about the interface.

4.3 Plans, Events and Data

The detailed design process continues with design of the *plans* that each capability contains, along with the triggering *events* and the associated *data* (or beliefs). The relationships between these are captured in the *capability overview* diagram, while the details are in the various descriptors.

[7] For further information on standard Activity Diagrams see (Fowler and Scott, 2003).

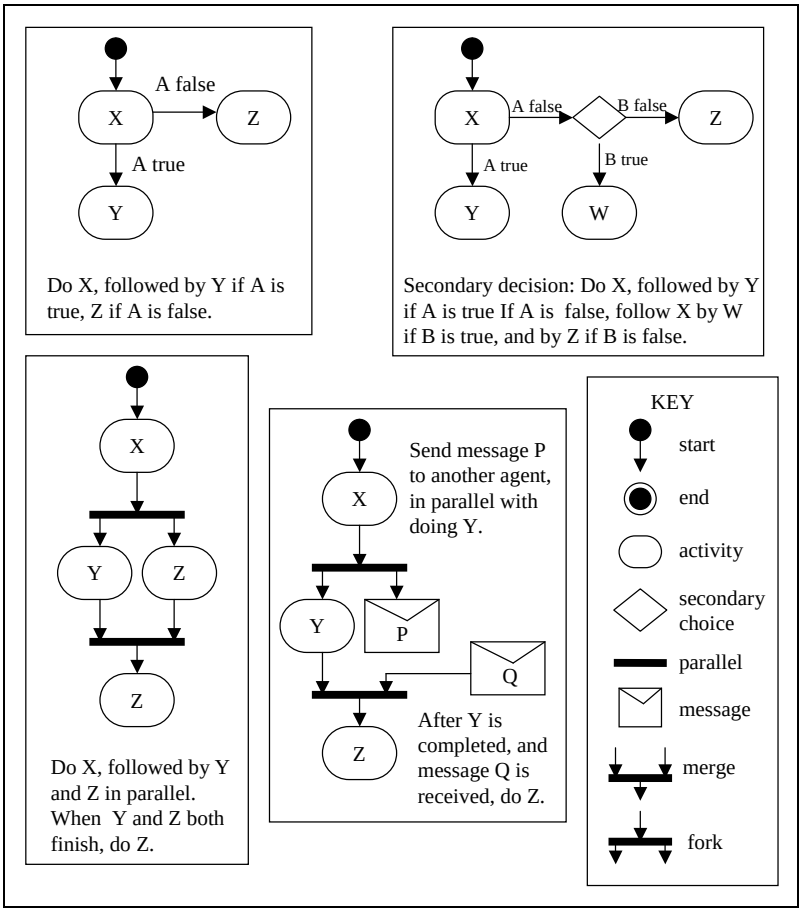

Fig. 10. Diagram illustrating notation for Process diagrams

It is only at this stage that a commitment is made to a particular implementation platfrom. Specifically, there is an assumption that plans are triggered by events and that it is possible to have multiple plans that handle a given event type, where the choice of plan to be used is determined at run-time. This assumption corresponds to a whole class of implementation platforms including BDI systems such as JACK (Busetta et al., 1998) and systems based on hierarchical task networks such as RETSINA (Sycara et al., 2003).

Plans are dynamic, and in developing them one must take into account both their static context (i.e. what triggers them, what actions they can perform, what messages they can send/receive) and also the process specifications.

Events/messages must be specified in detail regarding what information they carry as well as indicating coverage (whether there will always be an appropriate plan to respond to this event) and overlap (whether it is possible there will be more than one plan available to respond to this event). Data must also be specified in detail, typically using either database or object oriented specifications. Plan descriptors contain pseudo-code describing how the plan will operate.

The descriptors for the various entities 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.

As well as finalising descriptors for all entities in the system, the data dictionary (or entity dictionary) should be updated and checked for completeness.

5 Debugging Using Design Artifacts

An important phase of developing software is debugging it. This is often described as the process of locating and fixing a specific piece of code responsible for the violation of a known specification (Hailpern and Santhanam, 2002). Thus it includes detecting errors, identifying what is causing them, and then fixing them. It is suggested that debugging and testing may occupy between 25 and 50 percent of the total cost and time of system development with much of this time spent locating the cause of a problem (Boehm, 1981; Zelkowitz, 1978).

A common technique for debugging multi-agent systems involves the collection and visualisation of information to improve the understanding of the execution of multi-agent systems, in the hope that this will assist in both identifying that an error exists, and also identifying its source. For example, it may be possible to see that no messages are directed to a specific agent.

Communication is integral to a multi-agent system and therefore the exchange of messages between agents is the typical candidate for visualisation (Liedekerke and Avouris, 1995; Ndumu et al., 1999), although other properties are gathered and visualised such as the decomposition and status of tasks, or the internal states of individual agents as identified in (Nwana et al., 1999). These methods all rely on the developer observing errors, which may be very difficult, given the amount of information being shown.

One way to address this problem of information overload, is to automate detection of errors, and to present information only when there is a potential problem identified. The possibilities for automatic detection of bugs have been traditionally limited to environments where the requirements have been formally specified. However, structured non-formal specifications of system behaviour, such as those found in Prometheus also offer opportunities for detecting run-time executions inconsistent with the specification. Using the design specifications for this purpose, in addition to aiding debugging, also helps

to ensure that design models and code remain consistent with one another. The benefits of linking the debugging process to the overall software development process have in fact been recognised since the early days of Software Engineering Development Environments (Müllerburg, 1983).

The debugging mechanisms described in the following sections have been implemented and were introduced in (Poutakidis et al., 2002). They have also been tested on a substantial implementation project of a meeting scheduler, originally implemented as a class assignment. Work is currently underway to experimentally evaluate, with a range of developers, to what extent the debugging tool actually expedites the identification and location of some of the typical agent system problems described in (Poutakidis et al., 2003).

The following sections describe first how protocol specifications are used to identify and locate interaction errors. Section 5.4 then describes how event and plan descriptors developed during detailed design in Prometheus, can be used to identify common errors in plan specification leading to unintended consequences for plan selection.

