Debugging Multi-Agent Systems With Design Documents

A thesis submitted for the degree of
Doctor of Philosophy

David Poutakidis
School Of Computer Science and Information Technology
Science, Engineering and Technology Portfolio
RMIT University
Melbourne, Victoria, Australia.

March 2008
Declaration

I certify that except where due acknowledgement has been made, the work is that of the author alone; the work has not been submitted previously, in whole or in part, to qualify for any other academic award; the content of the thesis is the result of work which has been carried out since the official commencement date of the approved research program; and, any editorial work, paid or unpaid, carried out by a third party is acknowledged.

Preliminary versions of some results or discussions in this thesis have been previously published:


David Poutakidis
School of Computer Science and Information Technology
RMIT University
March 29, 2008
**Acknowledgements**

I am fortunate to have had the support and encouragement of two excellent supervisors, Professor Lin Padgham and Associate Professor Michael Winikoff. As supervisors they complement each other very well and both have been instrumental in my development as a researcher. I wish to offer my thanks to them for everything and am truly grateful to have worked with them.

I would also like to thank my fellow research students, past and present, as well as the other members of the Intelligent Systems Group. I have seen our group grow and flourish and am proud to be a member. A special thanks to John Thangarajah, a great research companion that has been with me from the beginning.

To those who volunteered their time and energy to participate in the experiment used for the evaluation of my work, a big thank you.

Finally, I wish to thank my friends and family who were always supportive and had faith that I would do well. It has been tough, but also very rewarding. I’m fortunate to have had so much support. Thanks to all.
# Contents

<table>
<thead>
<tr>
<th>Summary</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 Introduction</td>
<td>3</td>
</tr>
<tr>
<td>2 Background</td>
<td>9</td>
</tr>
<tr>
<td>2.1 Agents</td>
<td>9</td>
</tr>
<tr>
<td>2.1.1 Plans and Events</td>
<td>11</td>
</tr>
<tr>
<td>2.2 Communication</td>
<td>12</td>
</tr>
<tr>
<td>2.2.1 AUML</td>
<td>14</td>
</tr>
<tr>
<td>2.2.2 Petri Nets</td>
<td>15</td>
</tr>
<tr>
<td>2.3 Agent Oriented Software Engineering</td>
<td>19</td>
</tr>
<tr>
<td>2.3.1 Gaia</td>
<td>19</td>
</tr>
<tr>
<td>2.3.2 Prometheus</td>
<td>21</td>
</tr>
<tr>
<td>2.3.3 Tropos</td>
<td>22</td>
</tr>
<tr>
<td>2.4 Software Debugging</td>
<td>24</td>
</tr>
<tr>
<td>2.4.1 An Introduction to Debugging</td>
<td>25</td>
</tr>
<tr>
<td>2.4.2 Informational Needs for Debugging</td>
<td>26</td>
</tr>
<tr>
<td>2.4.3 Debugging Tools and Techniques</td>
<td>27</td>
</tr>
<tr>
<td>2.4.4 The Role of Understanding in Debugging</td>
<td>29</td>
</tr>
<tr>
<td>2.5 Monitoring and Debugging Multi-Agent Systems</td>
<td>32</td>
</tr>
<tr>
<td>2.5.1 Debugging in Agent Development Environments</td>
<td>33</td>
</tr>
<tr>
<td>2.5.2 Other Agent Developer Environments</td>
<td>39</td>
</tr>
<tr>
<td>2.5.3 Platform Independent Tracing and Logging</td>
<td>39</td>
</tr>
<tr>
<td>Section</td>
<td>Title</td>
</tr>
<tr>
<td>---------</td>
<td>--------------------------------------------</td>
</tr>
<tr>
<td>2.5.4</td>
<td>Limitations of Current Systems</td>
</tr>
<tr>
<td>2.5.5</td>
<td>Approaches to Monitoring Agents</td>
</tr>
<tr>
<td>2.6</td>
<td>Summary of Related Work</td>
</tr>
<tr>
<td>3</td>
<td>Debugging Framework</td>
</tr>
<tr>
<td>3.1</td>
<td>Focusing Debugging to the Agent Paradigm</td>
</tr>
<tr>
<td>3.2</td>
<td>System Design Artifacts as Debugging Components</td>
</tr>
<tr>
<td>3.3</td>
<td>Architecture of the Debugging Framework</td>
</tr>
<tr>
<td>3.4</td>
<td>Analysis and Classification of Bugs in MAS</td>
</tr>
<tr>
<td>3.5</td>
<td>Interaction Related Bugs</td>
</tr>
<tr>
<td>3.6</td>
<td>Plan Selection Bugs</td>
</tr>
<tr>
<td>3.7</td>
<td>Debugging Plan Related Bugs</td>
</tr>
<tr>
<td>3.7.1</td>
<td>Overlap</td>
</tr>
<tr>
<td>3.7.2</td>
<td>Coverage</td>
</tr>
<tr>
<td>3.7.3</td>
<td>Reporting to the User</td>
</tr>
<tr>
<td>4</td>
<td>Translating AUML Protocols to Petri Nets</td>
</tr>
<tr>
<td>4.1</td>
<td>Converting AUML Components to Petri Net Fragments</td>
</tr>
<tr>
<td>4.2</td>
<td>Translations for the Interaction Operators</td>
</tr>
<tr>
<td>4.2.1</td>
<td>Overview</td>
</tr>
<tr>
<td>4.2.2</td>
<td>Alternative</td>
</tr>
<tr>
<td>4.2.3</td>
<td>Parallelism</td>
</tr>
<tr>
<td>4.2.4</td>
<td>Optional</td>
</tr>
<tr>
<td>4.2.5</td>
<td>Loops</td>
</tr>
<tr>
<td>4.2.6</td>
<td>Continuation</td>
</tr>
<tr>
<td>4.2.7</td>
<td>Termination</td>
</tr>
<tr>
<td>4.3</td>
<td>Problematic AUML Protocol Specifications</td>
</tr>
<tr>
<td>4.3.1</td>
<td>Optional</td>
</tr>
<tr>
<td>4.3.2</td>
<td>Loop</td>
</tr>
<tr>
<td>4.3.3</td>
<td>Parallel</td>
</tr>
<tr>
<td>4.4</td>
<td>Interleaved Protocols</td>
</tr>
</tbody>
</table>
## CONTENTS

6.2.2 Locating the Cause of the Bug ............................................. 129
6.2.3 Fixing the Bug ................................................................. 133
6.2.4 Within Subject Analysis ..................................................... 135
6.2.5 Discussion ................................................................. 137

7 Conclusion and Future Work .................................................. 141
7.1 Proof of Concept of Debugging Approach ................................. 142
7.2 Translation of AUML Protocols to Petri Nets ........................... 143
7.3 A Domain Independent Framework for Debugging ..................... 143
7.4 Future Work ................................................................. 144

A Pre-experiment Survey ......................................................... 145

B Instructions for experiment .................................................... 149

C Bug Data Collection Form .................................................... 157
List of Figures

2.1 AUML Propose Interaction Protocol ........................................ 14
2.2 Example of a Petri Net firing .................................................. 15

3.1 Agent Acquaintance Model .................................................... 50
3.2 Debugging Framework .......................................................... 51

4.1 State labels applied to AUML protocol. ..................................... 66
4.2 FIPA Propose Protocol with protocol states marked ..................... 67
4.3 Simple translation of a message to the equivalent Petri Net structure. .... 69
4.4 Alternative interaction operator .............................................. 72
4.5 Parallel interaction operator ............................................... 73
4.6 Merge after leaving a parallel region ...................................... 75
4.7 Optional operator affecting labeling process. ........................... 76
4.8 Merged Petri Net fragments. ................................................ 76
4.9 Loop interaction operator .................................................... 77
4.10 Merged Petri Net fragments .................................................. 77
4.11 Alternative inside a loop with the Petri Net fragments after translation. .... 79
4.12 Alternative inside a loop interaction fragment. .......................... 79
4.13 Continuation interaction operator. ........................................ 80
4.14 Termination Interaction Operator ........................................... 81
4.15 Problematic use of the alternative operator with message x appearing twice. .... 82
4.16 Modified version of figure 4.15 which can now be translated. ........... 82
4.17 Problematic use of the optional operator with message a appearing twice. .... 83
4.18 Modified version of figure 4.17 which can now be translated. 83
4.19 Problematic use of the loop operator with message a appearing twice. 84
4.20 Modified version of figure 4.19 which can now be translated. 84
4.21 Problematic use of the parallel operator 85
4.22 Example of an interleaved protocol 86
4.23 AUML Payment protocol and Petri Net equivalent 87
4.24 Combined version of the Request Information protocol 87
4.25 Alternative interleaved protocol 88
4.26 Alternative Request Information Petri Net protocol 89
4.27 Request When FIPA protocol 90
4.28 Petri Net version of the FIPA Request When protocol 91
4.29 AUML connectors combined 92
4.30 Petri Net equivalent of the combined AUML protocol 94

5.1 Overview of the Interaction Monitor. 96
5.2 Modeling a conversation on the FIPA request protocol. 103
5.3 The debugging toolkit with a single message received. 110
5.4 Debugging toolkit with two conversations active. 111
5.5 Message received that caused the conversation to be in error (Case A) 112
5.6 A warning message is displayed and a warning icon is added to the conversation tab. 113
5.7 A conversation that has successfully completed. 114

6.1 Abstract System Overview, 3 users. 122
6.2 Identification success 127
6.3 Identified timing data 128
6.4 Location success 130
6.5 Locating timing data 131
6.6 Providing fix success 133
6.7 Providing fix timing data 135
6.8 Difference in time to locate the bugs 137
6.9 Difference in time to provide a suitable fix for the bugs 138
LIST OF FIGURES ix

6.10 Summary data ......................................................... 139
A.1 Pre-experiment Survey (Page 1) ................................. 146
A.2 Pre-experiment Survey (Page 2) ................................. 147
A.3 Pre-experiment Survey (Page 3) ................................. 148
B.1 Instructions for Completing Experiment (Page 1) .......... 150
B.2 Instructions for Completing Experiment (Page 2) .......... 151
B.3 Instructions for Completing Experiment (Page 3) .......... 152
B.4 Instructions for Completing Experiment (Page 4) .......... 153
B.5 Instructions for Completing Experiment (Page 5) .......... 154
B.6 Instructions for Completing Experiment (Page 6) .......... 155
B.7 Instructions for Completing Experiment (Page 7) .......... 156
C.1 Data Collection Form ............................................... 158
Summary

Debugging multi-agent systems, which are concurrent, distributed, and consist of complex components is difficult, yet crucial. The development of these complex systems is supported by agent-oriented software engineering methodologies which utilise agents as the central design metaphor. The systems that are developed are inherently complex since the components of these systems may interact in flexible and sophisticated ways and traditional debugging techniques are not appropriate. Despite this, very little effort has been applied to developing appropriate debugging tools and techniques. Debugging multi-agent systems without good debugging tools is highly impractical and without suitable debugging support developing and maintaining multi-agent systems will be more difficult than it need be.

In this thesis we propose that the debugging process can be supported by following an agent-oriented design methodology, and then using the developed design artifacts in the debugging phase. We propose a domain independent debugging framework which comprises the developed processes and components that are necessary in using design artifacts as debugging artifacts. Our approach is to take a non-formal design artifact, such as an AUML protocol design, and encode it in a machine interpretable manner such that the design can be used as a model of correct system behaviour. These models are used by a run-time debugging system to compare observed behaviour against specified behaviour. We provide details for transforming two design artifact types into equivalent debugging artifacts and show how these can be used to detect bugs.

During a debugging episode in which a bug has been identified our debugging approach can provide detailed information about the possible reason for the bug occurring. To determine if this information was useful in helping to debug programs we undertook a thorough empirical study and identified that use of the debugging tool translated to an improvement in debugging performance. We conclude that the debugging techniques developed in this thesis provide effective debugging support.
for multi-agent systems and by having an extensible framework new design artifacts can be explored and as translations are developed they can be added to the debugging system.
Chapter 1

Introduction

Agents are seen as a promising technology for dealing with increasingly complex system development. The complex interactions that occur in many systems are naturally modeled with agents. To support the development of agent systems a new field of software engineering, commonly referred to as agent-oriented software engineering, has emerged, in which the agent is proposed as the central design metaphor. Debugging is a central and time consuming part of any software engineering process, and debugging of complex distributed systems, which agent systems typically are, is especially difficult. This thesis is about using the design documents for an agent system to assist in the provision of effective debugging support.

Agents provide a flexible and robust approach to task achievement making them ideal for deployment in challenging environments. Agents can be equipped with multiple ways of achieving tasks, and depending on the task and the context in which the task should be completed, can select the most appropriate way for dealing with it. The autonomous and distributed nature of agent systems, while modular and powerful is notoriously difficult to test and debug (Flater 2001).

Many people use visualisation to assist with debugging through presentation of complex system behaviour, for example Nwana, Ndumu, Lee & Collis (1999) and Jin, Maheswaran, Sanchez & Szekely (2007). Although useful this is primarily a user directed approach in which the user needs to manually review the visual data presented. An issue that arises with this approach is information overload. Our work proposes a principle of filtering the presentation of system behaviour using the design documents as a specification of intended behaviour, and indicating any diversion from that. This approach transfers the reviewing of raw system data from the user to a debugging agent which
is able to monitor multiple streams of data simultaneously.

An important class of errors that has received considerable attention are those concerned with agent interactions. To facilitate effective interaction between agents interaction protocols are typically used. Design of interaction protocols is concerned with specifying standard patterns of interaction for accomplishing specific tasks. These protocols consist of the set of messages that can be sent and the conditions under which these messages can be sent. The major challenge with debugging multi-agent system is ensuring that these interaction protocols are correctly followed. Debugging interactions is difficult partly because agents can participate in multiple interactions in parallel, resulting in inter-leaving of messages that can be difficult to follow as an observer. Observation is typically facilitated by logging and visualisation of the messages exchanged between the agents. The volume of messages makes it difficult to comprehend, and even with careful filtering applied it is still difficult to determine what is happening (Liedekerke & Avouris 1995).

In addition to the visualisation approach many debugging frameworks provide more traditional debugging support, for example Ndumu, Nwana, Lee & Collis (1999), Busetta, Rönquist, Hodgson & Lucas (1998) and Pokahr, Braubach & Lamersdorf (2003). Techniques such as breakpoints, stepping through code, an ability to display agent specific properties, such as goals and tasks, are provided. However, as with monitoring the visualisation this is a manual process, and when dealing with multiple, concurrently executing agents determining where to focus attention is very difficult.

During debugging the aim is to reconcile any differences between the actual program behaviour and the expected behaviour in order to uncover and resolve bugs. Current debugging techniques fail to take advantage of the underlying design of systems to support the debugging task. This problem is best summed up by Hailpern & Santhanam (2002):

> There is a clear need for a stronger (automatic) link between the software design (what the code is intended to do) . . . and test execution (what is actually tested) in order to minimize the difficulty in identifying the offending code. . .

Effective debugging requires more than providing a visual representation of elements of a multi-agent system. Information needs to be collected and presented in terms of the concepts used to design and model the system. In order to identify where problems are occurring we require extra information to guide us. We propose to use models developed during the design of the system to assist with this process. We develop tools and techniques to identify when execution diverges from what is specified
so the programmer is alerted to a possible problem that they may then investigate further. We provide a general framework for realising this approach and develop in detail the necessary steps for adding this debugging support to the development of multi-agent systems.

Our approach leverages the growing body of work which formalises agent design practices into agent based methodologies, for example Padgham & Winikoff (2004), Juan, Pearce & Sterling (2002), DeLoach (2001), Wooldridge, Jennings & Kinny (2000) and Fuxman, Pistore, Mylopoulos & Traverso (2001). These agent-oriented methodologies that have emerged define processes and design artifacts to guide system development. In using design artifacts to identify where an executing system diverges from the design model we must develop mechanisms for tracking important aspects of the various models. A significant part of this thesis is concerned with transforming AUML interaction protocols to a representation using Petri Nets which facilitates easy tracking with respect to the model during execution. In addition to the protocol transformations we also develop mechanisms to use plan and event information to aid debugging.

Reducing the complexity of debugging multi-agent systems reduces development time and should result in more robust systems being built. Furthermore, combining the debugging process with other stages of software development provides a more complete and usable agent-oriented paradigm. A debugging toolkit that focuses on the models created during design can be expected to improve the debugging process and be beneficial to developers working with multi-agent systems.

Our central thesis is that the design documents and system models developed when following an agent based software engineering methodology will be valuable resources during the debugging process and should facilitate the automatic or semi-automatic detection of errors. To explore this we propose the following research questions.

What types of bugs commonly occur in multi-agent systems?

The paradigm used to develop a system affects the way the system is conceptualised and introduces specific design and programming constructs that cause paradigm specific bugs. This question is concerned with identifying and cataloging bugs that are common in multi-agent systems so we can propose methods and develop tools that aid the programmer with the detection of these common bugs.
Can design artifacts be effectively used to support debugging, and if so what needs to be done to convert a design artifact to a debugging artifact?

Our aim is to explore the use of design models as debugging artifacts and to identify how they relate to the common problems identified, and how they can be used for automatic detection of incorrect system execution. Specific models that we investigate are interaction protocol models specified using AUML (Odell, Parunak & Bauer 2001, Low 1999). We also use information models particular to the Prometheus methodology (Padgham & Winikoff 2004) for identifying bugs in plan selection. A significant part of this work is in identifying how we can transform a design model into a debugging model that supports efficient comparison of real-time data against the model of correct behaviour.

How can design based debugging be integrated into a domain independent framework for debugging?

To support the use of design artifacts in the debugging of multi-agent systems a framework for developing debugging artifacts and incorporating debugging into the development of systems is proposed. We consider the issues of providing an architecture that is not tied to any specific implementation platform and endeavor to develop simple interface mechanisms to support a variety of platforms.

Consideration is given to how the debugging system should be incorporated into the run-time system. The debugging system comprises a library of debugging artifacts, a method for receiving run-time system information and then comparing it to the artifacts in the library, and a reporting component that provides the user with information about program errors when they are identified. We focus on information that should be readily extracted from most agent implementations, such as messages sent and event and plan executions, and provide an interface to the monitoring structures so that different implementation platforms can be supported.

Does the support provided by the methods proposed in this thesis translate to an improvement in debugging performance?

The debugging techniques proposed in this thesis are expected to help uncover errors in an executing multi-agent system and present them to the user in concepts closely aligned to the agent paradigm. We seek to explore through experimental evaluation whether our approach does in fact assist users in debugging multi-agent systems. To facilitate our evaluation we develop a debugging tool that
implements the debugging techniques proposed in this thesis. The debugging tool is a functioning prototype debugger that can be inserted into the run-time environment of an agent system to verify and further explore the capabilities and limitations of our proposed debugging framework.

**Summary of contributions**

This thesis contributes to the state of the art in debugging multi-agent systems with the introduction of design based debugging which supports the automatic detection of errors accompanied by detailed error reports framed with the concepts used to develop the system. We identify and present two classes of bugs that are characteristic of bugs that are commonly encountered during the development of multi-agent systems; those that relate to the selection of plans in response to the adoption of goals and those that relate to the interactions between agents. We develop debugging support for these bug types based on the design documents used to specify behaviour.

To help identify bugs related to the selection of plans in response to the adoption of goals we identify specific properties from the event descriptor design artifact and provide a process for extracting these properties, so at run-time we can determine if the observed execution matches the specified execution. If an erroneous execution is detected the debugging tool is capable of describing the problem accurately. Debugging agent interactions is an important and difficult task that we spend significant time on in this thesis. The AUML specification allows for complex interactions to be specified and there are numerous design and control structures that need to be addressed. We have developed a set of general purpose translation rules for converting AUML protocols into a suitable representation for debugging. Petri Nets are used to represent the AUML protocols internally to the debugger.

We have also developed a general approach for providing support to detect and report about these errors and have conceptualised this in a domain independent debugging framework. This framework provides the details for introducing the debugging support into an agent system as well as the processes that need to be followed to generate debugging artifacts from design artifacts. We have demonstrated this framework and the debugging techniques by developing a prototype debugging tool.

Evaluation via a user study using our prototype debugging tool showed that our approach to debugging multi-agent systems was helpful in isolating and resolving the cause of problems.
Thesis structure

The remainder of this thesis is organised as follows. In chapter 2 we present our review of the literature relevant to the debugging of multi-agent systems. We begin with a discussion of the agent paradigm, in particular its use as a software engineering design metaphor. Following this we describe some key background on debugging approaches and then focus attention to the specific details of debugging multi-agent systems. We begin chapter 3 with a discussion on common multi-agent bugs and then propose a framework for debugging multi-agent systems. We conclude this chapter with a specific example of converting a plan descriptor design artifact to a debugging artifact that can detect violations of plan selection rules. Chapter 4 is devoted to the process of converting AUML specifications to machine interpretable Petri Nets. We provide rules for converting a number of important AUML interaction operators and also discuss some problematic structures that require special attention when the conversion process is applied. In chapter 5 we describe the monitoring and reporting component of our debugging framework. A prototype debugging interface is presented and we give examples of converting and debugging interaction protocols. Chapter 6 describes the experiments undertaken to assess the efficacy of the debugging approach taken. The experimental plan and the results obtained are described and analysed. We conclude with a summary of the thesis and some potential areas for further development in the area of multi-agent debugging.
Chapter 2

Background

This chapter presents relevant literature in the area of agent-oriented development and debugging of agent systems. Section 2.1 begins the review with a brief introduction to the concepts and characteristics of agents. Given that we focus on multi-agent systems there is a detailed overview of how agents interact in section 2.2. Agents are used as a design metaphor for the development of complex systems and we discuss a number of agent oriented software engineering methodologies in section 2.3. Section 2.4 introduces the concepts of debugging as a step in developing agent software systems. Approaches to debugging are reviewed and particular attention is given to the fact that different development paradigms require different debugging tools, approaches and information. This leads to a detailed review of debugging multi-agent systems as presented in section 2.5. We complete the chapter with a brief summary of closely related work.

2.1 Agents

An agent can be described as a software entity that operates independently within an environment by sensing and reacting to the environment for the purpose of achieving some desired state of affairs. Agents are used to model some kind of entity within an environment and encompass the knowledge and ability required to operate in that environment. How this encoding of knowledge and ability is achieved is the subject of many different styles of agency. Brooks’ subsumption architecture (Brooks 1986) focuses on autonomous mobile robots that do not require any explicit representation of the environment, or indeed any mental attitudes whatsoever. Instead, agents are composed of a hierarchy
of task achieving behaviours, with each behaviour specifying a complete cycle from perception to action without the need for any internal representation of the environment. Chapman and Agre’s model of agents (Chapman & Agre 1987) is based on the notion that most activity is spent in generic routines, such as avoiding obstacles, or focusing on stimuli, and is based on a reactive model to achieve this. No internal representation of the environment is modeled. Agents operate with a set of simple rules which dictate what action should be taken based on the current perceived events. Another example of a reactive agent system is Rosenschein and Kaelbling’s *situated automata* (Rosenschein 1985, Rosenschein & Kaelbling 1986).

Reactive agent systems have been shown to be suitable for certain types of environments in which behaviour emerges as a result of trial and error experimentation. The reliance on emergent behaviour makes engineering agents to perform specific tasks difficult (Wooldridge 2004, page 97). A common alternative is to develop agents with more explicit representations for their reasoning mechanisms. To achieve such an agent a number of characteristics have been proposed by Wooldridge & Jennings (1995).

- autonomy: agents operate without the direct intervention of humans or others, and have some kind of control over their actions and internal state (Castelfranchi 1995).
- social ability: agents interact with other agents (and possibly humans) via some kind of *agent-communication language* (Genesereth & Ketchpel 1994)
- reactivity: agents perceive their environment and respond in a timely fashion to changes that occur in it.
- pro-activeness: agents do not simply act in response to their environment, they are able to exhibit goal-directed behaviour by taking initiative.

It is this style of agency that we focus on in which agents have an explicit representation about the world they are situated in. A popular paradigm that formalises this style is the Belief-Desire-Intention (BDI) paradigm. The BDI model specifies the mental components of a rational deliberative agent and is based on the philosophical work of Bratman (1987, 1988). In this work Bratman explores the relationship between intention and the way intelligent agents undertake to plan and act. Some aspects of this work have been formalised by Rao and Georgeff into a theory of intention which considers the relationships between beliefs, goals and intentions and provides semantics and logics suitable
for a persistent rational agent. They also categorise a variety of rational agents based on different commitment strategies (Rao & Georgeff 1991).

BDI style systems have been used successfully to solve a range of interesting and important real world problems, such as air traffic control (Ljungberg & Lucas 1992), telecommunications networks (Tapia, Bajo, Corchado, Rodríguez & Manzano 2007), traffic management systems (Rossetti, Bordini, Bazzan & Bampi 2002), medical services (Chiu, Cheung & Leung 2005) and simulated combat (Tidhar, Heinz & Selvestrel 1998).

2.1.1 Plans and Events

Agents need to reason about the goals that they have adopted and must have some means of achieving them. Plans are an abstract specification of the means for achieving certain goals and represent the options available to the agent (Rao & Georgeff 1992). Plans represent beliefs about what sequences of actions achieve certain conditions and are considered as a subset of an agents beliefs (Bratman, Israel & Pollack 1988). The actual method of representing a plan differs among researchers, in fact the term *recipe* (Grosz & Kraus 1996) is often used in the place of plan. Even though small differences exist there is a basic structure that all plans have. A plan is normally specified with a *name*, *body* and an *invocation condition*. An invocation condition is the condition under which a plan should be considered. It is usually triggered by an event and specifies when the plan should be used. The invocation condition is also called a *trigger* (Busetta, Rönquist, Hodgson & Lucas 1998) or the *motivation* for adopting a plan (Jennings 1995).

The set of plans available to an agent are often stored in a plan library. This library forms the set of beliefs that an agent has about how it should respond to events. An important characteristic of developing agents is the ability to define multiple different ways for handling a given event. This is achieved by specifying the different ways for dealing with an event each in their own plan, and each with the same invocation condition. When an event occurs all plans with a matching invocation condition are considered potential candidates for handling the event. Which plan is chosen (plan selection) is typically determined by a further processing of contextual conditions and of plan priorities.

The objective of plan selection is to generate an applicable plan set containing those plans that are suitable for execution based on the current environmental conditions (the context). There can be multiple levels of this type of filtering and once all filtering has been applied the result will be a set
of plans that are each valid responses to the event. These valid plans are what we call the applicable plan set. From this applicable plan set a plan needs to be selected for execution. This can be done by providing plan priorities to each of the plans and selecting the plan with the highest priority, by meta level reasoning or by random selection. Whatever strategy is employed at most one plan will be selected from the applicable plan set for execution, although if a plan fails an additional plan from the applicable plan set may be executed.

The body of a plan specifies what steps the agent should take. By adopting a plan, the agent is committed to, if things go as expected, performing all the steps in the plan, and believes that the plan will result in achieving the goal that the plan is meant to satisfy. The plan body need not be fully specified at the time the agent selects the plan. As the plan unfolds and the environment changes the agent can reason about the details of how it will attempt a goal or sub-goal. This is known as partial plan specification (Grosz & Kraus 1996) and is important because it is often not possible for an agent to know all the atomic steps in the plan when committing to it, as details of the situation may change during execution.

Dynamic environments pose significant difficulties in that the environment often changes in such a way that the current executing plan may no longer be appropriate (Tambe 1996b). To cope with this, agents need some method of reasoning about when a plan should be dropped so a different course of action could be taken. One approach is to include a maintenance condition in the plan. Maintenance conditions identify facts that must remain true throughout execution of the plan. If at any stage during execution the maintenance condition is violated the plan must be terminated. The maintenance condition is continually monitored and provides the agent with a means to reevaluate the situation by allowing it to stop executing a plan that is deemed to be unachievable or no longer relevant and select another (Tambe 1996a).

### 2.2 Communication

The notion of a multi-agent system implies some form of communication to enable the agents to exchange information. Typically agent communication is achieved by the transmission of messages between agents. Thus it is necessary for messages, their types and contents, to be understood by the agents within the system.

In closed systems there is often no need to use formal communication languages as it is expected
that the agents would share an understanding of the knowledge of the system. However, in open systems, where agents are developed by different groups and there is no common internal representation of knowledge, a common interaction language is required, such as KQML (Finin, Fritzson, McKay & McEntire 1994) or the FIPA Agent Communication Language (FIPA 2002). Languages such as these are made up of two parts: an outer language and a content language. The outer language specifies details about the sender, receiver, language type and ontology being used. This part of the message is often referred to as the “envelope” since it does not directly encode the message contents. Message content is based on the specific language chosen, such as the Knowledge Interchange Format (KIF) or the FIPA Semantic Language (SL) and contains the actual content of the message being sent.

Message exchanges can be as simple as a request for information and response with the required information, but in many multi-agent systems more complex patterns of interaction are often required. Interaction protocols define more complex and flexible interactions between agents. An interaction protocol specifies a standard pattern of interaction between two or more agents and consists of the set of messages that can be sent and a structure that specifies allowable sequences for the messages. Negotiation (Aknine, Pinson & Shakun 2004), argumentation (Artikis, Sergot & Pitt 2007), auctioning (David, Azoulay-Schwartz & Kraus 2002), and task distribution, such as the contract net protocol (Smith 1980), are all examples of the sort of protocols that can be useful in a multi-agent system.

Interaction protocols can be defined in a number of ways: as state machines (Dignum & Sierra 1991, page 110), in which the states might express the concept of waiting for a message, and the transitions express the concept of sending a message (Sprinkle, van Buskirk & Karsai 2000); as statecharts backed by a program logic with formal semantics (Paurobally, Cunningham & Jennings 2004); as Petri Nets where Petri Net places specify protocol state and Petri Net transitions encode message types (Cost, Chen, Finin, Labrou & Peng 2000, Purvis, Cranefield, Nowostawski & Purvis 2004); as standard UML (Lind 2001), or more commonly with an extension to UML in the form of the Agent UML (AUML) notation (Bauer, Müller & Odell 2001). More flexible interaction protocols are purportedly possible by using landmarks, where instead of single states as in the case of state machines, a partially ordered set of states is specified with different actions resulting in the same state being reached (Kumar, Huber & Cohen 2002). Additionally, commitment machines can be used in which instead of specifying required messages to be sent based on the current state of the protocol, agents progress through interactions by acquiring, manipulating, fulfilling and discharging
commitments. In the process of addressing these commitments messages are sent if they are defined as achieving a certain commitment (Yolum & Singh 2002).

Of these approaches we give special attention to the AUML protocol specification and the Petri Net modeling technique as they are the subject of much of the work in chapter 4.

2.2.1 AUML

Agent UML (AUML\(^1\)) is a graphical notation for designing agent interactions. It provides notations at various levels of abstraction, for example, interaction overview diagrams and sequence diagrams (Odell, Parunak & Bauer 2001) and is based on UML 2.0. Sequence diagrams, or as they are more commonly referred to, AUML Interaction Protocols, express the exchange of messages between agent roles arranged in a time sequence (see figure 2.1 for an example protocol). The AUML specification has evolved out of an earlier version, with significant changes having been introduced. Of particular importance is the change in notation which allows for more structured and legible protocol diagrams.

An AUML protocol is drawn within a frame which encapsulates all the information for the protocol. Any artifacts that are positioned outside this frame are external to the protocol. The protocol itself consists of a lifeline for each of the agent roles (or agent classes), marked at the top of a vertically

\(^1\)http://www.auml.org
dashed lifeline. Messages are based on the idea of a FIPA ACL message and are placed horizontally between two agent roles. They are depicted by arrows with a message name attached to the arrow indicating the message type. Time is modeled as advancing down the page thus specifying a natural ordering of messages.

This sequential ordering can be constrained and modified by interaction fragments, marked by inner boxes with various AUML operator labels. Boxes can contain messages and can be divided into a number of regions defining parts of the sequence diagram to allow for control on what message paths should be taken based on the current state of the protocol. The different box types, or interaction operators, include alternative, parallel, option, break, and loop. An alternative fragment specifies that only one of the regions in the box is executed. For example, in the Propose protocol of figure 2.1, after the propose message is received the recipient must choose to either reply with a reject-proposal message or an accept-proposal message. Either message is valid, but both must not be sent. We leave the explanation of the semantics of the remaining interaction operators until chapter 4.

### 2.2.2 Petri Nets

Petri Nets are a popular way of representing protocols, either as developed protocols from inception or as a translated version of an AUML protocol. Given that in this thesis we address the issues of converting AUML protocol specification to Petri Net equivalents for the purpose of debugging we give a brief introduction to Petri Nets here.

Petri Nets are a model of procedures that support the flow of information, in particular the con-
current flow of information. A Petri net (named after Carl Adam Petri) consists of places (depicted graphically as circles) and transitions (depicted graphically as rectangles). Places and transitions are linked by arcs which indicate the relation between the elements in the net. This relation is called the flow-relation, and the flow-relation may only connect places to transitions and transitions to places (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, figure 2.2 shows a very simple Petri Net, the transition in this Petri Net is enabled because both state $P$ and state $A$ are marked. The transition fires by removing a token from state $A$ and from state $P$ and placing a token on state $Q$.

In this thesis we present most of our discussions on Petri Nets using this graphical notation. The underlying formal definition is also required for some later discussion so we present a basic definition here. The Petri Net here is partially specified as we do not show the specification for the marking of the nets.

(a) A triple $N = (S, T, F)$ is called a Petri Net iff

(i) $S$ (places) and $T$ (transitions) are disjoint sets

(ii) $F \subseteq (S \times T) \cup (T \times S)$ is a binary flow relation of $N$

(b) Let $N$ be a net

For $x \in S \cup T$,

- $\bullet x = \{y | yF x\}$ is called the preset of $x$
- $x\bullet = \{y | xF y\}$ is called the postset of $x$

Both places and transitions can have a preset and a postset. The preset of a place element is made up of all transition elements that feed into it. Similarly the preset of a transition element is the made up of all the place elements that feed into it. For the postset we consider the elements of an outgoing
nature. Within this definition there is a natural requirement that place elements can only connect to transition elements.

For example, the Petri Net of figure 2.2 would be defined as follows;

\[ S = \{ P, A, Q \} \]
\[ T = \{ T_1 \} \]
\[ F = \{ (P, T_1), (A, T_1), (T_1, Q) \} \]

In addition, the preset of \( P \), written as \( \bullet P \), is \( \emptyset \) and the postset of \( P \), written as \( P \bullet \), is \( \{ T_1 \} \). For transition \( T_1 \); \( \bullet T_1 = \{ P, A \} \) and \( T_1 \bullet = \{ Q \} \)

Petri Nets have been used to model various characteristics of agent systems in a number of interesting ways. Early work mapped functional components of agents into an explicit Petri Net model that specified the structure and behaviour of agents. The focus was on coordination activities within and between agents and on the analysis of these structures (Purvis & Cranefield 1996). Following this a number of other approaches appeared that use Petri Nets to specify properties of agent systems (Billington, Du & Farrington 1998, Cost, Chen, Finin, Labrou & Peng 1999, Fallah-Seghrouchni, Haddad & Mazouzi 1999), with some using Petri Nets as a means for both the specification and implementation of a system (Köhler, Moldt & Rölke 2001).

Specification of the interaction patterns between agents using Petri Nets is one area that has received most attention. Initially Petri Nets were used to specify interaction protocols and to model conversations (Cost, Chen, Finin, Labrou & Peng 2000). This approach utilised a variation of Petri Nets called coloured Petri Nets. The main addition being that the tokens in a coloured Petri Net can contain information and arcs can be marked with constraints such that they will only enable a transition if the information contained in the token satisfies the arc constraint. A protocol can be modeled by defining the states of the protocol as places. Transitions, and the relation between places and transitions define the logic of the protocol. This is a static view of the protocol but it also enables the modeling of the running protocol. The marking of the Petri Net (the placement of tokens) can be used to indicate the dynamic state of the protocol (the conversation). As a conversation progresses tokens are removed and generated on the appropriate place to indicate the protocol state.

One of the advantages of using Petri Nets is that the notation comprises few components, and yet
these simple components can be used to build complex interaction patterns. The drawback to this approach is that it is difficult to develop a methodology or provide instruction for building protocols from the ground up. As the more expressive, yet less exactly specified AUML modeling notation gained traction, some used this as a basis for developing their own Petri Net versions of the AUML protocols (Fallah-Seghrouchni, Haddad & Mazouzi 1999, Nowostawski, Purvis & Cranefield 2001, Cranefield, Purvis, Nowostawski & Hwang 2002). AUML protocols were viewed as needing more precise specification and Petri Nets were seen as a means to achieve this. AUML protocols were translated to Petri Net versions and were further developed to remove any perceived ambiguities in the specification of the protocol. The translations from AUML to Petri Net, however, appear to be ad-hoc with no generic process provided. Example translations were provided but the focus was not on translating from one specification to another.

Our early work addressed the issue of translation when we desired to utilise AUML as more than just the notation for specifying protocols. We proposed the use of AUML protocols converted to Petri Nets for the purpose of debugging agent interactions (Poutakidis, Padgham & Winikoff 2002). In addition to the infrastructure required to debug interactions we also provided a set of generic rules for converting a subset of the original AUML specification to a Petri Net representation.\(^2\)

This work was followed by Cabac (2003) who take a different approach to translating AUML protocols to Petri Nets, and also use the Petri Nets for a different purpose. This work is situated within the Mulan framework, one in which the Petri Nets are used for design and implementation of agents. Given that AUML protocols are not directly executable, a process for translating a subset of an earlier specification of AUML to Petri Nets suitable for execution in the Mulan system was developed. The converted protocols are then used within the Mulan system to support the interaction between agents. Although this work appeared after ours it does not build on it, and is likely that development occurred in parallel with our work. Another example of using AUML for directly executing protocols, although not using Petri Nets, is the work on PAUL (Plug-in for AUML Linking) (Ehrler & Cranefield 2004). PAUL allows users to quickly develop and implement agent conversations through the implementation of directly executable protocols. In this approach a meta-model is developed for the AUML protocol specification to enable computers to read AUML diagrams. However, it only handles a small sub-set of the AUML notation.

\(^2\)Shortly after this work was published a new version of AUML was being developed and we switched our attention to it. When we discuss our translations from AUML to Petri Nets we are talking about the new version of AUML.
Recent work by (Gutnik & Kaminka 2006) aims at representing conversations for the purpose of supporting overhearing mechanisms in the monitoring of multi-agent systems. Based on the initial AUML specification the authors provide rules for translating AUML interaction building blocks, and also for including other properties, such as guards, in the Petri Net version of the protocol. Although coloured Petri Nets are used some aspects of their work builds on ours. For example, they follow a similar algorithm for processing messages within the Petri Net protocols, and utilise a similar mechanism for composing the basic building blocks of the Petri Net structure.

2.3 Agent Oriented Software Engineering

The research into agent technologies has resulted in concepts, techniques and approaches for specifying intelligent agents. To apply these to a software engineering undertaking to practically design and develop such complex systems requires an appropriate design methodology. Agent Oriented Software Engineering is a term used to describe the engineering of software systems with agents as the central design metaphor. Over the last decade there has been an increase in interest in this area and an increase in the number of methodologies proposed (Bergenti, Gleizes & Zambonelli 2004, Henderson-Sellers & Giorgini 2005).