5.1 Debugging agent interactions using protocol specifications

The protocols developed during design specify allowable message exchanges between agents. Proving that a system implemented with an arbitrary programming language is correct with respect to a particular protocol is typically not possible. However, it is possible to check that a given execution of a system does not violate existing protocols. A debugging tool which monitors execution can therefore detect violations of the protocols as specified and can notify the developer.

Violations of an interaction protocol, such as a failure to receive an expected message or receiving an unexpected message can be both automatically detected and precisely explained (e.g. "agent X received message m which was unexpected – the agent was participating in protocol P and was expecting either n or l"). The information both identifies that there is a problem and assists in locating the cause of the problem.

Monitoring can be done via eavesdropping on the communication medium (e.g. Heselius (2002)), or more directly by requiring that carbon copies of any messages sent also be sent to the debugging agent as done in the ZEUS toolkit (Nwana et al., 1999). The approach developed in relation to Prometheus involves automatically adding code that sends copies of messages to a monitoring agent (Poutakidis et al., 2002).

For the debugger to reason about the correctness of a message within a par-

ticular conversation it needs to compare it against the protocol specification. AUML-2 protocols are a useful artifact for describing interactions between agents but they are not readily machine understandable. Instead Petri nets are used internally by the debugging agent to represent the interaction protocols. A range of notations for representing interaction protocols can be translated into Petri nets, including the original AUML notation which was initially used by Poutakidis et al. (2002), as well as the AUML-2 notation used here.

Additionally, Petri nets have clear formal semantics and provide both a static and dynamic view of the protocols they represent. The debugger monitors the executing system and is able to track interactions and detect problems by comparing the interactions against the protocol specifications.

5.2 Converting from protocol specifications to Petri nets

A Petri net (named after Carl Adam Petri) consists of places (depicted as circles) and transitions (depicted as rectangles) which are linked by arrows (Reisig, 1985). Additionally, places may contain tokens. The placement of tokens on a net is its *marking*, and executing ("firing") a Petri net consists of moving tokens around according to a simple rule; the places, transitions, and the links between them remain unchanged.

A transition in a Petri net is *enabled* if each incoming place (i.e. a place with an arrow going to the transition) has at least one token. An enabled transition can be *fired* by removing a token from each incoming place and placing a token on each outgoing place (i.e. each place with an arrow from the transition to it). For example, in figure 11, the transition is enabled; and fires by removing a token from a and from P and placing a token on Q.

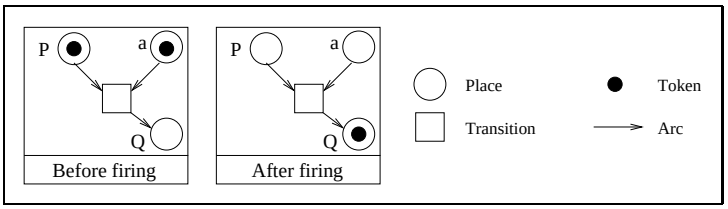

Fig. 11. Example of a Petri net firing

5.2.1 Translation process

The process of converting an AUML-2 protocol to an equivalent Petri net involves first converting protocol fragments to Petri net fragments, and then merging these into a single Petri net for the protocol. The derived Petri nets

contain two different kinds of places: *message places* and *state places*. Connections between the places are represented using Petri net transitions, with particular patterns identified by the protocol operators such as *ALTernative*, *PARallel*, *OPTional*, etc.

Message places are labelled with the message name appended to the role of the message sender. For example, a message of type A, sent by an agent of type B, would result in a message place with name B:A. State places are identified and named by a process of labelling the start and end points of messages in the protocol, and then using this labelling in forming the Petri net state places. The operator within which a protocol fragment exists affects the labelling of the protocol, as well as affecting the way in which places are linked via transitions.

Figure 12 shows the process for converting the most simple protocol – a single message – to a Petri net suitable for use by the debugger. The resulting Petri net has three places, the two state places, A and B, for the start and end of the message, and one message place, *Initiator:request*. The underlying intuition in connecting the places is that the place representing the receiving end of a message should receive a token when the place representing the initiating end of that message and the place representing the message itself are both marked with a token. In this example the transition *T1* is therefore introduced with places *A* and *Initiator:request* as incoming places and *B* as an outgoing place.

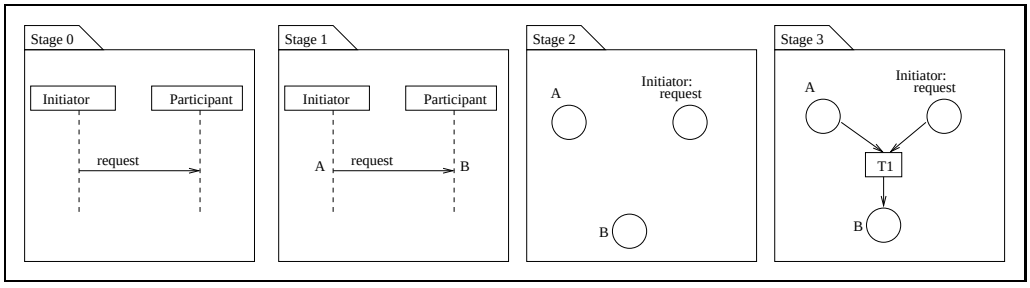

Fig. 12. Process of converting a single message to Petri net representation

The following sections describe the process of labelling and converting a subset of AUML components to their equivalent Petri net fragments [8]. Merging of Petri net fragments into the Petri net representing the complete protocol is done by merging places with the same name, and is trivial. An example is given when discussing the *OPTional* component.

[8] Due to space limitations it is not possible to show all patterns. The reader is referred to the upcoming thesis (expected 2004) by the third author for full details

5.2.2 ALTernative

Alternative (or selection), shown in figure 13, provides the agent with a choice as to which message can be sent. If the protocol is in state P the agent can send message x in which case the protocol will advance into state Q. Alternatively the agent can send message y and the protocol will advance into state R. The Petri net version of the *ALTernative* fragment is presented alongside the AUML-2 version. As the places representing the start of message x and the start of message y are alternatives available at a single point in the protocol, a single state place is used in the Petri net. The two alternative transitions enable the executing Petri net to follow either the branch into state Q, or into state R.