2.3.1 Gaia

The Gaia methodology was the first to address the development of multi-agent systems with a systematic approach for creating rational agent based systems (Wooldridge, Jennings & Kinny 2000), which has been developed and extended in a number of ways (Zambonelli, Jennings & Wooldridge 2003). Development should be viewed as organisational design with the system modeled as an organisational entity, including the rules and structure found in such systems. Abstract concepts such as roles, responsibilities, permissions and protocols are used to help frame the problem from this perspective. Requirements gathering and specification is seen as independent of the paradigm being used. Terminology and notation is borrowed from the Object Oriented FUSION method (Coleman, Arnold, Bodoff, Dollin, Gilchrist, Hayes & Jeremaes 1994) with the introduction of a set of agent-specific concepts to model and understand an agent based software system.

In supporting the view of the system from the perspective of an organisational entity there is a strong focus on the situatedness of the agents in the system. The environmental conditions of the sys-
tem are given considerable attention and should not be implicitly assumed. The characteristics of the environment are expected to be identified, modeled, and possibly shaped to meet application-specific purposes. The methodology begins with some abstract modeling of the system as an organisational entity with global behaviour identified and some loosely coupled sub-organisations identified. A computational representation of the environment in which the system is to be situated is modeled and some preliminary role types are considered. The basic methods by which the agents will interact are developed and modeled with a preliminary interaction model and the rules that the organization should follow in terms of the global behaviour are determined.

During further stages these initial models are revised and refined until all details are complete. At the lowest level a services model describing the functionality of the individual agents is developed. A service is similar to the notion of a plan and comprises a single, coherent block of activity in which an agent will be engaged. The services model therefore defines all of the services that a particular agent would provide. Concrete agent types are defined and it is possible to package a number of closely related and strongly interacting roles within a single agent entity. This decision should be made before finalising the interaction models where the specific details of the interactions between the agents and the environment are defined. One limitation of the interaction models are that they are informally described with few notational devices which may make it difficult to implement reliably. This issue has been identified elsewhere and some approaches to applying AUML to Gaia have been proposed (García-Ojeda, Arenas & de Jesús Pérez-Alcázar 2005, Cernuzzi & Zambonelli 2004).

The result of the design process is a number of detailed design models that prescribe how the system should be implemented. Implementation is not considered and it is argued that the models developed are detailed enough that the system could be implemented either with general purpose programming languages, or if desired, with an agent oriented programming language (Zambonelli, Jennings & Wooldridge 2003).

Gaia has undergone considerable development over the years and has been the foundation for various agent oriented software engineering extensions. For example, it has been extended to encompass features such as representations of knowledge, relationships and social structures (Juan, Pearce & Sterling 2002). A process for applying a Gaia design to the JADE agent programming language has been developed and presented with an example system (Moraitis, Petraki & Spanoudakis 2003).
2.3.2 Prometheus

Prometheus is an agent oriented software development methodology which covers all phases of development in the construction of intelligent agent systems (Padgham & Winikoff 2004). The methodology defines a detailed process for specifying the various properties of the system which lead to concrete, well structured, design artifacts which map directly to a number of important agent concepts. There are three design phases that follow on from one another although they are used iteratively as with most modern software engineering methodologies.

Design begins with the System Specification and is carried out in an interleaving iterative manner. The system actors are identified and modeled in the design along with actions and percepts. Actors can model humans or other software systems and represent the entities, roles or stakeholders that are external to the system, but interact with it in some way. The next step is the development of scenarios that describe how the actors will interact with the system. This process is similar to use case identification in Object Oriented analysis, however, scenarios are more comprehensive as they include a sequence of structured steps and indicate possible alternatives. The development of scenarios also results in the identification of high level system goals, with each scenario being coupled to a goal. Goal identification and refinement is an important part of this phase and further analysis can result in more goals being identified. Goals are refined by a process of abstraction and refinement and a goal hierarchy composed of goals and sub goals is developed. This goal hierarchy is used to group common parts together to help direct the identification of roles.

The second stage is the architectural design phase which refines the system specification with a focus on determining the agent types within the system along with how the agents will interact. The main steps in this phase involve deciding what agent types will be implemented and developing the agent descriptors for the agent, describing the dynamic behaviour of the system using interaction diagrams and interaction protocols, and capturing the system’s overall (static) structure using an overview diagram called the system overview diagram. Processes and suggestions for how to determine the agent types are provided. Which goals and roles will be handled by which agents is inherited from the roles defined in the system specification. As these choices are made scenarios can be revisited to determine where interaction is necessary. The result is a set of interaction protocols that describe how the agents will interact to achieve the system goals.

The final phase of design is the detailed design and is concerned with specifying the internal
structure of each of the agents with respect to how the system tasks will be achieved. Agents are refined such that the functionality of the system, expressed as event, plans and data structures, are specified in self contained entities known as capabilities. Each agent can be made up of a set of capabilities, as well as agent level plans and events. The internals of the agents and capabilities are depicted using agent and capability overview diagrams which are similar in nature to the system overview diagram. Events are defined with both a plain language explanation and some specific characteristics, such as plan coverage/overlap (see section 3.6 for further details). Plans, which define the recipe for action, are considered and fully specified. Protocols are revisited and the single agent part of each protocol is defined using process diagrams. Process diagrams describe the dynamic aspect of the protocol from the perspective of a single agent and include details of what actions and messages can be sent and the conditions and rules associated with them.

Prometheus is supported by the Prometheus Design Tool (PDT)\(^3\) (Padgham, Thangarajah & Winikoff 2005) which supports the specification and generation of the design artifacts used in the Prometheus methodology. Cross checking of design elements is supported and it is able to generate skeleton code to simplify the implementation phase. PDT currently supports the generation of JACK code, while Sudeikat et al have reported using the PDT design files to generate Jadex code (Sudeikat, Braubach, Pokahr & Lamersdorf 2004).

### 2.3.3 Tropos

Tropos (Castro, Kolp & Mylopoulos 2001, Bresciani, Giorgini, Giunchiglia, Mylopoulos & Perini 2004) is a popular methodology based on the concepts and notations of the \(i^*\) modeling framework (Yu 1997). Tropos provides a requirements driven methodology that focuses on the human-like, distributed behaviours of software entities and takes the software engineering process from early requirements through to implementation. A Tropos design begins with a model of the system-to-be and its environment, and is incrementally refined and extended throughout the five phases of development.

The first two phases of the Tropos methodology are concerned with capturing the requirements of a system and forms the basis for the following software development phases. Early requirements gathering is concerned with modeling the system as an organisational model with a focus on the

\(^3\)available from http://www.cs.rmit.edu.au.agents/pdt
actors and the goals of the system. Goals from the organisational model are expanded and refined via a means-end analysis to describe how the goals can be achieved. At this time new actors might be identified as necessary additions to the system. The dependencies between goals, and the actors that should achieve them are modeled. Both standard goals, those that appear to have an obvious way of being achieved, and soft goals, those that are “not well-defined” with no immediate idea of how they are to be realised in the system, should be considered.

Following this phase, one is required to consider both the functional and non-functional requirements of the system and should consider the dependencies between the goals and the actors that have been identified. This phase is called late requirements analysis and the focus is on considering the components of the system with how it will operate as a deployed application. Further sub-actors may be identified during this phase after analysing the system’s operational environment and human actors that will interface with the system can be modeled with associated dependencies.

The next two phases focus on defining the system architecture based on the requirements resulting from the requirements analysis modeling. The Architectural Design phase models the system’s global architecture taking into consideration sub-systems, external systems and their interconnections and dependencies. The actors are mapped to specific agent types with specific capabilities and the method for achieving the soft goals should be addressed. In addition, it is at this stage that the style of architecture is considered by understanding and comparing the different architectural settings in which the system can be deployed. The characteristics of the different types of architectural designs, such as thin web client, thick web client and web delivery are applied to the soft goals to identify the most suitable architectural choice.

In the Detailed Design phase agent behaviours, capabilities, and interactions are specified. Standard models from the Object Oriented community as well as models specific to agent development, such as AUML and plan diagrams are used. Given that the implementation platform has often already been selected it is possible to use this to develop a detailed design that will map directly to code. The Implementation Phase is supported by a mapping overview model which depicts how the concepts from *i* can be mapped to a BDI agent framework. This is demonstrated with specific mappings from the model to the concepts from the JACK Intelligent Agents Framework.

The Tropos methodology has been demonstrated in a case study (Giorgini, Perini, Mylopoulos, Giunchiglia & Bresciani 2001), and considers formal models for specifying and checking correctness
(Fuxman, Pistore, Mylopoulos & Traverso 2001). It also considers issues such as goal modeling (Giorgini, Mylopoulos, Nicchiarelli & Sebastiani 2003), security (Giorgini, Massacci, Mylopoulos & Zannone 2005) and design patterns (Giorgini, Kolp & Mylopoulos 2003).

2.4 Software Debugging

The process of developing software has evolved from small scale, single programmer environments to large scale team based environments where the programming component is just one of the parts of the entire software engineering task. Software engineering consists of requirements analysis and specification, software design and of course programming and verifying that the software behaves in accordance with specifications (Müllerburg 1983). This view mandates that software is more than just the resulting program, it includes the associated documentation required to develop, operate and maintain the program (Boehm 1976).

An important part of software engineering is concerned with ensuring that the developed program operates correctly with respect to the needs of the customer and with respect to the associated software design specification. This is commonly referred to as verification and validation of the software. The goal of this phase is to ensure that the developed software complies with expectations, and taking corrective action where necessary. Validation asks the question “Are we building the right product?”, whereas verification asks “Are we building the product correctly?” Both are important but in this thesis we are more interested in the second question.

Answering this latter question is characterised by the testing of the functionality of the program and the evaluation of the tests with respect to expected behaviour, as defined by the specification. The verification of a software systems begins with the testing of the system at a number of different stages: unit testing, testing individual components of the system sub-system testing, and integration testing. The testing typically ends with acceptance testing in which the client or end user is consulted to ensure the system meets operational expectations (Sommerville 2006).

The purpose of this testing phase is to determine if the program meets expectations, but it is also to uncover errors within the program so they can be fixed before the system is deployed. The identification of such errors and fixing them is what we commonly refer to as debugging. Debugging is seen as such an important phase of the software engineering process because it has been found to occupy a large portion of development time (Agrawal, DeMillo & Spafford 1991). Debugging is a
dynamic activity that involves the execution of a program compared against expected behaviour and is concerned with the identification, isolation and removal of errors from a program.

2.4.1 An Introduction to Debugging

Although there is some speculation as to where the term *bug* was first used (Cohen 1994, Johnson 1982) it is widely accepted that the term is used to describe a mistake, malfunction or error associated with a computer program. Most commonly we are able to identify that such a *bug* exists because some observed execution of a program (or observation of the recorded output of a program) does not conform with what is expected. From this we can define debugging in the following way: Debugging is the process of locating, analysing and correcting suspected errors (McDowell & Helmbold 1989).

Developing software is a complex task and this complexity gives rise to a surprisingly large number of different bug types. In the appendix of his software testing techniques book Beizer lists eight high level categories of bugs as part of a bug taxonomy and statistics report (Beizer 1990). These categories range from errors in the specification of functionalities and features with respect to the requirements of the system, to errors in the implemented program with respect to such things as control structures or data definitions and access, through to system integration errors and software architecture errors. Each of these categories contains several further sub classes of bugs with a total number of bug types in the hundreds!

Research into bug types, although few and far between, provides valuable information and supports the retention and dissemination of knowledge that can be helpful for other developers (Kajihara, Amamiya & Saya 1993). In addition to providing a common ground for developers to describe bugs, by identifying specific bug types or bug classes and educating developers accordingly, methods for resolving such bugs can be developed and propagated through the wider developer community. An important part of this work is to help resolve the problem of different developers describing the same bug with different terminology (Shooman & Bolsky 1975).

Debugging is predominantly a manual task and the first step is to identify or verify that a bug exists in the system. Given the definition of what a bug is this requires a reasonable level of understanding of both what the system is supposed to do and what it is observed to be doing. This is sometimes done with static code analysis, but more commonly from actual system runs. Developing understanding of the latter requires a level of understanding suitable to analyse a problem, locate
the cause of the problem and in most cases determine how it should be fixed (von Mayrhauser & Vans 1997). It has been proposed that as part of this effort a mental model of the program is constructed in the minds of the developer. During this construction the developer looks for clues that indicate common tasks and adds this information to their model. Interestingly switching between different levels of abstractions occurs throughout the process (Vessey 1985).

Abstractions are an important method for understanding complex processes. An example of an abstraction as cited by von Mayrhauser & Vans (1997) is that of the different views of the operations of an operating system. Initially one may view the operations in terms of the control flow between operating system modules. Moving to a lower level of abstraction we may consider the view of a scheduling function in terms of the doubly linked list that is used to store job information. This is a simple but important concept since different software engineering methodologies use vastly different abstraction models. From a debugging point of view this means that different software engineering methodologies may promote discussion of bugs with different terminology, but more importantly result in different types of bugs being introduced based on the methodology used (Tukiainen 2000).

This need to focus on a viewpoint that matches the paradigm is further evidenced by the myriad of debugging tools for any given paradigm; distributed debugging (Babaoğlu, Fromentin & Raynal 1995, Schwarz & Mattern 1994), parallel debugging (Heselius 2002, LeBlanc, Mellor-Crummey & Fowler 1990), Object Oriented debugging (Jo, Kim, Im, Paik & Lee 1997, Pauw, Helm, Kimelman & Vlissides 1993) and of course debuggers for multi-agent systems (Liedekerke & Avouris 1995, Nwana, Ndumu, Lee & Collis 1999).

2.4.2 Informational Needs for Debugging

In all of the different development paradigms available there are similar informational needs for debugging the resulting systems. In a review of automated debugging systems a summary of informational requirements for effective debugging is provided (Ducassé 1993). Understanding the intended I/O and comparing it against the actual I/O of a system supports the identification of bugs and does not require any modification to the system. The normal I/O activities are observed and compared against the specified values. The I/O knowledge is typically the first piece of information that the debugger has when starting to debug a program. At a more detailed level is that of the intended behaviour of the system, with respect to program control, data accesses and modification, and that of
the actual observed behaviour. This is again a comparison between specifications and observations, but the observation happens at a lower level of abstraction. Here the debugging is concerned with, for example, the interaction between components in the system. This is often supported by some kind of system trace in which a set of desired system events are recorded and or visualised.

Other important information is concerned with the specifics of the implementation, including such things as how coding standards are defined and used. The size of the program was also seen as important and it was argued that programs that could be completely reviewed (due to their small size) were more easily debugged than larger ones in which one could only digest small parts of a larger system. This latter point is a little outdated with the massive increase in program size and complexity that we deal with today. General programming expertise, and specific skill in the target language chosen as well as good knowledge of bugs and debugging strategies are also seen as vital. Furthermore, there is a clear correlation of debugging ability with programmer ability (Chmiel & Loui 2004, Ahmadzadeh, Elliman & Higgins 2005). Skilled programmers are faster and more accurate at the debugging task and in contrast to novice debuggers introduce fewer new bugs. The difference in skill level is attributed to the ability of expert programmers to develop better hypotheses about the bug as a result of superior ability in comprehending programs (Gugerty & Olson 1986).

2.4.3 Debugging Tools and Techniques

To aid the debugging process debugging tools have been developed to help identify, track down, and fix errors in software programs. Fault localisation, which is defined by Hall et al. as tracing a bug to its cause (Hall, Hammond & O’Donnell 1990), is seen by some as the most difficult part in debugging (Jones 2004, Ducassé 1993, Vessey 1985). Indeed, most of the debugging support provided by debugging tools focus on the process of localising a discovered fault. Such tools are typically tailored to a specific target programming language for which they have been designed. However, there are a number of features that one may come to expect from a debugging tool. Namely, program tracing, breakpointing, and variable or memory display and manipulation.

Program tracing allows one to follow the executable program as lines in the source code are executed. This can be useful for understanding the flow of control within a program. Although, in a large search space or when long iteration sequences are being followed this can become difficult. Breakpoints are a special instruction that can be inserted into a program such that the program will
halt when the instruction is reached. This is an efficient way of allowing a program to run to a specific
location and then halt to allow some other debugging activity to occur from that point, for example,
tracing from the breakpoint onwards, or inspecting the state of a variable and possibly changing it
before continuing execution.

These basic features can be augmented with more specialised debugging activities and research
into new and more effective debugging techniques has resulted in some interesting ideas. Program-
mers have been found to apply a method called slicing in which large programs are broken down into
smaller coherent and comprehensible pieces (Weiser 1984). The reason for the slicing is to better
enable the programmer to understand enough about a program to carry out the debugging task. When
traversing a program to locate the source of an error programmers might work backwards from the
point of failure (Gould 1975, Lukey 1980) or forwards, which is more likely if the program was au-
thored by the person performing the debugging task (Katz & Anderson 1988). During this time the
flow of control and variables that are integral to the flow of control for the current execution would be
considered. Importantly, tracing backwards from a specific variable and considering all statements
that can influence that variable reveals that many statements have no influence.

Programmers apply this technique as a necessity given the often large size of programs, but the
application is typically not rigorous or precise. Some important statements may be missed while
others that are not important might be unnecessarily inspected. To rectify this situation automatic
program slicing has been proposed (Weiser 1982). This approach formalises the slicing strategy such
that a sub program is generated based on a slicing criteria. A program is stripped of statements
that have no influence over some target variable leaving only those statements that affect the value
of the variable, resulting in a smaller and less complex program. An interesting enhancement to
the slicing approach that further reduces the number of statements that need to be considered is
called dicing (Lyle & Weiser 1987). Dicing involves first taking the slice of the program based on
the incorrect variable and then taking a second slice of variables that are known to have correct
results thus producing an even smaller slice. An interesting side effect of the slicing approach has
been suggested by Francel and Rugaber. They found that there is some evidence to suggest that by
applying the slicing approach programmers acquire a better understanding of the program than those
that do not apply the slicing approach (Francel & Rugaber 1999). Program slicing is still an active
research area with recent work investigating different aspects of slicing algorithms (Binkley, Gold &
2.4.4 The Role of Understanding in Debugging

For effective debugging sufficient understanding and comprehension of both the implemented system and the design that the system is based on are required. It is necessary to gain sufficient understanding of these two closely related parts of system development for the purposes of identifying and resolving behaviour that is not consistent with the design specification. Developing the necessary understanding of the implemented system can, to some degree, be accomplished by performing code walkthroughs, or more formally code inspections (Fagan 1986). Code inspections are incrementally applied to parts of the source code to develop the necessary understanding of the system to uncover code defects. The utility of this process has also been shown to be effective by Doolan (1992) and Madachy (1995). Observing the behaviour of the system as it executes is, however, still an extremely useful and common exercise that is employed by developers to obtain a more complete understanding of the behaviour of the implemented system.

Observation is an active process, it is “exploration, inquiry for the sake of discovering something previously hidden and unknown” (Dewey 1910). This is a fitting description of what is required of the initial phase of debugging: identifying the errors. As debuggers we are interested in identifying the hidden defects and errors in the program. By observing the system under execution we aim at initially discovering these hidden defects. Once discovered, the location and cause of the bug must be ascertained. In the process of discovering the cause of the bug one suggests and investigates a hypothesis to explain the cause (Vessey 1985). Observation is of prime importance. Dewey further remarks that the the purpose of observation is, “to locate the nature of a problem and thereby guide the formation of a hypothesis” (Dewey 1910).

It is clear that observation is vital to effective debugging, by observing the system we develop an understanding of the behaviour of the system and are then able to form a hypothesis to explain the problem. What is not clear is what it is that should be the focus of our observations. This is an issue for both the initial problem of identifying a bug and in trying to localise and fix the bug. We indicated previously that observation is an active process, it is therefore necessary that we make a conscious decision to focus on some part of that which can be observed. From an abundance of information one must decide what to focus attention on.
The issue is that the scope of things that can be observed is immense. At one extreme is the data available from traditional low level debugging techniques. For example, one could focus on a subset of interesting variable assignments, traces of method or function invocations, stack traces etc. Tools that provide this information are indeed indispensable. However, given the size and complexity of agent systems, using such techniques to help identify an otherwise unknown problem, or to determine where in the source code to focus attention is rarely practical.