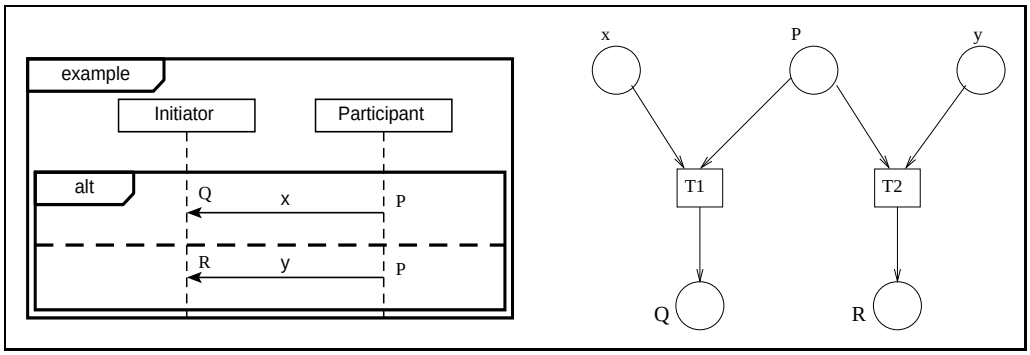

Fig. 13. ALTernative interaction operator

5.2.3 PARallelism

The PARallelism connector as shown in figure 14 requires that **both** message x and message y are sent when in state P, but the order is not specified. The translation adds two extra places, P' and P'', these places are intermediate places that take input from T1 and T2 respectively. The Petri net works as follows: when in state P either an x or a y message is expected. If, for example, an x messages is received then T1 will be enabled. Following the Petri net rules, tokens will be removed from place P and place x, a token will be deposited on each of P' and Q. The conversation has now split into multiple states, since multiple tokens now exist in the Petri net. After receiving the x message there is still the need to receive the y message. When the y message is received it is placed in the y place, since a token is on place P' transition $T4$ is enabled resulting in a successful firing of the net and a token being deposited on place R. It should be evident that this Petri net will also receive the opposite sequence of y followed by x, in which case $T2$, P'' and $T3$ will be utilised.

A conversation may not proceed beyond a parallel region until all of the messages inside that region have been sent. Therefore, if figure 14 had a message, say z, being sent from the Initiator to the Participant after the parallel region,

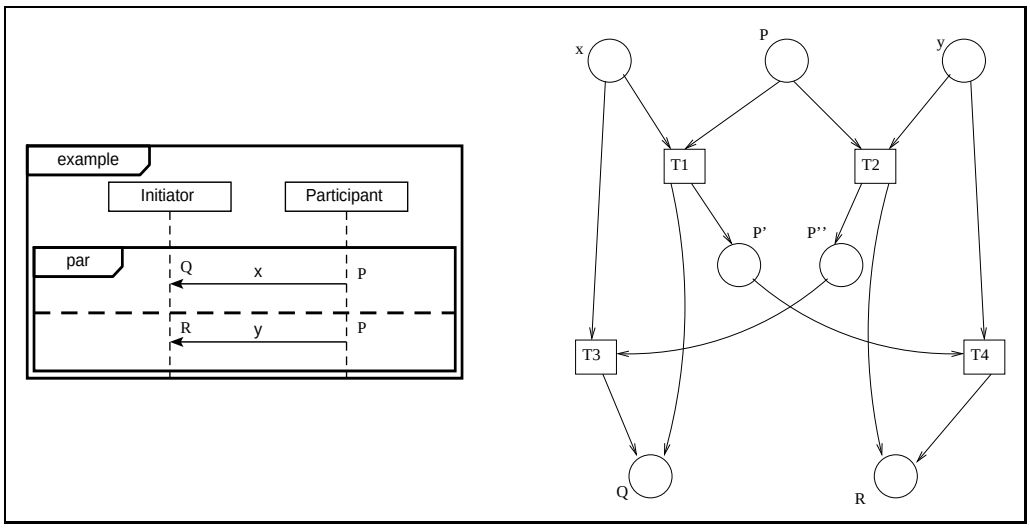

Fig. 14. PARallel interaction operator

then z can only be sent after both x and y have been sent. Furthermore, after the parallel region has been processed the conversation no longer has multiple paths of execution. Therefore the tokens from each of the final state places of the parallel paths need to be collected into a single intermediate state. A single transition takes input from the final states, Q and R, and will only fire when each input place contains a token. A single output place ensures that multiple tokens are merged into a single token representing a single possible path of execution.

5.2.4 OPTional

The optional operator shown in figure 15 specifies that everything bound by the optional box may or may not be executed. Thus when the protocol reaches state Q, it can either be followed by message y leading to state R, which is in turn followed by message z leading to state S; or it can be followed directly by message z leading to state S. This can be captured by a labelling that indicates that the start state for message z can be either Q or R. The initial message x from state P to state Q is exactly analogous to figure 12 and is not shown in the resulting Petri net. The three simple Petri net transitions shown in figure 15 immediately below the protocol, represent the three single message representations, where the multiple place labelling at the initiation of z leads to two separate Petri net fragments involving the message place z.

By merging both state and message places with the same names the Petri net shown in figure 16 is obtained. As can be seen this allows place S to receive a token by combining message place z with either state place R (resulting from the optional interaction) via transition T3, or with state place Q via transition T2. A similar mechanism is used to merge all the fragments obtained from mapping the protocol fragments into the final protocol.

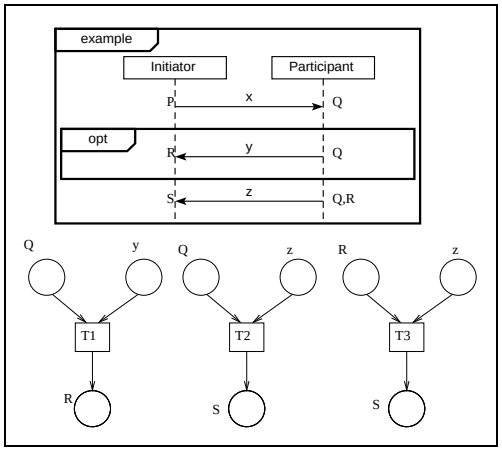

Fig. 15. OPTional operator affecting labelling process.

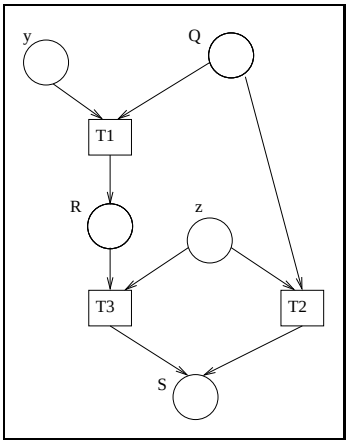

Fig. 16. Merged Petri net fragments.

5.3 Monitoring and reporting

The debugger has a library of the specified interaction protocols that it uses to detect errors. The debugger keeps a list of active conversations so that it can monitor multiple interleaved conversations simultaneously. When a message is received from an agent it is added to the appropriate conversation [9] (or a new conversation is instantiated), and is then processed by firing any enabled transitions.

For any given conversation the debugger does not know what protocol the agents are following. Therefore the debugger keeps a list of *possible protocols*, instantiated from the interaction protocol library, which currently match the sequence of messages within the conversation [10] . As the conversation progresses the possible protocols list is reduced whenever a message is received that causes an error in the individual protocol. The conversation is still considered valid as long as there is at least one entry in the possible protocols list.

Each time the debugger receives a message for a specific conversation an analysis is done on each protocol within the possible protocols list of that conversation to identify error situations. As long as there is more than one protocol in the possible protocols list, errors simply lead to the conclusion that this protocol was not in fact the one that was being followed, and it is discarded from the list. However if an error is detected in the only remaining protocol,

[9] Conversations are identified by a conversation id which is added automatically when compiled with the debugger option.

[10] FIPA allows for the inclusion of a protocol name in their messages and if it is included then this list would not be needed, see http://www.fipa.org/specs/fipa00061/XC00061E.html#_Toc505483417

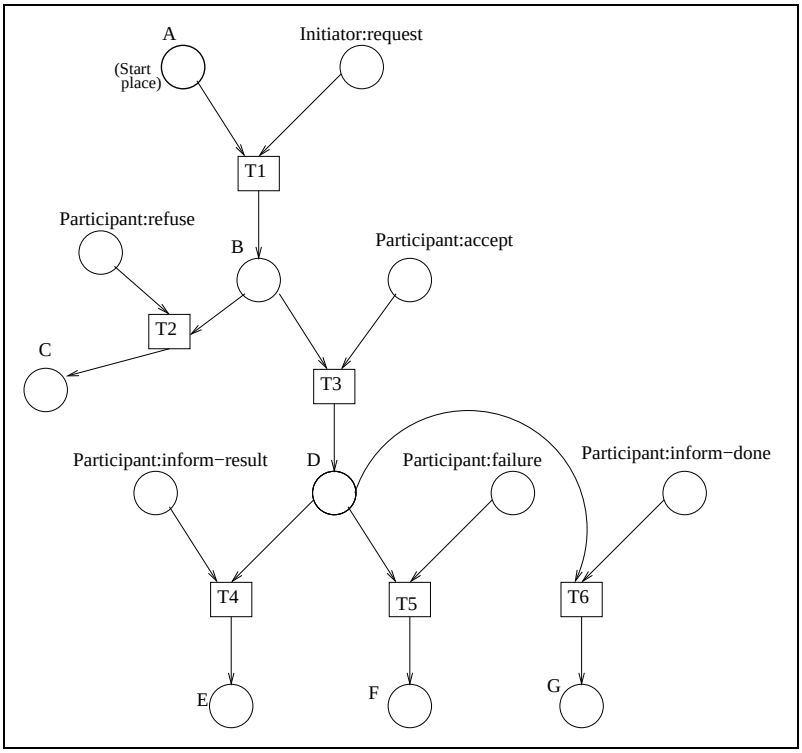

Fig. 17. Request protocol converted to equivalent Petri net

then this is reported.

An example error situation is when the Petri net has no message place matching the incoming message, on which a token could be placed. Another is when, after firing the Petri net, a token remains on a message place. Correct functioning leaves tokens only on state places [11] .

Figure 17 shows the request protocol from figure 7 converted into the Petri net representation. When the first *request* message is received, the protocol is initialised by placing a token in the start place A and in the message place *Initiator:request*. Transition $T1$ then fires, placing a token in state place B. At the next step in this protocol either a *refuse* or an *accept* message is expected. Receiving either message will place a token on the corresponding message place, causing either $T2$ or $T3$ to fire. Assume that an *accept* message is received and the conversation advances to state D. If there were a bug in the system, causing a *refuse* message to be sent erroneously, this will cause a token to be placed on the *refuse* message place. However, since there is no token on state place B (the token was removed when $T3$ fired), no transition is enabled. The token is left on the message place after the net fires, indicating an error, *an unexpected message was received*.

[11] Recall that the Petri net places were identified as either state places or message places in section 5.2.1.

An additional aspect of the debugger ensures, prior to placing a token on a message place, that a message being sent is appropriate for the role the agent has taken on in the protocol. For example, once it has been determined that agent A is playing the *initiator* role, and agent B is playing the *participant* role in the request protocol, then agent A sending a *result* message would be detected as an error. Roles are assigned to agents when the first message for a role within the protocol occurs. An agent can take on multiple roles within a single protocol.

Error detection based on reception of messages, as described, will fail to detect conditions where a message should be sent but is not. In order to detect such problems, whenever a token is placed on a new state place, a timer is set for that state place. If this timer expires then it is inferred that a message that should have been sent has not been sent. The debugger identifies that a problem could exist in the system and a warning is reported. If the message arrives after the warning is given, the conversation once again becomes valid, and is resumed.

The semantics of time within a protocol is currently not well specified in AUML-2. If a protocol were to specify that a message *must* be received before a particular time then the debugger could report an error rather than a warning.

This technique of monitoring conversations between agents is capable of automatically identifying incorrect interaction patterns. An incorrect execution pattern is typically the result of a lower level coding error in one of the agents. The debugger provides information about the protocol that the agents were engaging in, the agent types and instances that were executing, and the point at which a conversation diverged from the allowed behaviour. This provides valuable information to the developer to assist in locating the exact cause of the error.