The problem of using low level tools to help identify and guide the focus of observation is made more difficult due to the fact that often the developer doesn’t know what exactly they are looking for. During debugging a developer may propose many different hypotheses to help guide their search. However, interestingly, few will make specific efforts to prove the hypothesis directly. Instead they are exploring, keeping their eyes open for interesting occurrences that may provide more information about the problem (Vessey 1985). This process is described as a breadth first search for information. The hypothesis provides the basis for the search and whenever new information is encountered and deemed useful the search branches off into that direction. Utilising low level debugging techniques makes it more difficult to cover a wider area of the source code and makes it more difficult for the program to “unfold” before the developer.

An interesting approach to helping users understand the complex behaviours and interdependencies in applications is proposed in the Whyline framework where users are able to ask why or why not questions about observations they make while interacting with a system (Myers, Weitzman, Ko & Chau 2006). These questions, which are automatically derived, have answers generated using built-in techniques. Answers are derived following similar rules to those used for determining the relevant statements in a program slice. Questions are typically of the form, “why does property p of object o have value v?”. The Whyline system recursively traverses through the operations that cause properties to take on their values and results in a statement indicating the response to the question.

In a user study the Whyline approach was found to be very effective in improving understanding in computer programs. However, at present this approach can only be used if one implements their system using the underlying development framework, Crystal. The Crystal system takes care of storing the information that will be useful in generating and answering questions and provides the interface for allowing a user to select objects of interest. In addition, although questions and answers are automatically derived there are cases where, during development, the user must determine what
questions should be ignored when constructing the lists. This is stated as generally simple, but in complex programs it may prove difficult.

**Automatic and focused debugging**

Automation of the various parts of the software engineering process, such as automatic code generation or automatic test case generation are attractive because they have the potential to reduce the time taken to implement a system as well as reducing potential human error. Relieving the manual burden of debugging software programs by automating the debugging process has received much attention. Techniques such as program slicing, algorithmic debugging (Shapiro 1983) and model based diagnosis (Clarke, Grumberg & Long 1994, Mayer & Stumptner 2007, Yilmaz & Williams 2007) each provide support for automating, or partially automating the debugging process.

Model based diagnosis in particular has generated considerable interest in the diagnosis of faults in both hardware and software systems. This technique was initially devised for detecting hardware faults and has been formalised and generalised into a theory of diagnosis that can be applied to many different engineering areas (Reiter 1987). Model based diagnosis uses formal reasoning methods to uncover errors and automatically rectify them. From a hardware perspective the model based diagnosis approach utilises the system description (the model) which is based on a set of components and their connections. A behaviour model is derived from the system description and is encoded as a set of logical sentences based on the outputs of the system from certain input values. The derived behaviour is a correct model of the system and describes expected outputs from a set of inputs. The diagnosis problem uses this model and compares it against observations of the physical system. Discrepancy in the output of the observed system, against those of the model, are identified. Once identified the reason for the difference can be computed and corrections to the system can be suggested.

Model based diagnosis of software programs is situated within the testing phase of development. Test cases are defined describing correct behaviour which can then be checked against the model of the implemented program. The definition of test cases is the subject of a closely related body of work known as Model Based Testing (El-Far & Whittaker 2001). Model Based Testing involves deriving test cases from a model that describes some aspects of the system to be tested. The derived model encodes certain properties of the model that should be used for testing. For example, in utilising
UML state chart models certain properties of the model can be identified and encoded as constraints (Abdurazik & Offutt 2000). These constraints can then be used as the basis for automated test case generation and execution. Recently this approach has been used in the testing of agent based systems (Zhang, Thangarajah & Padgham 2007).

Another approach at focusing the debugging task takes the approach of abstractions over the target program. This is especially important in domains such as distributed programming where the data, especially event data, can be overwhelming. By using the abstractions appropriate to developing distributed software Bates has shown that a debugging system, consisting of a model builder, event models and an event recogniser can greatly reduce the amount of event information being propagated to the developer. Primitive event instances need to be defined such that they can be automatically identified in a program. Once identified the program needs to be modified to announce the event to an external component (such as the event recogniser). Models are built using an Event Description Language (EDL), as defined in (Bates 1987). With such a language one can build expressions and further abstractions over the primitive events. Instead of being informed of the primitive event data, the developer is instead alerted to the meta events defined in the models.

The benefit of such an approach is a greatly reduced amount of event information One of the major limitation of this approach is that one needs to learn the EDL and also must manually define the models used for comparison. The model is built on the users’ interpretation of how the system should behave, based on such things as their interpretation of potentially informal design documents. This leads to another concern that the abstractions that have been applied do not filter out any information required for a particular diagnosis. In addition the diagnosis can only be successful if the model developed is a correct representation of expected behaviour.

2.5 Monitoring and Debugging Multi-Agent Systems

It has been argued that multi-agent systems merely represent a specific form of distributed systems (O’Hare & Wooldridge 1992). Several methods have been developed to assist in the debugging of distributed systems: recording a history of execution for analysis or replay (LeBlanc, Mellor-Crummey & Fowler 1990); animating the execution of a system at run-time by providing a visual representation of the program (Bruegge, Gottschalk & Luo 1993), and race detection algorithms to facilitate the detection of simultaneous access to shared resources (Schwarz & Mattern 1994, Naish
The debugging techniques developed for distributed systems can be used to facilitate the debugging of multi-agent systems to some extent. However, there are characteristics of agent systems that require specific attention. Traditional distributed systems support distributed information and algorithms whereas multi-agent systems address distributed tasks achieved by coarse grained agents. The individual agents within a multi-agent system are autonomous and they can act in complicated and sophisticated ways. Furthermore, the interactions between agents are complex and often unexpected. These issues and others need to be addressed for a multi-agent debugging approach.

2.5.1 Debugging in Agent Development Environments

JACK Intelligent Agents

The JACK Intelligent Agents™ system (Busetta, Rönquist, Hodgson & Lucas 1998), from Agent Oriented Software4 is an agent based development framework which provides a high performance lightweight implementation of the BDI architecture to enable the development of rational agents. It is a third generation agent framework following the development of dMARS (d’Inverno, Kinny, Luck & Wooldridge 1998) which itself was based on the Procedural Reasoning System (PRS) (Ingrand, Georgeff & Rao 1992)

JACK is a Java based framework that provides a programmer with access to the JACK agent language, which is used to implement the agents in the system. Being an extension to Java, JACK inherits all the features of Java, such as type safety, code portability and a widely supported and deployed execution environment (the Java Runtime Environment), and of course an extensive application programming interface. The JACK agent language is provided as a set of agent oriented keywords for the identification and specification of the agents (plans, events, beliefs etc) and statements which are used to specify the characteristics of agent components. Other statements are defined to allow for the manipulation of an agent’s state and to declare how the agent should respond to events that occur in the system.

JACK code is compiled to Java code and executes in the JACK kernel, which provides the management for concurrency among tasks, the default behaviour for handling events and failures, and a lightweight communications infrastructure that supports inter agent communication over distributed

---

4www.agent-software.com
processes. The kernel implements the BDI execution cycle including plan selection and goal persistence in the event of failure. JACK agents utilise plans from a plan library to achieve their goals within an environment. A plan is made up of the conditions that trigger it (the invocation condition), filters for determining which plan out of a set of plans should be selected for execution, and a plan body which defines the steps that should be followed by the agent.

Designing and implementing a JACK agent system is supported by the JACK Development Environment (JDE). The JDE allows one to design and implement agents by specifying, among other things, the set of plans that the agent can use and the messages or events that trigger them. The JDE has been developed in such a way that an executing system can be monitored and debugged from within the JDE. The debugging support provided consists primarily of a tracing and logging approach. Under debug mode the system can be viewed from a number of different perspectives. The overall design of the system as it has been implemented in the JDE can be monitored. As the system is executed the state of the system is reflected directly in the design by highlighting the current component that is executing by tracing the transitions between design elements. The system can be slowed down or stepped through and a user can specify which design elements should be traced.

More detailed information can be obtained by tracing the individual steps of program execution. It is possible to step through the program and observe the flow of control between the different agent entities, such as event sending and plan invocation. Further details can be obtained by utilising the plan tracing tool. This tool allows the user to step through each of the steps within a plan and the values of any relevant variable assignments can be viewed and modified.

For inter agent debugging an Agent Interaction Diagram can be used which details the messages exchanged between the agents in the system. The diagram includes the name of the sending and receiving agents as well as a user defined description that is expected to be defined when declaring messages. The diagram is customisable and messages can be filtered to allow the user to focus on a particular subset of messages.

**ZEUS Agent Toolkit**

ZEUS (Nwana, Ndumu, Lee & Collis 1999) is an advanced development toolkit for constructing collaborative agent applications. Collaborative agent applications are described as having rich, deliberative internal models that operate in open and time constrained environments. The focus of the
methodology and the tools is on providing support for building complex distributed agent systems. Some of the benefits of this methodology are rapid development time due to the tools available for agent construction as well as reusable agent components and code. Within the ZEUS framework two levels of functionality are proposed. The agent level functionality which includes the communication, cooperation and coordination functionalities that are likely to be used in most multi-agent systems. In addition task execution and the associated monitoring and exception handling should be done at this level. These functionalities are provided directly by the toolkit limiting the need for developers to create code that would be reused in future applications. The second level of functionality is where the specific problem is modeled and consists of the domain level problem solving abilities of the agents. Here, an application developer provides the domain specific knowledge to enable the agents to perform their tasks.

The agents in a ZEUS implementation can have their abilities, goals, resources, skills, beliefs and preferences defined in the Agent Definition Layer. These properties can then be used by the generic planning and scheduling algorithms provided. Above this layer is the Organisational Layer that defines the relationships between agents, as well as the knowledge that each agent has about other agents. Next is the Coordination Layer in which the agents in the system are modeled as social entities in terms of the negotiation styles and coordination styles that are relevant for the application. Finally the formal interaction protocols that are to be followed by the agents are selected, thus defining and implementing the inter agent communication for the application.

In an attempt to provide useful debugging information that could be absorbed effectively by a programmer Ndumu et al (1999) provide debugging tools based on multiple views of computation. The intention is that by combining results from different views the programmer will be better able to identify incorrect system behaviour without being overwhelmed by the large volume of debugging data that is associated with distributed agent applications. This idea is borrowed from earlier research applied to the ARCHON architecture (Avouris, Liedekerke, Lekkas & Hall 1993).

The approach taken to debug agents in ZEUS is also one of reporting and logging. A set of visualisation tools have been developed that allow for the visualisation of a system from the many different perspectives that are active in a multi-agent system. The collection of logging data is supported by a dedicated monitoring agent. Event data that is deemed necessary for any of the defined views is sent to this agent for real-time update or future replay of a system run. ZEUS defines the following views:
AGENT VIEWER: Shows details of the internal state of individual agents. It depicts the messages being received and sent out by the agent, a summary of the actions taken in response to incoming messages, a graphical depiction of the co-ordination process of different goals by the agent, status information for tasks the agent is performing, and a list of the resources available to the agent.

SOCIETY VIEWER: Shows the physical relationships between the agents and can also be used to show the messages exchanged between the agents.

REPORTS TOOL: Provides a set of reporting functions showing the society-wide decomposition and distribution of tasks including the status of tasks and their sub-tasks. One can select a set of agents and request that they report status of all their tasks/goals. Since each agent only has a local view of the problem-solving effort, this tool collates the local views to provide a more complete picture. A GANTT chart showing the decomposition of a task and the allocation of sub-tasks to other agents can be shown.

STATISTICS TOOL: Provides statistics about individual agents and society-wide statistics. For example, the volume and type of messages sent.

It is argued that by giving the user control over which of these views is adopted at any given time during a debugging activity the user will be better able to comprehend what is happening in the system. This appears to be a valid assumption, however, only anecdotal evidence is provided to support this claim and although the volume of data presented at any given time is reduced through this decomposition there is still a considerable amount of data to be considered.

A version of the ZEUS development framework has been released under an open source license and some minor software updates have been released over the last several years.

OBJECTAGENT

ObjectAgent is a software architecture supporting real time agent based deployment in a distributed autonomous setting (Surka 2001). Used for the simulation of spacecraft and satellites it allows for the easy creation and initialisation of agents, and agent communities. Agent parameters can be easily accessed and set in an interactive manner. ObjectAgent is a complete implementation platform that implements all the necessary functionality for agents and the simulation environment.
Similar to other agent development environments, agents are multi-threaded, and composed of skills (cf, plans/capabilities), have inputs/outputs and triggers. A flexible messaging architecture allows for message exchange between agents on the same machine or in a distributed manner. A simple message format that is based on natural language is supported. This architecture is said to “alleviate the need for extremely intelligent high-level agents” (Thomas, Mueller, Harvey & Surka 2001) thus simplifying the design of applications.  

However, as with most agent development environments support for monitoring and debugging the software could be improved. To this end, development of AgentCommand began with an initial objective of providing monitoring and analysis support for the ObjectAgent system. The monitoring and analysis support that is available is similar to that offered by other agent based platforms such as ZEUS. The focus is on message recording and replay. Initially all messages are recorded (by copying to an external recording agent) and can subsequently be inspected. Inspection occurs on what they call a MonitorWindow. Messages are loaded into this monitoring system and statistics can be plotted based on various parameters. A search feature allows users to find specific messages and filters, such as only showing messages between certain sets of agents. This can be applied to limit the data to be interpreted.

**Jadex**

The Jadex agent system is an agent development platform with support for specifying the mental properties of agents in the context of a BDI architecture (Pokahr, Braubach & Lamersdorf 2003). Jadex, having evolved from the Java based JADE (Bellifemine, Poggi & Rimassa 2001), allows for the definition of agents using a combination of XML and pure Java. As with JACK, syntactic extensions have been defined for concepts such as plans, goals and events. The Jadex framework consists of an API, an execution model that implements the BDI architecture, and predefined reusable generic functionality.

Agents are specified using an Agent Definition File (ADF) and a plan library containing the plans that implement the functionalities that the agent should provide. In the ADF one can specify the beliefs, goals, and plans for the agent in XML notation by tagging objects with the keywords. The API defines the Jadex specific concepts relevant for developing the plans. Beliefs can be any arbitrary Java object and a notable feature of Jadex is the provided intuitive OQL-like query language.
which allows for the formation of arbitrary complex expressions using the objects contained in the belief base. Only the plan header is defined in the ADF, each plan is declared in its own class with invocation conditions similar to those in JACK and other BDI systems.

Jadex provides a number of tools for monitoring and debugging a Jadex agent system. In addition to functionality for single stepping through the source code of the agents there is a BDI Viewer Tool. This tool is useful for viewing the internal state of a Jadex agent. Including such properties as the beliefs, goals and plans of the agent. In addition to this there is the Jadex Introspector which not only allows for the monitoring of the system but can also be used to modify how the system behaves at run-time, for example, by modifying how events are handled. As with most other systems a logging feature is used to collect and log the messages exchanged between agents.

The logging of messages has been leveraged by the ACLAnalzer tool which extends the standard logging functionality of Jadex (Botia, Hernansez & Skarmeta 2004). The underlying approach of this tool is to log messages and represent conversations as finite state automata which can be monitored to determine if certain conditions are breached. Conversations are partitioned by conversation id’s and once modeled in the FSA are considered to be in one of four states. An initial and final state, a possible final state that allows for more messages to be sent, and an error state that indicates the conversation is in error. The error state is only triggered if a message is not sent after a specified timeout is reached. There is no checking to determine if a given message was a valid response based on previous messages received.

In addition to logging the state of conversations the visualisation aspects of the system are enhanced by allowing different views over the message exchanges. Users can focus their attention on specific conversations and can also be alerted to situations where a conversation times out, in which case the logs will show which agent failed to act. Work on dealing with the issues surrounding visualisation of “huge” multi-agent systems has also been implemented into this tool. A data mining approach to clustering similar agents to provide more comprehensible graphs is proposed in which different groupings can be made on the same data by applying different rules (Botia, Hernansaeez & Gomez-Skarmeta 2007). This is seen to be advantageous because the user is able to obtain multiple views of the information.
2.5.2 Other Agent Developer Environments

There are numerous other Agent Developer Environments which support debugging with a few commonly expected tools. Platforms, including 3APL (Dastani, de Boer, Dignum & Meyer 2003) and JASON (Bordini & Hübner 2004), provide an agent internal inspection tool allowing a user to identify the variable assignments for the agent. In addition message display and step/run controls are supported.

2.5.3 Platform Independent Tracing and Logging

Lam & Barber (2004) have developed a tracing and logging approach with much the same functionality as many of the development environments we have discussed. Two things stand out about their approach. Firstly, they have developed a platform independent approach to debugging in which they propose a four step Tracing Method that begins with adding logging code to an agent system and finishes with verifying the system against the models of expected agent behaviour. The idea is similar to the model checking approach discussed in section 2.4.4, however, the verifying of behaviours must be done manually. What is interesting about this approach is that they try to present the logged information in a way that is close to some of the available design artifacts, such as state transition diagrams or message sequence diagrams to make it easier to identify any differences.

The other way in which they support the verification process is by creating relational graphs based on the logged data. This is done by filtering the logged data through a set of causal rules. An example of such a rule is as follows: If event $e$ occurs after action $a$ and $e$’s precondition is equivalent to $a$’s postcondition, then $a$ caused $e$. These rules, along with an optional set of user defined domain specific rules enables the construction of a detailed relational graph that helps to explain why certain agent behaviours were chosen.

This approach of creating causal graphs is also taken up by Botia, Hernanze & Skarmeta (2004) where they focus on the issue of graphing agent interactions. Using logical clock vectors (Lamport 1978) they are able to determine the ordering of messages and provide a visual representation of these messages which is said to be useful in reasoning about message exchanges. While helpful, it appears that in large systems with heavy interaction these graphs could quickly grow making it difficult to comprehend.
2.5.4 Limitations of Current Systems

The main limitation of current debugging tools is the reliance on the user for directing the search and discovery of errors. Each of the debugging tools provides good support for recording and presenting program information to the user but it is ultimately up to the user to identify where to focus their attention. All allow the user to trace some part of execution but there is no way to judge correctness. Furthermore, with the large and complex interactions occurring in multi-agent systems it is often very difficult to determine, even approximately, where one should focus attention after discovering a problem with the system. A notable exception to this general reliance on user intervention is the ACLAnalyzer tool which is able to identify when a conversation fails to make progress.

In addition to traditional debugging techniques where users can, for example, step through code, and in the case of JACK step through the program at the level of the design documents, the most common method for supporting the debugging task is the use of information gathering and visualisation to present a graphical depiction of system behaviour to the programmer. The focus is on the collection of information, usually agent messages but also the agent internals, and the presentation to the user. However, with the vast amounts of information the developer is often presented with too much information and experiences information overload reducing the effectiveness of the visualisation technique. Liedekerke & 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. In ZEUS providing different views does limit the information flow to some degree, however, it does not overcome the problem.

In summary some limitations of current multi-agent debugging techniques are:

- Determining where to look for errors is entirely user directed which is difficult when dealing with such complex programs.

- Programmers are generally presented with too much information making it difficult to understand what is really happening in the system.

- Without a procedure for identifying what sorts of information to look for it is unrealistic to know in advance what information will be useful when trying to debug the system.

- Most systems have no means of identifying where problems may be occurring. Even if the
developer notices an error it could take an unnecessarily long time to pinpoint the location of the error.

- They rely on the programmer interpreting the information correctly. Since the output of most of the debugging tools is raw messages the developer needs to inspect the contents of the messages and the flow of messages and try to determine what is going wrong. With a large number of messages this can be extremely complex, slow and error prone.

An alternative approach to providing a visual representation of agent data is to monitor the multi-agent system using knowledge about the expected activity of the system to automatically detect behaviour that may be incorrect.

### 2.5.5 Approaches to Monitoring Agents

Monitoring agent systems is the most applicable method for gathering information about what the agents in a system are doing, and why they are doing it. The previous section focused on the different approaches of monitoring agent systems for the purpose of debugging. Monitoring is also desirable in other more specialised areas; such as facilitating collaboration of team members engaging in team plans (Grosz & Kraus 1996), for fault identification with a view to agent adaption (Horling, Lesser, Vincent, Bazzan & Xuan 1999), for exception handling for the purpose of surviving common or generic failures (Klein & Dallarocas 1999), and for executing team plans, or individual plans in a group environment (Kaminka & Tambe 1999a).