5.4 Debugging plan interrelationships

In BDI agent systems such as JACK (Busetta et al., 1998), JAM (Huber, 1999), and JADEX (A. Pokahr, 2003) which choose an appropriate pre-defined plan from a plan library, one common cause of errors is incorrect specification of when a plan should be used. This often results in one of two situations: either there is no plan suitable to respond to a given goal or event, resulting in the goal not being attempted or the event not being reacted to; or alternatively there may be multiple suitable plans, and the one chosen is not the one intended.

The detailed design part of the Prometheus methodology focuses on BDI style implementation platforms, which use a plan library. Each plan is tagged as

being *relevant* to a particular goal or event. Usually there will be multiple plans relevant for any given goal/event. A *context condition* in the plan specifies the particular environmental situation in which that plan can be used for responding to the event/goal for which it is relevant. The set of plans which are relevant for a particular goal/event, and whose context conditions are true at a particular time, are referred to as the *applicable plans* at that time. These are the plans suitable for responding to the event/goal.

The Prometheus methodology prompts the developer to consider how many plans are expected to be suitable for various situations. For each event the developer is asked to specify whether it is ever expected that either multiple plans will be applicable, or that no plans will be applicable. Two concepts are introduced within Prometheus in order to facilitate this consideration. They are *coverage* and *overlap*. Having full coverage specifies that the event is expected to have an applicable plan found under all circumstances. Overlap specifies that it is possible, although not required, that multiple plans are applicable at the time the event occurs.

Full coverage means that the context conditions of the plans relevant for that event must not have any "holes". A typical unintended hole that can occur is when two plans are specified for an event, one with context say $temperature < 0°$ and the other with context $temperature > 0°$. $Temperature = 0°$ is then a "hole" and if that is the situation when the event occurs, no plan will be applicable. If at design time the developer specifies that an event type has full coverage, and yet at runtime a situation occurs when there is no applicable plan for an event of that type, then an error can be reported.

For an event to have *no overlap* requires that the context conditions of plans relevant for that event are mutually exclusive. If overlap is intended, the developer is prompted to specify whether plans should be tried in a particular order, and if so how that will be accomplished. Overlap can occur due either to *two or more plan types* being applicable, or due to there being *multiple instances of a single plan type*, due to different variable bindings. The developer is also prompted at design time to specify which of these situations is expected if overlap is possible.

Violations of expectations regarding coverage and overlap can both be accomplished in theory by simply examining the applicable plan set at the appropriate point after an event has been processed. However, depending on the implementation platform being used, the applicable plan set may not be directly available. In the implemented debugger two different mechanisms are used for detecting overlap and lack of coverage. This is due to the fact that access to the applicable plan set in JACK is only supported when the plan set is non-empty.

If an event is specified as having no overlap, but the applicable plan set for that event is at some point found to be greater than one, then an error is reported, along with information regarding the event and the applicable plans.

In cases where an event is specified as having full coverage, a small piece of code is inserted when compiling with the debugger, to ensure that the debugger is notified each time an event of this type occurs. Code is also inserted within all plans defined as relevant for that event, so that when they are executed, the debugger is notified. If the debugger is notified that an event occurred, but there is no corresponding notification that a relevant plan has started executing (within a given time period), then it can be assumed that there is an error and this is reported.

The overhead in detecting coverage and overlap problems is very small. Only a few statements for each of the events and plans that are specified as having full coverage or no overlap and a small monitoring module local to the running application needs to be added. The reporting of the coverage and overlap bugs is integrated with the interaction monitoring tool at the user interface, although the two debugging modules are currently independent of each other. Monitoring of coverage and overlap very easily detects at an early stage, a class of bugs often found in (BDI style) agent systems.

6 The Prometheus Design Tool

The Prometheus Design Tool (PDT) is a prototype tool designed to provide support for the Prometheus methodology. It is under ongoing development and the vision is for it to provide, or be integrated into, a full development environment.

The current version of the tool provides a graphical user interface which supports developers in designing most of the design artifacts within the Prometheus methodology, including form based descriptors for the various entities. It also does a range of automated propagation where possible, as well as cross-checking to help ensure consistency and completeness. In addition PDT can produce diagrams for inclusion in a report, or a full HTML design document.

The following sections describe the support provided for various aspects of the Prometheus design process, the automation and cross checking, and the envisioned extension to a fuller development environment, including the debugging and testing described in the previous sections.

Figure 18 shows a screen shot of the interface to PDT. As can be seen, the three main phases of system specification, architectural design and detailed design each provide a range of diagrams which can be chosen from the menu on the left hand side. Below this menu is a scrollable list of all the individual entities within the project being developed, organised by entity type. This list can be filtered to provide only certain entity types, for example only goals and scenarios. The bottom right hand window provides the descriptor form for whichever entity is selected, either from the diagram above this window, or from the entity list to the left.