In Horling, Lesser, Vincent, Bazzan & Xuan (1999) diagnosis is used to accurately detect faults based on observable symptoms. As with other approaches that compare observable behaviour with expected behaviour a model of expected behaviour is required. Expected behaviour is encoded in a domain independent way using a goal/task decomposition language. This structure dictates the specific goal/task hierarchy required to complete a task. Each step in the task can be annotated with expected behaviour, which includes such things as expected quality, cost and duration of a node within a task. Probabilities for failure can also be encoded along with the expected interactions with resources and other external methods. The actual diagnosis is performed using a causal model that is based on a large body of coordination and failure data that encode the possible domain independent failure symptoms. A key aspect of this work is that the diagnosis tasks need to be explicitly engineered into the system by the developers. Information about expected task behaviour needs to
be added to the system as well as the specific types of diagnosis that should be considered by the diagnosis system. Although this does require a certain amount of effort and expertise it does allow for control over the diagnosis process. Developers can set and modify both the types of diagnostic activities that should be performed as well as the specific levels for triggering these diagnostic activities. This is pointed out as an important characteristic given that more diagnostic information does not always mean better diagnoses.

The issue of what information should be monitored, how much, and by who is addressed in (Kaminka & Tambe 1999b). In this work it is argued that for effective cooperation to occur in a multi-agent setting team members must have a way of determining the state and actions of their team members. The focus is on monitoring the social relationships between agents in terms of the individual states as a relation over a required team state. By monitoring individual states it is possible to recover from failures in cooperation and coordination. How this information is obtained, and what pieces of information are required is referred to as the monitoring selectivity problem. The monitoring of teamwork relationships is achieved by modeling agents in terms of their hierarchical reactive plans. A plan-recognition algorithm is employed which matches observations of agent behaviour against the hierarchical plan trees. Detecting violations of relationship constraints is investigated and it is found that a simpler distributed detection algorithm is preferable over a centralised, more complex algorithm (Kaminka & Tambe 1999b).

One of the challenges in modeling teams is how the information that needs to be modeled is obtained. It is not always the case that diagnosis can be built into a system, as is done in (Horling, Lesser, Vincent, Bazzan & Xuan 1999). This is especially true when the system to be monitored is already implemented. Although Kaminka et al. found that a distributed algorithm was more effective than a centralised one, such an algorithm required that the agents performing the diagnosis were equipped with individual monitoring and diagnosis functionality. In situations where such additions are not possible other methods for monitoring teams is necessary.

More recent work by Kaminka et al. looks into this issue and proposes a method whereby teams are monitored and modeled as a function of the routine communications that are exchanged between team members to afford coordination (Kaminka, Pynadath & Tambe 2000, Kaminka, Pynadath & Tambe 2002). An observer that is aware of an individual’s plan hierarchy can monitor the communications exchanged and can hypothesise about the state of the agent. Given that a single message may
not uniquely map to a plan state it is necessary to disambiguate between possible plan states. This is achieved by generating a probabilistic model of the plan hierarchy whereby probabilities are gathered either from domain experts or from repeated observations of the system. Beliefs about the state of an agent are represented by a time series of state variables depicting which plans are active at a given time slice. With a probability distribution over these variables it is possible to represent the current belief held about a particular agent. The observer applies an initial belief about the top level plan that an agent is executing and this belief is propagated through the hierarchy at each time step. Algorithms for performing this belief propagation are provided for the situation where an observation is made (a message was intercepted) and also for when no observation was made. It is shown that although this method is an efficient method for probabilistic reasoning the scarcity of messages exchanged leads to an unsuitably low rate of accurately recognising behaviour (Kaminka, Pynadath & Tambe 2001).

A superior method is shown to be one in which the social relationships between the agents are taken into consideration. From the stance that the agents are expected to work together in achieving joint goals (Jennings 1995) it is possible to reason about the state of one agent as a function of other team members. To achieve this a global, fully expanded task model is generated by taking the union of all the individual task models. Given that individual plans may be utilised by multiple agents there is a reduction in the space complexity in the single model over the individual models since only one version of the plan needs to be modeled. There is also a reduction in the time complexity given that fewer nodes need to be explored. This approach is also used as a basis for work in which scalability is considered (Kalech & Kaminka 2005).

Diagnosing incorrect behaviour in multi-agent systems is also a focus of Klein & Dallarocas (1999). They identify what they see as the challenge to creating fault tolerant agents within a multi-agent system and refer to the failures that occur as exceptions. Examples of the characteristics of multi-agent systems that make them susceptible to failure are:

- **Unreliable Infrastructure**: In large distributed systems like the Internet, unpredictable node and link failures may cause agents to die unexpectedly, messages to be delayed, garbled or lost.

- **Non-compliant agents caused by buggy code or programmer malice**.

- **Emergent dysfunction**: Complex and dynamic interactions may result in emergent dysfunction and is mainly attributed to the lightweight coordination protocols. An example of emergent
dysfunction is the problem of resource usage oscillation in which, due to a delay in status messages, the resource availability is misrepresented resulting in agents switching back and forward between resources in an effort to achieve the least used resource (Klein, Metzler & Bar-Yam 2004)

The types of problems that are seen as important indicate that the focus of the work is on post implementation where the deployed agents are in operation but may encounter unexpected conditions (exceptions).

The authors introduce a domain independent exception handling service but unlike Horling et al. handling exceptions is removed from the agents themselves, which it is argued makes them easier to implement and results in better exception handling techniques. The approach involves the use of sentinel agents, one per agent, that monitor agent behaviour to identify symptoms of failure. If any symptoms are present the agent instigates its diagnosis component. Symptoms are encoded from a large body of coordination and exception handling research (Klein 1997). This research has led to the development of a taxonomy of generic problem solving processes and the associated failure types that can occur. When an agent is added to a system its behaviours are checked and the applicable failure modes are added to the sentinel.

The diagnosis mechanism is based on a heuristic model and on the presentation of symptoms through the sentinel agents. It is akin to a medical diagnosis process and the diagnoses are to be considered as hypotheses rather than deductions. To utilise the exception handler the problem solving agents need to include a set of basic exception handling functions to interface with the exception handling system. The interface makes use of a query language and an action language. Upon identification of an exception the action language can be used to direct agents to apply standard strategies to help avoid or recover from failure. This approach has been demonstrated in a number of situations. One such example is of how the exception handling mechanism can be used to recover from agent death while engaging in the contract net protocol (Klein, Rodríguez-Aguilar & Dellarocas 2003).

This work differs from Kaminka et al. in that the faults are pre-prescribed, and since each agent is accessible to the sentinel agent, complete information about the agent is available. This availability of information is in contrast to Kaminka et al. who see the unavailability of information as likely and have therefore examined methods to reason about agents in such situations.
2.6 Summary of Related Work

In the process of placing our work in context we have covered some basic areas such as agent concepts, AOSE methodologies, agent communication, as well as identifying related work. We summarise below the main groups of related work.

Petri Net Representation of Interactions

Petri Nets have been used to specify various aspects of agent development. Most attention has been on modeling and analysing the interactions between agents by specifying interaction protocols as Petri Nets. Protocol specification with Petri Nets was initially carried out without any supporting notation, but this quickly changed when an initial standard of AUML was proposed. Many saw the development of AUML as a useful way of initially specifying protocols but felt that they could be improved by modeling the protocols in a more precise way with Petri Nets.

AUML was essentially used as an intermediate notation and developments were made to convert the different AUML protocols to equivalent Petri Net versions. The development of Petri Net versions of the AUML protocols has certain advantages that were exploited for different purposes. In our work we wished to use the Petri Net version of a protocol to monitor and analyse conversations for use in debugging agent interactions.

Although there were examples and descriptions on how to convert specific AUML protocols there was no generic procedure for translating an arbitrary AUML protocol to an equivalent Petri Net. This is the main difference between our work on Petri Net translation. In this thesis we provide generic translation rules for the building blocks of the AUML notation such that a protocol specified in AUML can be converted to a Petri Net version by following our translation rules.

Since our initial publications in this other approaches have appeared in the area of translating AUML protocols to Petri Net representations. Cabac (2003) provide a different method for translating protocols and they focus on the initial AUML specification whereas we provide translation rules for the new AUML specification. Gutnik & Kaminka (2006) utilises similar notational structures to our work, however, they use coloured Petri Nets with a view to explicitly represent certain contents of the messages that feed into the Petri Nets. Given that they are working in the area of monitoring agent systems they provide a monitoring algorithm that also has some similarities to our work. They also focus attention on the initial version of AUML.
Debugging in Agent Development Environments

A number of agent development environments have been reviewed with a focus on the debugging support they offer. Traditional debugging support such as stepping through the code, accessing variable assignments and in some cases changing values is supported by all. One interesting variation to the stepping through the program is from JACK in which the code is matched to the design and one can step through the code and design at the same time. Another common feature is the logging and tracing of message data so that it can be visualised. To help reduce the amount of data being presented all provide various methods for filtering and displaying data.

With the exception of the ACLAnalyzer tool in Jadex all debugging tools lack an ability to help the user identify bugs in their programs. All provide good debugging support, but it is a manual task in which the user needs to determine where to focus attention. With large complex programs this is extremely difficult as it is often a time consuming task identifying the general problem area.
Chapter 3

Debugging Framework

In order to more effectively address the issues surrounding the debugging of multi-agent systems we
first explore the kinds of bugs most likely to occur when developing such a system. We begin by
discussing the need for specialised debugging support that is based on the characteristics of the agent
paradigm. Such a consideration is necessary as the types of bugs that emerge during development
are influenced by the paradigm adopted. We present two classes of bugs that are characteristic of the
types of bugs that are commonly encountered during the development of multi-agent systems: those
that relate to the selection of plans in response to the adoption of goals and those that relate to the
interactions between agents.

We also describe our general approach for providing support to detect and report about these
bugs. We present the debugging framework that we have developed which proposes methods and
processes for generating debugging support from the design artifacts that are developed during the
pre-implementation phases of an agent-oriented software engineering process. We conclude the chap-
ter with the details for generating debugging support to help identify bugs related to the selection of
plans in response to the adoption of agent goals. Chapters 4 and 5 provide the details of generating
debugging support for interaction related bugs.

3.1 Focusing Debugging to the Agent Paradigm

The debugging problem is one of resolving differences between what the system designers expect of
their system and how that system behaves during execution. These differences are said to most often
manifest to the developer as output that differs from that which is expected (Vessey 1985). To debug a system we must, therefore, understand how a system is supposed to behave as well as understand how the system does behave if we are to identify and resolve these differences.

The software paradigm that is adopted affects the way that a developer understands the implementation with respect to the design. The concepts and notations that the paradigm supports are used by the developer to describe the system, and to reason about the execution. During debugging the developer creates a mental model of the system based on the concepts from the paradigm that is being used. For the debugging of a multi-agent system we expect that the developer will be considering any observed problem in terms of concepts that define the agent paradigm: events, plans, agents, messages, protocols, and so on.

Providing debugging support, in the form of debugging tools, should therefore focus on the properties of the target paradigm. Indeed, in Forin (1988) and also in Auguston, Jeffery & Underwood (2002), they remark that a common goal of supporting the debugging process is to assist the developer in their understanding of the system in a way that is consistent with the abstractions that they have used to describe the system. A further reason for providing debugging support that is focused on the target paradigm is given by Tukiainen (2000). It is argued that the types of errors that are introduced into a program are dependent on the paradigm used to develop the program. This means that for any given paradigm there may be a set of bugs that are specific to that paradigm. In response to this we undertook to identify the types of bugs that are characteristic of the multi-agent paradigm in an effort to ensure that our developed framework was appropriate for the detection of these types of bugs.

3.2 System Design Artifacts as Debugging Components

We wish to develop debugging techniques with the following goals in mind.

- Reduce the volume of debugging information presented to the programmer to more effectively support understanding of the system.

- Provide a means to automatically identify and locate potential problems based on the knowledge of common bugs so that the developer can be informed of, and directed to the cause of such problems.

- Deliver an abstracted or summarised report of debugging information to reduce the amount of
messages that the developer has to review, and to present the information in terminology that is consistent with the agent paradigm.

By focusing our attention to these points we aim to support the developer in reconciling the differences between the run-time execution of a system and that of the design. Instead of simply focusing on the implemented system, we must identify what the system is supposed to do and explicitly connect that to what is observed at run-time.

One way to address the problem of information overload, and to help reconcile the differences between the design and the implementation, 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 the Prometheus methodology also offer opportunities for detecting run-time executions inconsistent with the specification.

The benefits of linking the debugging process to the overall software development process have been recognised since the early days of Software Engineering Development Environments (Müllerburg 1983). System design artifacts encode the requirements of the system in such a way that the system can be implemented to realise the design expectations. The design artifacts provide the understanding of the implementation. Understanding of the rules that the design artifacts encode supports communication between the different phases of development and provides for a consistent understanding of how the system should be implemented.

If we take, for example, an agent acquaintance model that is common among a number of methodologies, such as Prometheus (Padgham & Winikoff 2004), Roadmap (Juan, Pearce & Sterling 2002) and GAIA (Wooldridge, Jennings & Kinny 2000), we see that this model depicts communication links between agent types, see figure 3.1. This model is a high level model indicating the possible communication paths that exist in an agent community. It can be used to help identify dependencies between agents and can also be used to identify potential communication bottlenecks (Padgham & Winikoff 2004). The communication paths define which agent types can interact with which other agent types and can therefore be used as a rule indicating allowable communications. Figure 3.1 shows that an agent of type \( P \) will interact with an agent of type \( Q \) but not an agent of type \( R \). If an agent of type \( P \) were observed to have sent a message to an agent of type \( R \) then this would be an
error as the interaction does not conform to the design specification.

Our central thesis is: *The design documents and system models developed when following an agent based software engineering methodology can be incorporated in an agent and used at run-time to provide for run-time error detection and debugging.*

### 3.3 Architecture of the Debugging Framework

The debugging framework that we have developed uses the design artifacts, applying to them a process to produce debugging components to facilitate the automatic debugging of agent systems. The debugging framework is based on the premise that we can utilise the system design artifacts as a partial specification of correct system behaviour. We describe this framework in terms of the processes that are applied as well as the underlying debugging infrastructure required to support the observation of the system, comparison of the system against the developed debugging artifacts, and the reporting of the system to the user.

Figure 3.2 provides an overview of our debugging framework. This consists of a set of debugging components, framed with a solid line and annotated with C1, C2, and so on, that together represent the run-time debugging environment. In addition to the debugging components are a set of processes, framed with a broken line and annotated with P1, P2, and so on, that represent the processes that need
CHAPTER 3. DEBUGGING FRAMEWORK

Monitoring Components

(P1) System Design Artifacts

(P2) Identify suitable design artifacts.

(P3) Generate suitable machine interpretable format.

(P4) Instrument source code.

(C1) Runtime System

(C2) Library of debugging artifacts specifying correct behaviour.

(C3) Monitoring Components

Event Monitor
Interaction Monitor
Other Monitoring Structures

(C4) Reporting to the User

Figure 3.2: Debugging Framework
to be applied to generate the debugging components.

The run-time system (C1) in the center of the figure depicts the agent system that is the focus of the debugging exercise. It is developed using the system design artifacts (P1). During execution the run-time system sends information to one or more monitoring component (C3). The monitoring components are supplied by a library of debugging artifacts that specify correct system behaviour (C2). The debugging artifacts represent a partial model of correct behaviour that is generated by following processes P1 through P3.

The processes in the debugging framework specify how to develop a suitable debugging component from a system design artifact. Each of the design artifacts from the set of system design artifacts (P1) are considered. From these we identify and select suitable design artifacts that could be used as debugging components (P2). From the identified artifacts we develop a partial model of correct system behaviour. This requires that we develop a machine interpretable format for the design artifacts (P3). Each of the developed debugging artifacts feed into the library of debugging artifacts that is used in the monitoring component for run-time debugging.

The monitoring components are where the comparison between actual system behaviour and expected system behaviour is carried out. Before such a comparison can be carried out we must determine a method for extracting the relevant run-time information from the run-time system that should be sent to the monitoring components. The necessary information is identified and the source code is instrumented (P4) so that when certain events of interest occur in the system they are forwarded onto the monitoring components for consideration. Once the system has been modified to send the relevant information it can be compared to the debugging artifact and then a report can be sent to the user via the user interface in (C4).

In this thesis we focus our attention on two important design artifacts, Interaction Protocols for detecting interaction related bugs and Event Descriptors for detecting incorrect interactions between events and plans. Depending on the artifact in question, and on the level of support for detecting violations of the artifact, the effort that goes into generating suitable machine interpretable debugging components can be quite substantial. Indeed, chapter 4 of this thesis is devoted to describing the process of generating the machine interpretable format for AUML interaction protocols and chapter 5 is devoted to describing how these protocols can be monitored.

This framework forms the foundations of our approach to debugging multi-agent system. Before
we move forward and discuss the generation of the different debugging artifacts we will first provide a review of the types of bugs that we will expect to resolve.

### 3.4 Analysis and Classification of Bugs in MAS

Through our own experience in research, implementation and teaching agent oriented development we have some insights into the characteristics of the types of bugs that are common to multi-agent systems. These experiences formed the basis of our investigation into how these bugs manifest, and to what types of debugging support would be required to support their location and resolution. In addition to this we undertook a systematic investigation into the types of bugs that are characteristic of those found in multi-agent systems.

Our approach was to gather debugging information from projects developed following an agent oriented paradigm. We did this by requesting that students from an agent oriented design and development class keep detailed bug logs for any bugs that they discovered while developing their projects. The projects were developed using JACK, a commercial, Java based agent development environment. Students design and implement a system with a focus on applying an agent oriented software engineering methodology (Prometheus). The following information was requested for each bug that was discovered in the course of implementing their projects.

- A general description of the bug. We did not specify what types of bugs we were interested in or how they should describe the bugs.
- How they discovered the presence of the bug.
- How they located the cause of the bug. We requested that they spend some time indicating the actions that they took to discover the location of the bug. We were interested in identifying what tactics the students found useful in finding the bug and what information they needed to help locate the bug.
- How they fixed the bug.

In addition to the bug log we requested that for each bug the source code that exhibits the bug behaviour be submitted. This enabled us to reproduce the bug to verify the bug report and to consider other strategies to help identify and locate the bug. We also asked about the students’ own reflection
on their efforts to debug their programs. We were particularly interested in learning what the students felt would have helped them to uncover and resolve the bugs more easily.

After receiving the submissions the bug reports were analysed and the source code for each bug version was reviewed. We also tested the final submissions to uncover any undisclosed bugs, of which many were found. The bug logs and the assignment submissions indicated that the bugs that were discovered were not all specific to agent oriented development. Students identified a number of low level bugs, such as incorrect variable assignments, logic errors and incorrectly referencing variables. We refer to these errors as *low level* errors and although they are an important part of the debugging process we do not focus on them in this thesis.

Since such errors are not specific to development in the agent paradigm they can be targeted by more general debugging techniques. However, the presence of a low level bug often manifests to the observer at a level of abstraction that more closely matches their mental model of the system. It is possible that the students had at first identified a bug at a higher level of abstraction but after pinpointing the problem opted to describe it at the statement level.

Our investigation led us to focus our attention on two common categories of bugs. The first concerns the selection of plans in response to event triggers. We classify these bugs as *plan selection bugs* and they are to do with the internal processing activities of the agent. The other category of bugs concerns those bugs that relate to the interaction, via messages, between agents. We classify these bugs as *interaction related bugs*.

### 3.5 Interaction Related Bugs

In a multi-agent system the required functionalities of the system are decomposed and allocated to individual agents, each with the specialisation necessary to meet some of the goals or tasks of the system. It is common that for any given task the decomposition does not result in independent execution of the task. That is, agents are required to engage in communication activities to support their individual activities. This requirement is well stated by Akkermans, Gustavsson & Ygge (1998)

> “... a task that is carried out by one agent may produce results in the form of information objects that need to be communicated to other agents.”
To facilitate this communication interaction protocols are typically used. An interaction protocol defines the valid conversations regarding some particular task. The state of the protocol constrains the set of valid messages from which an agent can select to send next. The interaction protocols define the interface between agents for the purpose of ensuring that agents can communicate to bring about some desired state in the system. In this research we assume that interaction protocols are represented using the AUML interaction protocol specification language.

These types of interaction protocols enable autonomous agents to interact and together achieve tasks and goals that cannot be achieved independently. As such, it is necessary that the interaction protocols are followed correctly to ensure that the correct information and requests are being communicated and understood. The interaction protocols specify the set of appropriate responses for a given state of an interaction. They allow the designer to specify all legal options at a given point in an interaction. The correct functioning of the multi-agent system as a whole is dependent on the correct use of the messages that comprise the protocols of the agent system.

Interaction related bugs, where agents at run-time do not interact with each other as expected, are a common source of problems in multi-agent systems. Following is a discussion of several types of interaction related bugs that we have identified as being characteristic in multi-agent systems.

Sending the wrong message

We define the sending the wrong message bug as the act of an agent sending a message that is not appropriate given the current expectations as defined by the protocol.

What constitutes an appropriate message is defined during development and is encoded in the protocol. The current state of a conversation based on a specific protocol dictates which messages are valid next messages.

This bug represents the case where the protocol requires an agent to send message $m_1$ but instead some other message, $m_2$, is sent. For this bug we note the following variations. Firstly, there is the case that the message $m_2$ is a valid message in the protocol but is currently not valid based on the state of the conversation. It may be that the conversation has not yet advanced to the stage where the message is valid or that the conversation has advanced beyond the stage where the message is valid.

Another situation is where message $m_2$ is sent but is not a valid message in the protocol. For this case message $m_2$ is not valid for any of the protocol states. The message belongs to a different
protocol and is not relevant to the protocol being used under any circumstances.

**Failure to send a message**

*We define failure to send a message as the act of an agent failing to send a message when the protocol required that one be sent.*

As developers we define the conditions under which a message should be sent from one agent to another. These conditions are based on the capabilities of the individual agent and the requirements of the tasks that need to be completed by the agents. The agents act autonomously and can select what actions to adopt. However, for effective inter-agent communications it is expected that agents will consider their obligations as a member of a community and as such will reply according to the protocols they choose to use.

Failing to send a message when one is expected is often symptomatic of a failure in some part of the agent system. When developing the agent goals and plans the requirements of the protocols are considered. Failure to respond according to the protocol can be an indication that the agent or some part of the agent has stopped functioning correctly.

**Sending a message to the wrong recipient**

*We define sending a message to the wrong recipient as the act of sending a message to an agent that is not the intended recipient as specified by the protocol design.*

When sending a message to another agent the receiver is chosen and explicitly referenced in the message header. If at run-time the message is sent to a different agent than that specified in the design this is incorrect. The wrong recipient may be wrong based on the agent role that received the message, or could be wrong based on the agent bindings that may have already occurred in a conversation.

The behaviour of the system when this bug is encountered is very difficult to predict. It is possible that sending the message to the wrong agent does not have any obvious adverse effects. For instance, if the incorrect recipient is capable of understanding and servicing the message then the recipient can reply to the sender with the required information. Alternatively, the agent that is addressed, may not exist, or may not have the capability to handle the message, in which case the request will not be handled appropriately. If an agent is sent a message that it does not understand, that is, the message is
from a protocol for which it has no knowledge about, there is no expectation that the agent will reply in a consistent way. It is possible that the agent is defined to reply with a *not-understood* message, but there is no guarantee that such a convention is followed.

### Sending the same message multiple times

*We define sending the same message multiple times as the act of an agent incorrectly sending the same message multiple times when only one message should have been sent.*

When an agent wishes to send a message to another agent it should do so only once, unless the interaction protocol or some other logic dictates otherwise. If the same message is sent multiple times it is possible that the message will be processed by the receiving agent multiple times. Doing so could result in incorrect or unexpected behaviour. For instance if a customer agent sends a message to purchase goods from a merchant multiple times, it is likely that the merchant will also process the order multiple times, sending more than one order.

In BDI systems it is common to have a number of different ways of achieving the goals that the agent adopts. These different ways are represented as plans, each with a context condition that defines in what situation the plan is applicable. The choice of plan is made at run-time and it is inside these plans that the messages are created and transmitted to other agents. The plan failure mechanism within this framework enables the agent to select alternative plans if a plan fails to achieve the goal for which it is selected. If the same plan can be retried after a message is sent but before the goal is achieved, or if alternative plans can be tried that send the same message upon failure then unless care is taken it is possible that the agent is unaware that it might have sent the same message multiple times.

### 3.6 Plan Selection Bugs

In BDI agent systems such as JACK (Busetta, Rönnquist, Hodgson & Lucas 1998), JAM (Huber 1999), and Jadex (Pokahr, Braubach & Lamersdorf 2003) in which agents select an appropriate pre-defined plan from a plan library, one common cause of errors is incorrectly specifying when a plan should be selected by the agent for execution. 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.\footnote{Both these situations may occur legitimately, however, they are sometimes an indication of a problem.}

It is important to correctly specify the relationship between the events that the agents respond to and the plans that they use to handle the events. In complex agent systems it is often difficult to understand the execution and selection of plans in response to the goals, because of the variety of legitimate execution paths. This, along with the ability to retry those plans that fail, result in there being many different execution paths that are correct. Manually following these numerous execution paths to verify that a given execution did not violate a specification can be extremely difficult and time consuming.

The detailed design part of the Prometheus methodology focuses on implementation platforms which use a plan library with each plan being tagged as relevant to a particular goal or event. Often there will be multiple plans relevant for any given goal/event. A \textit{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 \textit{applicable plans} at that time. These are the plans suitable for responding to the event/goal, at the particular time.

The Prometheus methodology prompts the developer to consider how many plans are expected to be suitable for each event type in all possible 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 \textit{coverage} and \textit{overlap}. Having full coverage specifies that the event is expected to have at least one 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 that are relevant for the event must not have any “holes”. An example of an unintended hole that can occur is if two plans are specified for an event, one with context \textit{temperature} < 0\degree and the other with context \textit{temperature} > 0\degree. \textit{Temperature} = 0\degree 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 run-time a situation occurs when there is no applicable plan for an event of that type, then an error
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 when multiple plan types are applicable or when a single plan can result in multiple versions of itself based on the variable assignments that may occur during plan initialisation. For example, in JACK if there is more than one way to satisfy a context method’s logical expression, there will be multiple instances of the plan that are applicable. One applicable instance will be generated for each set of bindings that satisfy the context condition. The developer is also prompted at design time to specify which of these situations is expected if overlap is possible.

### 3.7 Debugging Plan Related Bugs

Incorrectly specifying the selection conditions for plans is a common problem that we have identified. This can result in either no plans being selected, when it was expected that a plan should be selected, or the wrong plan being selected. We wish to provide support for these two kinds of errors.

The Prometheus methodology provides an Event Descriptor template for describing the properties of an event which triggers or activates some plan(s) in the agent’s plan library. One aspect which the developer is asked to consider and document is whether it is ever expected that no plans could be triggered or if multiple plans could be triggered. These are coverage and overlap properties described in the previous section.

Coverage is the term used to refer to the concept of whether, for a given event, under all correct executions, there will be a plan with a matching context condition. Overlap refers to the concept of whether, for a given event, it is possible to have more than one plan that is applicable. In such a case it would then be necessary to choose between the multiple applicable plans.

To encode these properties into a machine interpretable format we take the original design artifact and determine suitable transformation rules. The output of this step is a process or mechanism which can take as input a set of event descriptors and produces some machine interpretable artifact. Each event descriptor includes a boolean field to indicate if the event should have full coverage and also if it is possible to have overlap. Extracting this information is a simple case of iterating over each of the events and extracting the value of the coverage and overlap fields. If the event is seen to have
full coverage, that is coverage is \textit{true}, then the event name is appended to a record that lists all events with full coverage. In addition, if the event does not allow overlap, that is overlap is \textit{false} then the event name is appended to a record of all events that do not allow overlap.

The resulting debugging library artifact for the event descriptor is a simple text file for the coverage data, indicating which events are expected to have full coverage, and another for the overlap data, indicating which events allow overlap. The files contain a single event name per line and play more of an intermediary role, rather than an active artifact that is compared at run-time. We use these intermediary files to identify which source files need to be instrumented to enable the verification of coverage and overlap data at run-time.

\textbf{Instrumenting Source Code}

To enable the automatic detection of bugs we need to have the run-time system providing our monitoring components with data that can be analysed and compared with expectations of the system. The process of instrumenting the source code is concerned with identifying how and what information needs to be sent to the monitoring components such that we can identify if the events are handled appropriately. For coverage we are interested in identifying any situation in which an event that has been declared to have full coverage does not have a plan selected to handle it. For overlap we are interested in identifying any situation in which an event that is declared to have no overlap has multiple plans applicable.

The process of instrumenting the source code is necessarily platform dependent since one must modify the source code of the agent application to generate the required debugging events. We explain the modifications in terms of a JACK agent system, however, the concepts are generic and should be easily applied to other platforms. When an event is posted an applicable plan set is generated which contains the set of plans from which one plan should be selected to handle the event. Ideally, after an event has been posted the applicable plan set could be directly examined to identify violations of expectations regarding coverage and overlap. JACK provides a meta plan facility that allows access to the applicable plan set which is intended to support reasoning about plan selection. We use the information this provides for our debugging purposes.
3.7.1 Overlap

The procedure for identifying overlap is to identify any situation where the applicable plan set contains more than one plan. In JACK, when multiple plans are applicable an event is posted to allow for reasoning about which of the applicable plans should be executed. Support for making this decision can be encoded in a meta level plan where the logic for choosing between the applicable plans can be defined. This meta level plan is triggered by an event called a plan choice event. Since a plan choice event is only posted if the applicable plan set contains multiple plans we can use the fact that the event is posted as evidence that overlap has occurred. If a plan choice event is posted with respect to an event declared as not having overlap, then we can conclude that an error exists.

To identify such an occurrence we define a single plan to handle all occurrences of the plan choice event for all events that are declared as having no overlap. To ensure we only check relevant events we query the artifact library and extract all the event names that are declared to have no overlap. This ensures that we do not interfere with the normal meta level reasoning that may occur for events which support overlap. The function of this plan is to simply inform the monitor that a violation has occurred and to provide the necessary event information.

3.7.2 Coverage

Detecting coverage violations deals with the situation where something does not happen, rather than does happen. When an event is declared as having full coverage the event should always be handled by a plan. To detect coverage violations we are interested in identifying the situation where an event does not get handled at all. Unfortunately, as observers we do not have any direct information available to inform us that some action has not happened. In JACK, if there are no applicable plans then there is no applicable plan set to check. Nor will there be a plan choice event as was the case for overlap. We have the problem that we have no way of determining that no plans were applicable other than observing that none of the plans were executed.

There are two options for identifying a coverage violation.

- Register when an event is selected, and when a plan responding to that event fires. If no plan is triggered within a predetermined time limit we assume the event has not been handled.

- Define a plan that will always be applicable and ensure it has the lowest priority of any plan
that handles the event. If it executes without any other plan being tried and failing then we assume the event was not correctly handled.

Both these approaches require modification to the source code, which can be readily automated. We use the first method.

A registration module was developed that can be added to an agent system. Events that require full coverage are modified so that when they fire they report to the registration module. Plans that handle these events are also modified so they report to the registration module when they begin execution. When an event reports to the registration module a timer is set for that event. The timer will continue to run until either a plan reports to the registration module, in which case the event was successfully handled, or the timer for the event expires, in which case the event was not handled and a coverage error has occurred. If the latter occurs the module will report to the monitoring component with information detailing the coverage error. Scripts have been developed to automatically add the necessary statement to the source code for any event that requires full coverage. The debugging artifact library is queried to identify which events require full coverage. Those that require it have code added so that upon firing they report to the registration module.

Similarly, all plans that are defined to handle an event that requires full coverage are modified to register with the monitoring component as soon as they execute. These modifications have the effect that the first action performed by the plan is to inform the registration module that the plan has begun execution. The reporting statement includes the unique identification number of the event instance which enables the monitoring module to match the event instance with the plan instance. These modifications to the source code have been fully automated and can easily be added to the build scripts of the agent application so that any time the system is recompiled the source code is also updated with these necessary debugging enhancements.

### 3.7.3 Reporting to the User

The information that the user receives from the debugging tools is of course dependent on the monitoring component being used. In the event that the individual monitoring components identify a bug that monitoring component sends a report to the debugging interface. The reports are of a pre-defined format and are in terms of the design artifacts which the monitoring component represents.

In the case of both coverage and overlap the monitor reports the details of the error, the event
name and event id and a possible reason for the error. In the case of an overlap error the report will state that the event being reported about requires that no overlap should occur. It will state that multiple applicable plans were found and that this is an error. For coverage errors the report will state that the event being reported about requires that full coverage should occur. The report will indicate that the event was posted but no plans were executed to handle the event and will indicate which plans have been defined to handle the event.

A suggestion as to what action should be taken to try to identify the location of the bug is also provided. In both cases the debugger will advise that the context and relevance conditions of the plans that are defined to handle the event be checked and that the posting method and the parameters passed in are checked. The reporting interface can query the debugging library to determine which plans should handle the event. This provides the user with the set of plans that should be checked for error and is valuable information for helping to quickly identify the cause of the error.

The process followed in this section is the general approach to taking a design artifact and using it as a debugging aid. Using coverage and overlap specifications as a debugging aid is a relatively straightforward exercise yet the debugging support it offers is substantial. We will explore these benefits in the evaluation chapter.
Chapter 4

Translating AUML Protocols to Petri Nets

In the previous chapter we presented the framework for generating debugging artifacts from well-formed design artifacts and discussed how event specifications can be used to develop debugging support for plan selection. In a multi-agent system correct interaction between agents is vital for the correct functioning of the system. Providing debugging support based on the interaction protocols is an excellent way to help identify agent behaviour that does not meet the developer's expectations.

In this chapter we describe how interaction protocols, in particular those specified using AUML notation, can be converted into a suitable underlying representation to be used inside an automatic debugger. We have selected Petri Nets as the model that is to be used to represent the AUML protocols internally to the debugger. Petri Nets are an appropriate representation because they are a simple, formal and machine interpretable model. In addition there exist algorithms (and tools) for checking for various properties (deadlock, liveness, etc.) of Petri Nets and given an AUML protocol we could translate it to a Petri Net then check its properties.

We first describe our process for converting AUML fragments into Petri Net fragments. This involves identifying the possible states in a protocol and applying some specific transformation rules to convert the different AUML interaction operators into a suitable equivalent Petri Net fragment. Our transformations are general purpose and local so they can be used to convert most arbitrary AUML protocols.

There are certain issues that can arise when trying to convert awkwardly defined AUML patterns
to Petri Nets. To ensure protocols converted using the rules set out in this chapter are correct, we discuss these issues and suggest how such protocols can be modified so the intent of the protocol remains, while allowing the protocol to be converted correctly.

We conclude the chapter with some examples showing how to convert a complete AUML protocol to an equivalent Petri Net version which could then be used in a debugging tool.

### 4.1 Converting AUML Components to Petri Net Fragments

In selecting AUML interaction protocols as a candidate for debugging agent systems we are working from the assumption that, in a system where protocols have been defined to facilitate the exchange of messages, it is expected that at run-time the protocols are adhered to. For a given AUML interaction protocol, the protocol defines the set of allowable messages as well as the rules for when each message can be sent. There is, therefore, a dynamic aspect to these protocols. At run-time, when agents exchange messages we say that they are engaged in a conversation. When we wish to identify a conversation based on the protocol that is being used we will use the term protocol instance to clarify that we are talking about a specific set of messages that have been exchanged within the constraints of the protocol.

AUML protocols are a useful artifact for humans to communicate the rules for the interaction protocols that their agents should follow, but these are not readily machine interpretable, as is required for automated debugging. To determine if a protocol has been violated we need both a model for comparison (the protocol) and a way to record the interactions that occur at run-time (the messages sent). We chose the Petri Net notation as the internal representation for our interaction protocol debugging artifact. Using Petri Nets we can re-define the AUML protocols and then at run-time model a conversation in an instance of the Petri Net protocol to reason about the message exchanges. By using Petri Nets we are not tied to any specific protocol specification and can handle other design notations.

For the purposes of generating the Petri Net protocol from an AUML protocol we consider an AUML protocol as a set of protocol states between agents in which messages sent by the agents cause an instance of the protocol to advance from one state to another. The first step in the conversion is to add state labels to the AUML protocol. These states are not part of the original AUML specification but help to describe the dynamic behaviour of the protocol. Furthermore, the transformation
procedure utilises the state labels when performing the local transformations. We do not distinguish between the state of sending a message and receiving a message. We consider sending a message as an atomic action that advances the protocol into a subsequent state from which a following message can be sent. By identifying states in this manner we ensure that each message that is sent in the protocol advances the protocol to a unique state.

We label states as follows: select a state label to begin labeling the states. The choice of state labels is arbitrary, we prefer to use a series such as \{S_0, S_1, \ldots, S_N\}. Using such a series the first state label would be \(S_0\). The start state of the protocol, that is, the point in the protocol from which the first message in the protocol can be sent, is marked with this initial state label. Following this we generate the next label in the series and apply it to the other end of the message, the head of the message. By labeling states in this way we are explicitly indicating that messages link protocol states and as such messages are used to advance a protocol from one state to another.

We consider this newly marked state as the state of having sent a message, but also as the state of being ready to send some next message. As such, we use this same state label and apply it to the tail of any message that can be sent next. Graphically this means that there may be multiple points on the AUML protocol with the same state label. From the point of view of the protocol we consider any point on the protocol with the same state label as the same state. We continue the labeling procedure by generating a new label in the series and applying it to the head of the message. This continues until all messages have labels at their head and their tail.

Figure 4.1 depicts how we would label a simple AUML protocol. We first select a start label, \(S_0\) and apply it to the start state of the protocol. Following this we generate the next label in the...
series, $S1$ and apply it to the head of this message. From state $S1$ it is possible to send message $b$, as such we take the current label $S1$ and apply it to the tail of message $b$. We continue the procedure by generating the next label in the series and applying it to the head of message $b$. There are no messages left and as such the labeling procedure terminates.

This labeling procedure can be seen as a reachability problem. From a given state in the protocol we need to determine what messages can be sent from that state. In a simple example like the one we have just shown there is only a single message that can be sent. However, in many AUML protocols there will be interaction operators that offer more flexible choices of subsequent messages. When we are applying the state label to the next message, we must identify all messages that can be sent from the current state.

To illustrate this consider figure 4.2, this is a marked up version of the FIPA Propose Protocol introduced in section 2.2.1 (figure 2.1). After a propose message is sent the protocol will advance to state $S1$. From this state we reach an alternative interaction fragment, this means that the agent has a choice of sending one of two messages. Both messages are thus reachable from state $S1$, as such we label both of these messages with state label $S1$. When the protocol is in state $S1$ the participant agent has the option of sending an accept-proposal message, which would advance the protocol to a new state $S2$, or a reject-proposal message which would advance the protocol to a different state $S3$.

This initial labeling of the AUML protocol does not take into account the specific type of interaction operator that the message is found inside. We simply generate new labels, apply them and continue until the protocol is completely labeled. It is in the second stage, where we generate the
Petri Net protocol that we consider the specific interaction operators. For some interaction operators, such as the Loop interaction operator, we apply new state labels on the AUML protocol before generating the Petri Net. In others, such as the Alternative interaction operator, we have developed specific transformation patterns that are to be created when generating the Petri Net. We will discuss these rules in the following section. We now turn to the issue of generating the Petri Net protocol from the AUML protocol, following this we discuss the specific rules for applying the transformations for these interaction operators.