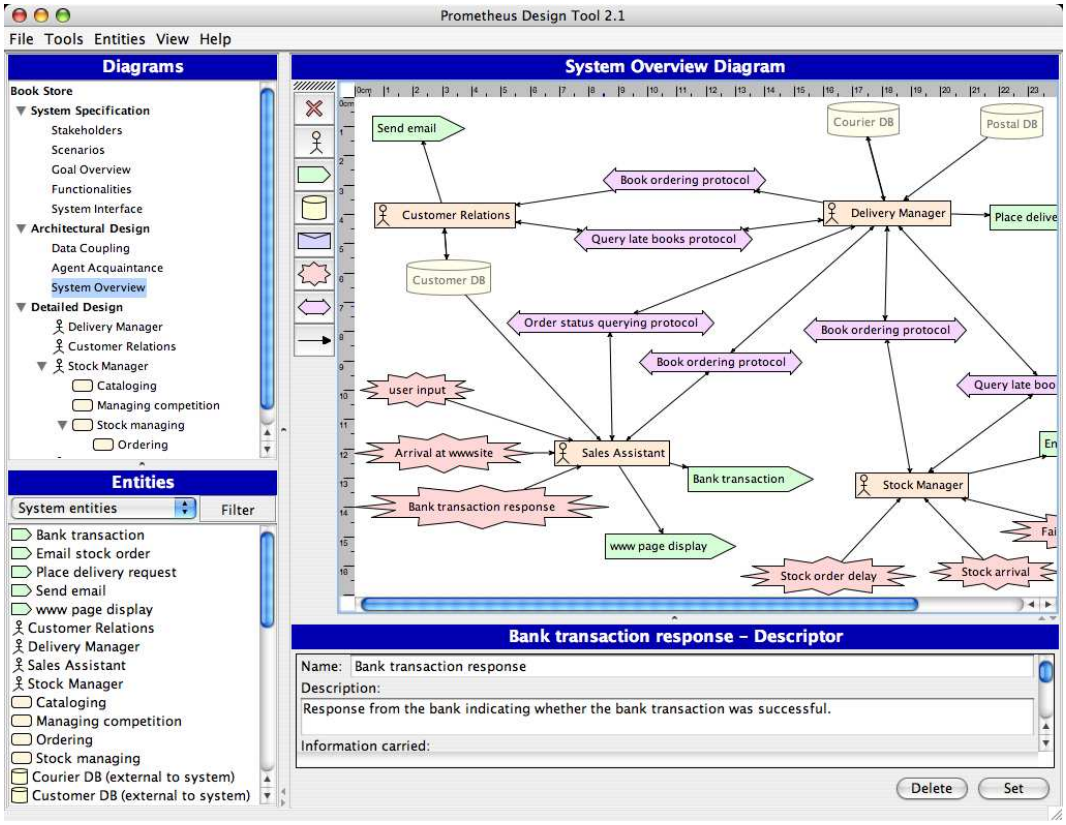

Fig. 18. Screen shot of Prometheus Design Tool

All diagram types allow the user to place already created entities available for that diagram type, onto the canvas, or to instantiate and place new entities of an appropriate type. Linking and unlinking of entities is supported and only *legal* links are allowed.

6.1.1 System Specification

The actors or stakeholders diagram allows developers to place actors, scenarios, percepts and actions onto the canvas, thus creating the initial top level view of the system. Identification of scenarios leads also to automatic creation of corresponding goal entities. Percepts and actions which are the inputs to and outputs from a scenario are automatically inserted as initial steps into the scenario representation, which is available by either double clicking on the icon in a diagram, or from the "edit" menu.

The goal diagram allows the user to arrange existing goals, identified via creation of scenarios, and to add and link in new ones, identified by asking "how?" and "why?". The functionality diagram allows the user to group goals, percepts and actions into functionalities.

6.1.2 Architectural Design

The main support for architectural design is the ability to produce the system overview diagram, and to maintain the consistency of this diagram with the protocol specifications with respect to interactions between agents.

Currently there is no graphical editor for interaction diagrams or for the full protocol specification. This must be done outside PDT. However, protocol entities can be created and edited, either via the system overview diagram, or the entities menu. Information stored within the protocol representation indicates which messages pass between which agents. The messages can be ordered to assist in capturing some intuitive understanding of the possible full specification of the protocol. The information represented is sufficient to support generation and consistency maintenance of the system overview diagram, which is the central visual representation of the overall system and its subsystems.

There is not really any significant support for making decisions regarding agent types, although it is possible to produce data coupling diagrams within the tool, and relationships between data and functionalities described in the data coupling diagram are propagated into descriptors.

6.1.3 Detailed Design

The primary support for the detailed design are the agent and capability overview diagrams, available from the detailed design menu, after the agents and the capabilities have been identified. The interface elements for an agent

overview diagram [12] are automatically propagated from the system overview diagram. The developer can then add from the graphical interface, the internals of the agent, in terms of capabilities, plans and within-agent messages (or events). There is currently no support for process diagrams within PDT.

6.2 Checking for Consistency and Completeness

One of the most important advantages of using PDT is the assistance it provides in developing a consistent design. It reduces a large number of small errors which can cause considerable confusion when multiple people are working on a design and must understand each others thinking by reading the documentation provided via the design artifacts. Even the simple functionality of ensuring naming consistency by offering menu choices, rather than requiring the user to recall and type in the names of various entities, is extremely helpful.

There are also a number of checks that are done to help ensure consistency and completeness. These include such things as checking that all data is used somewhere and produced somewhere; checking for interface consistency between the detailed specifications of entities (such as agents or capabilities) and their specification within a parent entity (such as system or agent); and checking that all functionalities are included in some agent. There are a wide range of consistency checks that are able to be run in PDT, producing a list of errors and warnings.

6.3 Integrating Debugging and Testing into PDT

The vision is that PDT should support all stages of system development, from specification through to implementation and testing and debugging. The debugging work that has been developed within the methodology lends itself to being incorporated as it is based on the design artifacts of the methodology. However it is not yet incorporated within PDT. There are a number of pieces of software development that must be accomplished before that will be achieved.

Firstly, it must be possible to generate code. This has not been a high priority to date for two reasons: reluctance to be too closely tied to a particular agent development environment; and the fact that once the design is developed in PDT it can fairly easily be manually transferred to the Jack Development

[12] Capability overview diagrams operate in the same way as agent overview diagrams, but with the interface taken from the relevant agent overview, rather than from the system overview.

Environment (JDE) produced by Agent Oriented Software, which does produce skeleton code. Of course in the longer term, manual translation between different systems is not sufficient.

Secondly it must be possibly to more fully specify protocols. This has also been a lower priority than other things due to the desire to conform to standards (AUML) that are still emerging. The hope is that PDT will be able to incorporate an AUML tool developed elsewhere.

Finally there are still some parts of the conversion process from protocols to the Petri net representation used for debugging which require manual processing. Either these must be successfully automated, or an appropriate interface to allow specification of necessary information by the developer must be added.

Thus it appears that the research issues have been sufficiently addressed that debugging could certainly be included within PDT already. However due to the resources required for the actual development of the software there will be some delay before it is incorporated.

6.4 Future Developments

PDT is under ongoing development, driven in part by research goals, and in part by feedback from users. Currently it is too focussed on documentation of the final (or interim) design artifacts, with too little support for exploration of design possibilities. For example there is currently no mechanism for identifying explicitly, within the data coupling diagram, potential alternate groupings. Neither is there any real support for exploring potential groupings using an agent acquaintance diagram. This is a priority to address, in order to better support the approach of the Prometheus methodology.

Also, there is currently no support for multiple users working on the same design. This is clearly an area that needs to be addressed to some extent if it is to be more widely useful. The ability to read in partial files and add them to the design may be a way to relatively simply address this at some level. In this way developers could at least work separately on the design of individual agents, combining them with the system specification and architectural design, to build the overall system.

7 Related Work

A large number of agent-oriented methodologies have been proposed in recent years (Bergenti et al., 2004). Compared with other methodologies, Prometheus'

distinguishing features are that it is aimed at industrial practitioners (as well as students), that it aims to be detailed and complete, that it emphasises the importance of tool support and automated consistency checking, and that it supports the detailed design of plan-based agents. The structured nature of the design artifacts allow for development of support structures for debugging as demonstrated in this paper.

Prometheus has some similarities with the **Gaia** methodology (Wooldridge et al., 2000), for example, our agent acquaintance diagrams are essentially the same as those used by Gaia, and the roles of Gaia are somewhat similar in concept to functionalities in Prometheus. However Gaia's lack of a detailed design process means it does not provide sufficient guidance for inexperienced agent developers.

Similarly, Prometheus provides a more detailed process than the **Tropos** methodology (Bresciani et al., 2002; Giunchiglia et al., 2002), and provides tool support and cross checking. Tropos provides an early requirements process that goes beyond that provided by Prometheus, but we have found that the somewhat simpler process of the current Prometheus approach is preferable.

The **MaSE** methodology (DeLoach et al., 2001) is one of the few methodologies with significant tool support. However, the fact that MaSE views agents "... merely as a convenient abstraction, which may or may not possess intelligence" (DeLoach et al., 2001, p232) makes it less suitable than Prometheus for detailed design of plan based agents, including BDI systems.

PASSI (Cossentino and Potts, 2002; Burrafato and Cossentino, 2002) also provides tool support and processes for going from requirements to code. It is based on UML and particularly focuses on supporting reuse of agent patterns (Cossentino et al., 2003), which is an interesting area to incorporate into methodologies.

Comparisons of Prometheus with other agent-oriented methodologies can be found in Dam and Winikoff (2003), Dam (2003) and Sudeikat et al. (2004). Other comparisons between agent-oriented methodologies include Shehory and Sturm (2001), Cernuzzi and Rossi (2002) and Sturm and Shehory (2003).

In the area of debugging of agent systems, most work has been in the area of visualisation to present a graphical depiction of system behaviour to the programmer, so they can understand how the system and the agents are behaving and interacting. The focus is on the collection of information, usually agent messages, and the presentation to the user with filtering applied to the messages (Ndumu et al., 1999; Nwana et al., 1999; Liedekerke and Avouris, 1995). This approach does not provide rich enough debugging information and does not address the issue of knowing what information is necessary for debugging.

The result is that the programmer is presented with too much information and experiences information overload reducing the effectiveness of the visualisation technique. Liedekerke and Avouris (1995) tried to overcome this by using abstractions and omissions in the form of selective information hiding to try and regulate the amount of debugging information being presented to the user. However, it was found that it was still too difficult to get a clear picture of overall system behaviour.

Nwana et al. (1999) provide debugging tools based on multiple views of the computation. The intention is that by combining results from different views, the programmer will be better able to identify incorrect system behaviour. Providing different views does limit the information flow to some degree, however, it does not overcome the problem.

Other researchers have also used Petri nets for specifying agent communication (see e.g. (Cost et al., 1999; Nowostawski et al., 2001)). Nowostawski et al. (2001) use their Petri nets for conversation modelling to support the run-time execution of protocols. Some examples for converting from AUML protocols to an equivalent Petri net representation are provided yet there appears to be no systematic approach for converting or developing the protocols. More recent work by Ehrler and Cranefield (2004) proposes directly executing Agent UML diagrams by linking application specific source code to execution occurrences on the AUML protocols. This approach is in the early stages of development and requires significant infrastructure for conversation management. Many platforms do not support conversations and protocols at the level required and as such other techniques for understanding or debugging interactions are required.

Other approaches to debugging include the use of model checkers and declarative debuggers to automate the process of verifying that a given implementation matches a specification (Wooldridge et al., 2002). These techniques are typically limited to finite state systems and suffer from a state space explosion making verifying real systems difficult (Walton, 2004). Furthermore verifying the correctness is done using a logic specification of the problem, which is typically beyond the expertise of many developers.

References

A. Pokahr, L. Braubach, W. L., 2003. Jadex: Implementing a bdi-infrastructure for jade agents. EXP - In Search of Innovation (Special Issue on JADE) volume 3, Nr. 3, 76–85.

Bergenti, F., Gleizes, M.-P., Zambonelli, F. (Eds.), Jul. 2004. Methodologies and Software Engineering for Agent Systems. The Agent-Oriented Software Engineering Handbook. Kluwer Publishing, ISBN 1-4020-8057-3.

Boehm, B. W., 1981. Software engineering economics. Prentice Hall, Englewood Cliffs, NJ.

Bresciani, P., Giorgini, P., Giunchiglia, F., Mylopoulos, J., Perini, A., 2002. Tropos: An agent-oriented software development methodology. Tech. Rep. DIT-02-0015, University of Trento, Department of Information and Communication Technology.

Burrafato, P., Cossentino, M., May 2002. Designing a multi-agent solution for a bookstore with the PASSI methodology. Proceedings of the Fourth International Bi-Conference Workshop on Agent-Oriented Information Systems (AOIS-2002).

Busetta, P., Rönnquist, R., Hodgson, A., Lucas, A., 1998. JACK Intelligent Agents - Components for Intelligent Agents in Java. Tech. rep., Agent Oriented Software Pty. Ltd, Melbourne, Australia, available from http://www.agent-software.com.

Bussmann, S., Jennings, N. R., Wooldridge, M., 2004. Multiagent Systems for Manufacturing Control. Springer-Verlag.

Cernuzzi, L., Rossi, G., November 2002. On the evaluation of agent oriented modeling methods. In: Proceedings of the OOPSLA 2002 Workshop on Agent-Oriented Methodologies. Seattle, pp. 21–30.