Our equivalent Petri Net protocols are made up of the standard places, transitions and arcs that are found in pure Petri Nets. We have, however, introduced a further subclass of the place type so that we can represent both AUML protocol states as Petri Net places and AUML protocol messages as Petri Net places. Our Petri Net protocols are thus made up of the standard transitions, but we have two types of places: state places and message places. This change is purely for notational convenience and does not change the underlying mechanics of the Petri Net.

Formally, we define a subset of the set $P$ (the place set) which we call $MP$ (the message place set). $MP$ contains all those places that are message places. Message places are identified by checking a lookup table that we define for each protocol, this lookup table consists of all the valid messages in the protocol. The message places are special in that, unlike state places, they do not receive input from any transitions. Message places are used as the input places for adding tokens to the Petri Net. This is no different to how tokens are introduced in pure Petri Nets, however in pure Petri Nets there are typically one, or only a few input places. In our Petri Net protocols we will have one input place for each message in the AUML protocol.

The states from the AUML protocol are connected to transitions in the normal manner to enable the transition from one protocol state to another. To control when these transitions, from protocol state to protocol state, should occur we connect the message places in such a way that the receipt of a message enables a transition from a specific place and causes a conversation to advance to some subsequent state. Figure 4.3 shows how the propose message from figure 4.2 is realised in our Petri Net notation. It should be noted that the appearance of a token in the message place enables the transition as long as there is a token on the state place which that message is partnered with. Stage 1 is the Petri Net structure for the starting message from the Propose protocol. The Petri Net consists of the states at either end of the propose message and the propose message as a message place that
Figure 4.3: Simple translation of a message to the equivalent Petri Net structure.

acts as the input to the Petri Net.

The Petri Net is initialised when a start message is received by the debugger. We simulate the messages by placing tokens on the state place that match the incoming message. However, we must first place a token on the state place that is partnered with the incoming message. For the first message this is the special state place that we refer to as the start state of the protocol. Stage 2 shows a token deposited onto the start state place.

Stage 3 shows the result of the \textit{propose} message being sent. A token is inserted into the Petri Net onto the \textit{propose} message place. Recalling the rules for firing transitions should make it clear that adding the token onto this state enables transition \textit{T1}. When the Petri Net fires, tokens will be removed from state \textit{S0} and message place \textit{propose} and a token will be added to state \textit{S1}. Hence, the \textit{propose} message acts as the trigger for transitioning from state place \textit{S0} to state place \textit{S1} via transition \textit{T1}.

The underlying intuition in connecting the places to the transitions in this way is that the logic to model the transition from one protocol state to another is developed into the structure of the Petri Net. It is the messages that cause a protocol to advance from one state to another. Furthermore, the current state of a protocol dictates which messages can be sent. This is important when we wish to use the Petri Nets to model conversations. In this example, from state \textit{S0} a \textit{propose} message will advance the protocol to state \textit{S1}, hence we assign the preset of transition \textit{T1} to be state \textit{S0}, and message place
propose; and the postset to be state $S_1$.

The Petri Net fragment from figure 4.3 illustrates how we perform the translation for the simplest AUML structure, a single message. We use this translation for any occurrence of a single message, using the state labels and message types from the AUML protocol to specify the labels for the Petri Net components. To enable the translation of more useful protocols we have developed a set of local transformation rules so that each of the AUML interaction operators has a generic equivalent Petri Net structure. By applying the transformation rules we arrive at a set of Petri Net fragments, that when unified, form the complete Petri Net protocol. These transformation rules are the subject of the following sections.

## 4.2 Translations for the Interaction Operators

### 4.2.1 Overview

In the following sections we provide rules for translating a number of the most useful AUML interaction operators. We provided translation rules for the Alternative, Optional, Parallel, Loop, Termination, and Sequencing operator. In addition to this we also provide translation rules for interleaved protocols and show how nested interaction operators are handled by our translation rules.

There are a few interaction operators that we either do not provide translations for, or we handle using some combination of the other translation rules. We discuss these operators and the details surrounding these omissions and design choices below.

### Weak Sequencing and Strict Sequencing

The purpose of these operators are to define the ordering expectations of messages arriving at an agent. The distinction between this sequencing and how normal sequencing occurs is subtle. In practice we don’t normally need to distinguish and have opted not to attempt to develop a solution to these operators.

---

1 Unification is what we refer to when adding Petri Net fragments together. Since the Petri Net component labels are required to be unique the preset and postset of duplicate components are merged together resulting in a connected Petri Net.
Negative

Describes which messages are invalid. We regard any message that has not been defined for an interaction to be invalid (as will be shown in chapter 5) and as such do not offer any specific translation rules.

Critical Region

Refers to defining regions such that the messages within are executed atomically and are not interleaved with other messages. Although this may be a desirable property to define we do not provide any means to control it in our translation rules.

Ignore/Consider

Refers to the set of messages that have to be considered or ignored. When a message is to be considered or ignored varies over time during a protocol. Handling this requires that we keep information about these conditions. As will be discussed in the section on loops, we do not have such information at our disposal when observing interactions. Furthermore, providing an automatic conversion for arbitrary conditions is not viable. As such we do not handle these operators.

Assertion

Defines that the sequence of messages that is sent is the only valid sequence. We hold this view for all interaction fragments and as such assertion is inherent in all our transformation rules.

Break

A break is simply a specialisation of an interleaved protocol. It allows the agent to stop what it is executing and then execute another protocol. This is typically used when an error has been detected signaling that the protocol should be terminated. The protocol should not be resumed after sending the messages inside the combined fragment. We can handle the break easily by treating it as an optional block which ends with a goto to the end of the protocol. This should be clear when we discuss the continuation interaction operator.
4.2.2 Alternative

The AUML alternative (or selection) interaction operator, shown in figure 4.4 is used to model the situation where an agent can select a message to send from a set of possible messages. From the current state there are two or more branches that can be taken. The protocol will advance to a subsequent state depending on which branch is taken (or which message is sent). In the presented example the Participant agent can choose between sending message a or message b.²

In the presented figure we have already applied the state labels after following the labeling procedure set out in the previous section. In this example the alternative interaction operator begins from state P. From this state the agent can send message a in which case the protocol will advance into state Q. Alternatively, the agent can send message b and the protocol will advance into state R.

The Petri Net version of the alternative fragment is presented alongside the AUML version. Since the agent can send either an a message or a b message from state P this state is common to both transition T1 and transition T2. From state P, if the agent sends message a the conversation advances to state Q hence both state place P and message place a are inputs to transition T1. That is, the preset of transition T1 consists of P and a. Similarly, the preset of transition T2 consists of the common state place P and the alternative message place b, which ensures that from state P if message b is sent then the conversation will advance to state R.

This translation generalises for alternative interaction fragments with more than two alternatives.²

²Alternative is presented as requiring zero or one of the regions being taken, however, the one case is more intuitive. The zero case can be represented by containing it within an Optional region.
In such a situation the common state, P will simply be input to all transitions that link to each of the message choices.

### 4.2.3 Parallelism

The parallelism interaction operator as shown on the left of figure 4.5 requires that both message c and message d are sent when state R is reached, but the order is not specified. The purpose of the parallel operator is to provide a protocol with the ability to split its communicative processes such that the agent can communicate on multiple topics or with multiple other agents within the same conversation. Within our Petri Net protocols a token is used to represent the state of a protocol, however, when parallelism is modeled there are multiple concurrent states that need to be modeled. Petri Nets naturally support this behaviour by allowing multiple tokens, as such, we use one token to represent each of the concurrent communication processes.

To support the parallel operator we have developed a transformation rule whereby we introduce extra tokens into the Petri Net for each of the required concurrent interactions. Before we go into the details of the transformation rules it is important to note that we cannot generate these extra tokens when the protocol advances to a state where it could enter a parallel region, but may also select a different path of execution. The reason this would not work is because the parallel region could be nested inside an alternative, or other similar structure. Since the parallel region may not be chosen
by the agent we cannot introduce new tokens until we know that the parallel option has been taken
by the agent.

This problem is essentially one of lookahead whereby it is not possible to know what path will
be taken in advance. The solution is to develop a Petri Net structure that only generates new tokens
when it is known that the parallel interaction operator has been entered. We say that a parallel region
has been triggered when any of the initial messages inside the parallel region are sent. Once any of
these messages has been sent we can generate new tokens to represent each of the possible interaction
states.

We show the Petri Net translation for the parallel operator on the right hand side of figure 4.5. The
translation adds an intermediate state for each of the messages inside the parallel interaction operator,
namely \( R' \) and \( R'' \). These places are used to hold the extra token that is generated. When a message
triggers the Petri Net one token finishes on the state that the triggering message leads to, the other
moves into the intermediate state so that the remaining message can be sent.

The Petri Net works as follows: when in state \( R \) either a \( c \) or a \( d \) message is expected. If, for
example, a \( c \) messages is received then \( T3 \) will be enabled. Following the Petri Net rules, tokens
will be removed from place \( R \) and place \( c \), and a token will be deposited on each of \( R' \) and \( S \). The
conversation has now split into multiple states, since multiple tokens now exist in the Petri Net. After
receiving the \( c \) message there is still the need to receive the \( d \) message. When the \( d \) message is
received it is placed in the \( d \) place, since a token is on place \( R' \) transition \( T6 \) is enabled resulting in
a successful firing of the Petri Net and a token being deposited on place \( T \). It should be evident that
this Petri Net will also receive the opposite sequence of \( d \) followed by \( c \), in which case \( T4 \), \( R'' \) and
\( T5 \) will be utilised.

This translation generalises to more than two messages within the parallel region. If there were
another message, say \( e \) that was to be sent in parallel then all that would be needed would be to add
the necessary intermediate state and transition and make the necessary links between transitions. In
particular it is necessary that the first message received enables all of the intermediate transitions by
placing a token on each of the intermediate states. We do this by ensuring that the postset of the
upper level transitions (those at level \( T3 \) and \( T4 \) in the figure) consist of the intermediate state of all
messages other than the one that triggers the region. For example, if message \( c \) was sent we would
enable the intermediate states for message \( d \) and \( e \). In summary, for each message in the parallel
A conversation may not proceed beyond a parallel combined fragment until all of the messages inside that region have been sent. Therefore, if figure 4.5 had a message, say $z$, being sent from the Initiator to the Participant after the parallel region, then $z$ can only be sent after both $c$ and $d$ have been sent. Furthermore, after the parallel combined fragment has terminated the conversation no longer has multiple paths of execution. Therefore the tokens from each of the final state places from the parallel region need to be collected into a single intermediate state. When the end of a parallel region is reached the tokens from the final state places of each of the parallel blocks are consolidated. Figure 4.6 shows how the merging of paths is applied. A single transition takes input from the final states, $S$ and $T$, 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. Instead of using places $S$ or $T$ as the starting state for messages sent after the parallel combined fragment we use the combined state $ST$.

### 4.2.4 Optional

The optional interaction operator, shown in figure 4.7, specifies that everything within the optional box may or may not be executed. Thus, when the protocol is in state $T$, message $e$ can be sent leading to state $U$, which is in turn followed by message $f$ leading to state $V$. Alternatively, the agent can forego sending message $e$ and proceed immediately to send message $f$ which would directly advance the protocol to state $V$. We capture this logic by a labeling that indicates that the start state for message $f$ can be either $T$ or $U$ and then applying the transformation rule for each of the single messages. The
The bottom half of figure 4.7 shows the result of this labeling and subsequent translation. Importantly, state $T$ appears in the preset of a transition from two different Petri Net fragments.

The general procedure for labeling an optional block is thus to ensure that the initial state of the optional block is added to the state list of any message that can be sent after the optional block ends. In doing this the tail of some messages will have more than one state assigned to them. It is necessary to perform the translations once for each of the states in the state list. In the presented example message $f$ has both state $T$ and state $U$ in its state list, hence, we performed the standard translation for message $f$ twice.

We have provided an intermediate view of the resulting Petri Net immediately below the AUML optional example in figure 4.7. The three Petri Net fragments represent the three possible message exchanges in the original example. After each of the fragments have been created state places may appear in multiple Petri Net fragments. This is only because of the way we present the Petri Net in its graphical format. Since we are dealing with sets of places each state place only actually appears once in the Petri Net, as shown in figure 4.8.

In our example this means that the postset of state place $T$ consists of both $T7$ and $T8$ and the postset of message $f$ consist of both $T8$ and $T9$. This allows place $V$ to receive a token by combining message place $f$ with either state place $T$ via transition $T8$, or with state place $U$ via transition $T9$. The legal message sequences are thus: $\langle e,f \rangle$ or $\langle f \rangle$. 
4.2.5 Loops

The loop interaction operator is used to enable iteration over an interaction block. When the end of the interaction block is reached it is possible to begin execution again. The example of figure 4.9 shows an AUML interaction protocol that allows the sending of message sequence \( \langle g,h \rangle \) multiple times, and possibly zero times.

Below the AUML fragment are the equivalent Petri Net fragments that support the desired behaviour of the loop interaction operator. Inside a loop interaction operator all end states should enable the sending of any of the initial messages at the beginning of the loop interaction operator. To develop the correct Petri Net structures we first label the AUML interaction protocol. We identify the final state labels from within the loop interaction operator and append them to the state labels for the initial interaction occurrences of the loop operator. Following this labeling we perform the standard transformations. When the transformation of interaction occurrences to Petri Net fragments is performed these additional state labels generate additional state transitions that simulate the semantics of the loop operator.

In our example the start state is state \( V \) and the end state is state \( X \). Initially the identified state labels on the AUML example protocol for message \( g \) are state \( V \) at the tail and state \( W \) at the head. The loop interaction operator allows the \textit{Initiator} agent to restart the combined fragment with message \( g \) when state \( X \) is reached. To enable this we append the final state label(s) to the tail of any first possible message in the loop combined fragment. We thus append state \( X \) to the state label at the
tail of message $g$ and arrive at the final state labeling as shown in figure 4.9. With this labeling of states on the AUML protocol we perform the standard transformations to the Petri Net fragments. These fragments can be seen directly below the AUML protocol and as was the case for the optional interaction operator we have a set of disconnected fragments. The Petri Net fragments are unified and we arrive at the final Petri Net of figure 4.10. This Petri Net correctly implements the semantics of the AUML loop operator: after sending message $h$ the protocol will be in state $X$ and a token will reside on this state. From this state it is possible to restart the loop by re-sending message $g$, in which case transition $T_{11}$ will be enabled and after firing the Petri Net the conversation will advance to state $W$.

There are some aspects of the AUML loop specification which are not handled by our translation. Firstly our translation results in the requirement that the loop be executed at least once. If it is required that the loop be executed zero or more times then the loop should be nested inside an alternative fragment. Secondly, we do not consider conditions encoded in guard conditions. Guard conditions are logical conditions that can be specified to constrain when a loop should be executed, however, the information necessary to evaluate a guard condition at run-time is typically internal to the agent, and not communicated in the messages. As such it is usually not possible to evaluate guard conditions and we do not consider them in our transformation rules. Finally we do not consider loop conditions specifying the interval over which a loop should be executed, such as Loop[$3,5$] (meaning execute the loop between 3 and 5 times.)

We now present another example of how our loop transformation supports iteration over other interaction fragments, such as the alternative interaction operator. It is often the case that nested operators are encountered so we show how our transformation rules apply when such a nesting is to be transformed. Figure 4.11 depicts the situation where the Participant agent has the option of sending either message $x$ or message $y$, and since this choice is inside the loop the choice can be revisited multiple times. To translate this protocol we follow the standard rules for transforming a loop: we add state labels to the AUML protocol, and then apply the required transformations based on the interaction operators.

The loop structure requires that for each final state that the protocol can be in before exiting the loop, the message(s) that begin the loop need to be able to be sent from each of these states. To ensure this we add the state labels from the end states ($Q$ and $R$) to the start of sending either message.
$\text{Figure 4.11: Alternative inside a loop with the Petri Net fragments after translation.}$

$x$ or message $y$. Hence, the final label markings for this figure are message $x$: $P, Q, R$ and message $y$: $P, Q, R$. We generate an alternative Petri Net fragment for each of the states at the tail of message $x$ and $y$. This means that states $P, Q$ and $R$ will be the common state places for three different alternative Petri Net blocks as shown in figure 4.11 below the AUML protocol. After unifying the Petri Net fragments we arrive at the final Petri Net protocol shown in figure 4.12. Even though this Petri Net looks complicated, the places have simply been unified, requiring no special effort.

4.2.6 Continuation

The continuation interaction operator acts as a goto statement to enable an agent to return to a previous part of the protocol, or to jump forward. A matching continuation label is used to indicate where the interaction should continue from. When an outgoing continuation label of $x$ is reached the protocol will continue at the incoming continuation label of $x$.

In figure 4.13 we show that once the protocol reaches state $R$ at the start of the alt fragment, the agent has the choice of either sending message $z$, which would lead the protocol into state $S$ or the agent can take the continue option and return to the state of being ready to send message $y$ again. The translation for the continuation interaction operator is similar to that described for the loop operator. We add the necessary state labels so all possible paths of execution are generated when we perform
The idea for labeling the AUML protocol to cater for the continuation operator is as follows. If there is a continuation that specifies that from state $R$ the protocol can continue to some other state, $Q$. Then from state $R$ you can reach everything that state $Q$ could reach. To enable this we need to generate the same Petri Net structures from state $R$ as were generated from state $Q$. This is done by applying the label before the outgoing continuation (state $R$) to the state after the incoming continuation label (state $Q$).

In figure 4.13 the result is to add state $R$ to the tail of message $y$. Next to this figure is the Petri Net protocol for this continuation example after the translations have been applied. Notice that when in state $R$ if the continuation operator is used then receiving message $y$ will return the protocol back to state $R$. Since it is optional to take the continuation, the protocol can also advance from state $R$ to state $S$ if message $z$ is instead received.

### 4.2.7 Termination

The termination operator, or stop operator as it is sometimes referred to, is used to terminate interaction of a particular path of execution. Figure 4.14 is an example of using the stop operator. If message
$x$ is sent, the stop operator is reached and the execution for the path ends. However, if message $y$ is sent the protocol can continue on after the alternative interaction fragment.

The termination operator is handled at the labeling stage. Ordinarily both state $Q$ and state $R$ would be added after the alternative interaction operator completes so that execution can continue from either state $Q$ or state $R$. However, the stop operator requires that no execution continues if state $Q$ is reached. The solution we have adopted is to not propagate the state label that precedes the stop operator. By doing this we do not use the state before the stop operator as a source for any transformations. The result is that the state after the stop operator becomes a final state in the protocol, it has no postset.

It is useful to be able to identify the final states in the Petri Net protocol when it comes time to make assumptions about the state of a conversation. In the Petri Net version of the protocol we can determine if a state is a termination (or final) state because any non-message place that has no postset must be a final state. (For $x \in P$, $x$ is a termination state place iff, $x \bullet = \emptyset \land x \notin MP$). It should be noted that the second conjunct is not strictly necessary since all message places have a non-empty postset.

It should also be noted that given that state $R$ represents the alternative option it will still be an active state for any further messages that may appear after it.
4.3 Problematic AUML Protocol Specifications

AUML notation allows for a flexible specification of protocols such that the same interaction sequence can be specified in multiple ways. There are certain types of specifications that, although they are legal AUML, cause problems for our transformation rules. Of particular concern is certain uses of duplicate messages. If the same message type can be sent from a unique state but leads to different states our transformation rules will not work. The simplest example of this is shown in figure 4.15. In this example protocol the participant agent has the choice of sending message $x$ followed by message $y$ or sending message $x$ followed by message $z$. The state before sending message $x$ in either of the choices is state $P$. Depending on which choice is taken the protocol is expected to transition to a different state, either state $Q$ or state $S$. However, it is not possible, from an observer’s point of view, to identify which of the paths is being taken and as such our translation rules will not work if such a protocol is translated.