Cossentino, M., Potts, C., Jun. 2002. A CASE tool supported methodology for the design of multi-agent systems. Proceedings of the International Conference on Software Engineering Research and Practice (SERP'02).

Cossentino, M., Sabatucci, L., Sorace, S., Chella, A., October 2003. Patterns of reuse in the passi methodology.

Cost, R. S., Chen, Y., Finin, T., Labrou, Y., Peng, Y., 1999. Using colored petri nets for conversation modeling. Workshop on Agent Communication Languages at the Sixteenth International Joint Conference on Artificial Intelligence (IJCAI-99).

Dam, K. H., Jun. 2003. Evaluating agent-oriented software engineering methodologies. Master's thesis, School of Computer Science and Information Technology, RMIT University, Melbourne, Australia, (supervisors: Michael Winikoff and Lin Padgham).

Dam, K. H., Winikoff, M., Jul. 2003. Comparing agent-oriented methodologies. In: Giorgini, P., Winikoff, M. (Eds.), Proceedings of the Fifth International Bi-Conference Workshop on Agent-Oriented Information Systems. Melbourne, Australia, pp. 52–59.

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

Ehrler, L., Cranefield, S., 2004. Executing agent uml diagrams. Proceedings of the Third International Joint Conference on Autonomous Agents and Multi Agent Systems (AAMAS'04).

Fowler, M., Scott, K., Sep. 2003. UML Distilled: A Brief Guide to the Standard Object Modeling Language (third edition). Object Technology Series. Addison-Wesley.

Giunchiglia, F., Mylopoulos, J., Perini, A., Jul. 2002. The Tropos software development methodology: Processes, models and diagrams. Third International Workshop on Agent-Oriented Software Engineering.

Hailpern, B., Santhanam, P., 2002. Software debugging, testing, and verification. In: IBM Systems Journal. Vol. 41. pp. 4–12.

Heselius, J., 2002. Debugging parallel systems: A state of the art report. MRTC Report ISSN 1404-3041 ISRN MDH-MRTC-63/2002-1-SE, Mlardalen Real-Time Research Centre, Mlardalen University.

Huber, M. J., May 1999. JAM: A BDI-theoretic mobile agent architecture. In: Proceedings of the Third International Conference on Autonomous Agents (Agents'99). pp. 236–243.

Huget, M.-P., Odell, J., 2004. Representing agent interaction protocols with agent uml. Proceedings of the AAMAS04 Agent-oriented software engineering (AOSE) workshop.

Liedekerke, M., Avouris, N., 1995. Debugging multi-agent systems. Information and Software Technology 37(2), 103–112.

Müllerburg, M. A., 1983. The role of debugging within software engineering environments. In: Proceedings of the ACM SIGSOFT/SIGPLAN software engineering symposium on High-level debugging. pp. 81–90.

Ndumu, D., Nwana, H., Lee, L., Collins, J., 1999. Visualising and debugging distributed multi-agent systems. Proceedings of the Third Annual Conference on Autonomous Agents, 326–333.

Nowostawski, M., Purvis, M., Cranefield, S., 2001. A layered approach for modelling agent conversations. In: Proceedings of the 2nd International Workshop on Infrastructure for Agents, MAS, and Scalable MAS, 5th International Conference on Autonomous Agents, Montreal. pp. 163–170.

Nwana, H. S., Ndumu, D. T., Lee, L. C., Collis, J. C., 1999. ZEUS: a toolkit and approach for building distributed multi-agent systems. In: Proceedings of the Third International Conference on Autonomous Agents (Agents'99). ACM Press, Seattle, WA, USA, pp. 360–361.

Padgham, L., Winikoff, M., 2004. Developing Intelligent Agent Systems: A practical guide. Wiley Series in Agent Technology. John Wiley and Sons.

Poutakidis, D., Padgham, L., Winikoff, M., 2002. Debugging multi-agent systems using design artifacts: The case of interaction protocols. Proceedings of the First International Joint Conference on Autonomous Agents and Multi Agent Systems (AAMAS'02).

Poutakidis, D., Padgham, L., Winikoff, M., Oct. 2003. An exploration of bugs and debugging in multi-agent systems. In: Proceedings of the 14th International Symposium on Methodologies for Intelligent Systems (ISMIS). Maebashi City, Japan, pp. 628–632.

Reisig, W., 1985. Petri Nets: An Introduction. EATCS Monographs on Theoretical Computer Science. Springer-Verlag.

Russell, S., Norvig, P., 1995. Artificial Intelligence: A Modern Approach. Prentice Hall.

Shehory, O., Sturm, A., May 2001. Evaluation of modeling techniques for

agent-based systems. In: Proceedings of the Fifth International Conference on Autonomous Agents. ACM Press, pp. 624–631.

Sturm, A., Shehory, O., Jul. 2003. A framework for evaluating agent-oriented methodologies. In: Giorgini, P., Winikoff, M. (Eds.), Proceedings of the Fifth International Bi-Conference Workshop on Agent-Oriented Information Systems. Melbourne, Australia, pp. 60–67.

Sudeikat, J., Braubach, L., Pokahr, A., Lamersdorf, W., Jul. 2004. Evaluation of agent-oriented software methodologies: Examination of the gap between modeling and platform. Agent Oriented Software Engineering (AOSE).

Sycara, K., Paolucci, M., van Velsen, M., Giampapa, J., July 2003. The RETSINA MAS infrastructure. Journal of Autonomous Agents and Multi-agent Systems 7 (1 and 2).

van Lamsweerde, A., Aug. 2001. Goal-oriented requirements engineering: A guided tour. In: Proceedings of the 5th IEEE International Symposium on Requirements Engineering (RE'01). Toronto, pp. 249–263.

Walton, C. D., 2004. Model checking agent dialogues. Proceedings of the AAMAS04 Declarative Agent Languages and Technologies (DALT) workshop.

Wooldridge, M., Fisher, M., Huget, M.-P., Parsons, S., 2002. Model checking multi-agent systems with mable. In: Proceedings of the first international joint conference on Autonomous agents and multiagent systems. ACM Press, pp. 952–959.

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

Zelkowitz, M. V., 1978. Perspectives on software engineering. ACM Computing Surveys,10(2), 197–216.