We have addressed this by developing guidelines for how the AUML protocol should be modified so that the meaning of the protocol is retained but the problematic state transformations no longer exist. To do this we must identify the possible interaction sequences and determine a new way of specifying them. In the example of figure 4.15, there are two possible interaction sequences. If the first alternative path is taken we will have an $x$, $y$ sequence. And if the second choice is taken we will have an $x$, $z$ sequence. In both of these examples message $x$ is sent first and the choice is really between sending message $y$ next or message $z$ next. Therefore, an alternative way of representing this protocol is to remove message $x$ from the alternative box (since it will be sent regardless of which
path is taken) and place it outside the alternative interaction fragment.

This modified version of the protocol is shown in figure 4.16. This protocol allows for the same interaction sequences but no longer has the problem with the state transitions, now the choice only occurs at state $Q$. We follow this procedure of identifying problematic AUML syntax and providing an alternative method of specifying the AUML for each of the interaction operators that we have considered in this thesis. In order to use our approach it is necessary to identify any problematic protocol definitions and apply the procedure to modify the protocol before the transformation of an AUML protocol to the Petri Net protocol is done.

4.3.1 Optional

When an optional interaction operator is used in a protocol we consider the protocol state immediately before the optional combined fragment to be the same as the state immediately after the optional combined fragment if the optional combined fragment is not taken. Therefore, if the first message inside an optional combined fragment is the same as the first message that appears after an optional combined fragment ends, as is shown in figure 4.17, we must modify the AUML protocol before performing the conversation. In this example, message $a$ can be sent from state $P$ leading to state $Q$ by taking the optional block. Or if the optional block is not taken then message $a$ can be sent from state $P$ but this time leading into state $S$. As we did for the alternative interaction operator we provide alternative AUML to represent the same interaction requirements.

We identify that there are two possible message sequences for this interaction, $\langle a, b, a \rangle$ or $\langle a \rangle$. We can represent this as $a$ followed by an optional block containing $b, a$. Figure 4.18 is the resulting
4.3.2 Loop

The nature and resolution of the problematic use of the loop operator is the same as for the optional interaction operator. We have shown the original problematic AUML in figure 4.19. The process we use to label a protocol that includes a loop structure results in a labeling that can not be transformed. The first $a$ message in the protocol is initially labeled as being sent from state $P$. However, once the loop structure is traversed we must also add state $R$ as a possible state from which this initial $a$ message can be sent (see section 4.2.5). This labeling results in a duplication of transitions from state $R$ via an $a$ message in which each transition leads to a different state.

We note that the *initiator* agent can send any number of the sequence $a,b$, finally terminated by a single $a$. An example of a legal message sequence for this protocol would be $a,b,a,b,a$. Generating a suitable equivalent AUML protocol can also be done by using the same rules for resolving the optional interaction operator.

Figure 4.20 is an equivalent way of representing the problematic loop structure. The duplicate message inside the loop block is removed and placed immediately before the loop and the duplicate message that is found after the loop block is moved to the end of the loop block. This version of the protocol supports the generation of the same message sequences but can now be readily converted by our translation algorithm.
4.3.3 Parallel

When the parallel operator is used in the same problematic way as in the previous examples we encounter some difficulties in providing a modification to the AUML protocol. Figure 4.21 is an example of such use where message $x$ is the start message for both of the parallel interaction paths. In considering the meaning of this interaction protocol we note that it is an unusual use of the parallel operator. The parallel operator specifies that both interaction paths should be executed at the same time. In the context of this example it means that the participant agent sends message $x$ to the initiator agent twice, in parallel.

The difficulty with translating the parallel operator when it is used in this way arises because when the first $x$ message is received we need to generate a new path of execution in the Petri Net protocol so that the other parallel path can be taken. The problem is that we do not know which path the original $x$ message belongs to. Therefore, we are unable to determine which of the $x$ messages has been sent and which one we are still waiting for.

A protocol of this form requires that four messages be sent in total. For each of the parallel blocks message $x$ starts the interaction and the possible message sequences are; (a) $x, x, y, z$ (b) $x, x, z, y$, (c) $x, y, x, z$ (d) $x, z, x, y$. We are unable to apply the same technique as we did for the alternative interaction operator since even after we extract the first $x$ message we still have a conflicting situation with sequences (a) and (b). In fact, we are unable to derive a suitable equivalent AUML protocol structure.

![Figure 4.21: Problematic use of the parallel operator](image-url)
if the parallel operator is used in this way. This means that if we encounter such use we are unable to provide a translation to a Petri Net protocol.

4.4 Interleaved Protocols

The AUML specification supports the reuse of protocols by a process called interleaving. An interleaved protocol is one that is to be executed from within the confines of another protocol. When an interleaved protocol is reached the outer protocol is suspended until the interleaved protocol completes. Upon completion the outer protocol will continue from the termination state of the interleaved protocol. Figure 4.22 shows the Payment protocol being interleaved inside the Request Information protocol. In this example the Payment protocol should be executed after the send-info message is sent.

Translating this behaviour into the Petri Net model is achieved by adding the Petri Net version of the interleaved protocol into the Petri Net of the outer protocol. Any messages that appear in both protocols will be unified, however, when the interleaved protocol is added to the outer protocol it will be disjoint from the Petri Net because there will be no transitions from a state from the outer protocol leading to a state in the inner protocol.

The correct behaviour that we need to ensure is that the appropriate state of the outer protocol leads into the start state of the interleaved protocol. We do this by merging the state immediately prior to reaching the interleaved protocol with the start state of the interleaved protocol. The easiest
Figure 4.23: AUML Payment protocol and Petri Net equivalent

Figure 4.24: Combined version of the Request Information protocol

way to support this is to ensure both states share the same name.

Figure 4.23 shows the interleaved payment AUML protocol along side the Petri Net version of this protocol. Before adding this protocol to the Request Information protocol we change the name of the start state from state P to the name of the appropriate state in the Request Information protocol, which is state R. Finally, it should be noted that if there were any messages sent after the interleaved protocol ended then the end state of the interleaved protocol would form the start state for the next valid message of the outer protocol. Figure 4.24 shows the result of this translation.

Since we are merging protocols together that have been developed independent of each other it may appear that conflicts could arise in terms of the labelling of place names and transition names. This would be likely if we used the labeling method shown in our examples, i.e. P, Q or T1, T2, etc. Labels are actually required to be qualified with the name of the protocol that they belong to in order to protect against such conflicts. We have presented this shorthand version of the label names in our discussions to avoid cluttering of the figures and the explanations.

Note we have not shown the Petri Net for the Request Information protocol but it should be clear that the final state before entering the interleaved protocol will be state R.
A complication arises when a protocol could be in one of several possible states when an interleaved protocol is to be executed. For example, if the interleaved protocol follows from an alternative interaction fragment. Figure 4.25 depicts an alternative Request Information protocol. In this version of the protocol the InfoServer agent can receive requests for information that it does not hold but knows where the information can be retrieved from. Upon receiving a request, the InfoServer agent has the option of sending the information via a send-info message or it can send the location of where to collect the information from via a send-details message.

After this message exchange, the Customer agent is expected to pay for the service. Since the Payment protocol occurs outside the ALT interaction fragment it is the intention that it be executed no matter what state the previous interaction fragment was left in. Both state R and state S are potential protocol states from which the Payment protocol can be executed. Thus, for every state that the protocol can be in prior to the appearance of an interleaved protocol, the interleaved protocol must be executable from those states. In terms of making the transformation this means that we will need to generate the payment protocol twice, and have the protocol starting from state R and from state S. The reason for this duplication is that state R will not always equal state S (although in this example they are equal).
CHAPTER 4. TRANSLATING AUML PROTOCOLS TO PETRI NETS

This is shown diagrammatically in figure 4.26. The Payment protocol is added to the outer Petri Net protocol twice, one for each of state $R$ and $S$. When adding the interleaved protocol to the outer Petri Net we need to further qualify the Petri Net places, and transitions with version numbers. This is because each copy of the interleaved protocol is distinct and we do not want to incorrectly merge these distinct places. The left most protocol is qualified with version 1 and the right most protocol is qualified with version 2. Following the addition of the protocols, each of the Payment protocols needs to be merged with its respective start place, the merge point is shown on the figure. This figure is an intermediate version of the complete protocol, the message places, send-payment and ack-receipt need to be merged, this is straight-forward and is not shown.
4.5 Protocol Translation Examples

4.5.1 Translating a FIPA AUML Protocol

Having described how to convert different cases of AUML syntax to our Petri Net representation we now illustrate these processes by converting one of the standard FIPA AUML protocols.

Figure 4.27 shows the Request When protocol with labelling applied. With the initial identification of states complete we move onto identifying each of the interaction operators used and perform the required translation steps. We translate the first message, the request message, sent from the initiator agent to the participant agent.

Following the request message we encounter the first alternative block. From this block the agent has two choices, the request can be rejected by sending a refuse message or the request can be granted by sending an accept message. We apply the translation for the alternative interaction operator and arrive at a Petri Net fragment that has state B at its head and states C and D at the end of the fragment.

The large X underneath state C in the AUML protocol (figure 4.27) is AUML notation that indicates that the protocol should terminate if execution reaches this point in the protocol. As discussed
in section 4.2.7 after reaching the stop operator we no longer propagate the state label immediately prior to the stop operator. In this example we do not propagate label $C$. In addition to this we also wish to know what states are termination points for the protocol so we know when a protocol has reached a valid termination state.

In this example, the only valid state from which to continue is state $D$. This state is connected with the remaining part of the Petri Net protocol. We translate the final alternative fragment, which has three alternative messages; inform-done, inform-result and failure. Any one of these three messages can be sent when the protocol reaches state $D$, and depending on which is chosen the protocol will terminate at state $E,F$ or $G$. See figure 4.28.
4.5.2 Translating a Complex AUML Protocol

Figure 4.29 is an example of translating a complete protocol which is an amalgamation of a number of the interaction operators that we have discussed in this chapter. This example is an artificial one created to illustrate how a complex case is still handled easily. This is a more complicated protocol, however, the conversion is straightforward and is simply a matter of identifying the interaction operators and applying the translation rules.

The resulting Petri Net protocol is shown in figure 4.30. It should be clear that this protocol is made up of the same components that we introduced when describing each of the translation rules earlier. They have simply been combined together to form this complete Petri Net protocol.

In the AUML protocol we have message \( g \) appearing inside a loop interaction operator after an alternative interaction operator. Therefore, message \( g \) can be sent from three different states. In the resulting Petri Net protocol the part of the protocol that covers state \( g \) may be a little difficult to
follow. However, this is only because the Petri Net is presented in its graphical form. The procedure for generating the Petri Net is as simple as generating the structure for message $g$, once for each of the three start states: State $Q$ if message $a$ is chosen, state $V$ if message $b$ (and then all subsequent messages leading to state $V$) is chosen and finally state $W$ to cater for looping around state $W$. 
Figure 4.30: Petri Net equivalent of the combined AUML protocol
Chapter 5

Monitoring and Reporting on Interactions

In this chapter we discuss how we use derived Petri Net protocols to monitor and provide useful information about the interactions occurring between agents to support the debugging of multi-agent systems. We present the Interaction Monitor: an instance of the abstract Monitoring component discussed in section 3.3. The Interaction Monitor is responsible for processing messages that are exchanged within the system by modeling the messages as conversations within the Petri Net protocols. In addition to the mechanics of the Petri Nets, the Interaction Monitor utilises a number of other processes and algorithms to determine the correctness of the messages exchanged in the monitored system.

Following this, we describe how the Interaction Monitor detects the bugs types which we have identified. Each of the bug types that were introduced in section 3.4 are discussed. Reporting the results of the Interaction Monitor is carried out by a reporting interface that we have developed as part of our Debugging Toolkit. The reporting interface is a basic prototype tool that we use to convey the results of the Interaction Monitor to the developer.

5.1 Overview of the monitoring process

Modeling and processing conversations occurs within the debugging framework introduced in section 3.3. Figure 5.1 shows an abstract view of the the Interaction Monitor which is responsible for
debugging agent interactions in a deployed multi-agent system. The Interaction Monitor is responsible for processing messages within the Petri Net protocols for the purpose of detecting erroneous interactions. It is composed of a message queue that is used to store the incoming messages while an existing message is being processed and a Conversation List which contains the set of active conversations. Messages are removed from the message queue and partitioned into conversations based on the conversation id that is included with each message.

A conversation id uniquely identifies each conversation within the multi-agent system. The need for a conversation id is indicated in FIPA (2003) proposed standard, explains:

“Note that by their nature, agents can engage in multiple dialogues, perhaps with different agents, simultaneously. The term *conversation* is used to denote a particular instance
of such a dialogue. Thus, the agent may be concurrently engaged in multiple conversations, . . .”

The partitioning of messages into conversations greatly increases the ability to manage the often large number of messages exchanged within a multi-agent system. A conversation id facilitates this partitioning and enables us to consider messages that belong to one conversation in isolation from all other messages that do not belong to the conversation. There is, therefore, a natural reduction in complexity in modeling the message exchanges. Unlike distributed debugging, where a global event graph is used to model the relationships between events that occur in the system, we have a set of concurrent yet disjoint interactions that can be considered in isolation.

In dealing with the distribution of messages into the relevant conversations we have the Interaction Monitor maintain a list of active conversations based on a unique conversation id. When a message is removed from the queue the conversation id is inspected. The first step in processing the message is to determine which conversation it belongs to. If the conversation id matches any active conversations then the message is directed to the respective conversation. If, however, the conversation id is not matched to a conversation a new conversation is initialised.

We construct a conversation list which is used to hold all of the active conversations occurring in the multi-agent system. Each conversation is comprised of a conversation id, a Possible Protocol List (PPL), and a role map which is used to map agent instances with role types and is used to reason about role related bugs. The PPL contains an instantiation of each of the protocols that could be the protocol that the agents are using to direct their conversation. We need the PPL because we do not require that the agents include the name of the protocol when sending messages to one another, and take the view that each protocol in the PPL is potentially the protocol that the agents are following. When a message sequence violates a protocol we mark it as in error and continue to process any other remaining protocols.

Initially the conversation list is empty. Upon removing a message from the message queue a conversation is created and the message is modeled in the conversation. If a conversation already exists, because the currently dequeued message shares the same conversation id as a previously received message, then the message is added to that conversation where it is added to and processed in each of the protocols in the PPL. The message is simulated as a token in the Petri Nets and the Petri Nets are fired to enable any valid transitions. The protocols are then checked for errors and if necessary
the user is alerted.

The following summarises the monitoring procedure that we use to detect protocol related errors. The details for each of the steps are covered in the subsequent sections and an example of a valid conversation is provided before we provide details for the error detection and reporting steps.

1 intercept a message;
2 add message to queue;
3 while message queue is not empty do
4 remove message from head of queue;
5 if message does not belong to an existing conversation then
6 initialise a new conversation;
7 end
8 add message to each protocol in the PPL for the relevant conversation and fire Petri Nets;
9 check for errors;
10 report errors if necessary;
end

5.2 Intercepting Messages (steps 1 and 2)

The Interaction Monitor needs to receive a copy of every message that is transmitted in the system that it monitors. There are various ways in which this can be accomplished. In their work on modeling teams Kaminka et al. refer to an approach whereby a listening process is introduced to eavesdrop on the communication medium used (Kaminka, Pynadath & Tambe 2000). A drawback to such an approach is that if the sending parties are not aware of the presence of another listening party then there is no way to guarantee that a message that is sent onto the communication medium is picked up by the listening process. Indeed, handling such loss of messages is a key aspect of some of their work (Kaminka, Pynadath & Tambe 2001). We are, however, very interested in ensuring that for every message that is transmitted in the system being monitored that a copy is received by our monitoring device. As such instead of passively intercepting messages we require that a carbon copy of any message exchanged is also sent to the Interaction Monitor.

The mechanism for sending a copy of the message to the Interaction Monitor will be platform dependent. However, the technique of overloading which we use in our JACK prototype should be
generally applicable. We overload the JACK send method so that each time it is invoked a message is also sent to the Interaction Monitor. This approach does not intrude on the developer, nor does it complicate the source code that the developer works with.

The Message Queue is used to store the messages being intercepted by the Interaction Monitor. The Message Queue is a simple FIFO message queue which processes messages in the order that they are received. It is important that if a message is already being processed that any new messages must wait until the current message has been completely processed.

### 5.3 Initialising a Conversation (step 6)

Initialising a new conversation is primarily concerned with identifying and setting up the Petri Net protocols so that the conversation can be modeled. We note that any given message may require the initialisation of multiple interaction protocols, depending on the number of protocols for which the start message matches. As indicated earlier, a given message type may appear in a number of protocols and even the start message of a protocol is not guaranteed to be unique. For example, both the Contract Net protocol and the Iterated Contract Net protocol have a $cfp$ message as their starting message. If the Interaction Monitor were to intercept the $cfp$ message then it would need to add both these protocols to the PPL and initialise each protocol so that the conversation can be modeled in each.

The process for creating a new conversation involves searching the Protocol Library for all correct protocols that could possibly be used to model the conversation. The message type from the dequeued message $m$ is used as the search criteria. The protocols that will be used to model the conversation will be the protocols that have an initial message that matches the message type of $m$. We define an initial message as any message from the Protocol Library that has a postset in common with the start state (recall that the start state is the only non-message state that has an empty preset). A message $m$ is a valid initial message for a given protocol $P$, if the start state place of $P$ shares at least one outgoing transition with $m$. That is, if the intersection of the postset for the two places is not null. Formally, let $x$ be the start state place for $m \in MP$ (message places),

$$x \bullet \cap m \bullet \neq \emptyset$$

then $m$ is a valid initial message.
Identifying the initial messages for each protocol can be computed in advance, reducing identifying the initial messages to a simple comparison operation at run-time. It should also be noted that for a given protocol there could be multiple messages that are valid initial messages. One reason that this could occur would be if the protocol started with an Alternative interaction operator.

The Interaction Monitor scans the Protocol Library for the protocols that match the start message of the dequeued message. When a match is found a copy of the Petri Net protocol is made and added to the Possible Protocol List so the conversation can be modeled. A token is placed on the start place to indicate that the protocol is in the starting state. Once this has been completed for all relevant protocols the message itself can be modeled in each of the protocols by placing a token on the appropriate message place and firing the Petri Net.

During the initialisation phase we also assign agent instances to roles for each of the protocols in the PPL. The protocols are specified with role types and each role type is associated with a set of messages that it can send and a set of messages that it can receive. By mapping agent instances to roles in this way we are then in a position to identify certain role related errors.

Roles are assigned to agent instances based on the contents of the header of the initial message that triggered the creation of a new conversation. The sending agent, whose name appears in the sender field of the message header, is mapped to the role that triggers the conversation. The recipient of the message, whose name appears in the recipient field, is mapped to the role that receives the message in the protocol. Once roles have been mapped to agent instances any message that appears in the conversation will only be considered valid if it first matches the role mapping. We consider the information relating to both the sender and the recipient of a message when performing the role checks. In the event that there are more than two roles in a protocol the role map will not be completely assigned after the first message is sent. Subsequent messages will be used to assign any unmapped roles.

To illustrate the role mapping consider the situation where the Interaction Monitor receives a propose message that was sent from agent A to agent B. In this example there is only one protocol that matches the propose message. The messages that are valid for the initiator are:

\[ \text{initiator} \Rightarrow \text{propose} \]

The messages that are valid for the participant are:

\[ \text{participant} \Rightarrow \text{accept, refuse, failure, inform-done, inform-result} \]
Since we are in the initialisation phase we simply map the agents to the roles for this protocol. Agent A is mapped to the initiator role and agent B is mapped to the participant role. For all subsequent messages that are exchanged in this conversation the role map is queried to verify that the sender and receiver fields in the message header match those that have been defined in the role mapping. Or if roles are appearing for the first time the agent instances are mapped and added to the role map.

The role mapping procedure supports the mapping of agent instances to multiple roles within the same conversation. This is achieved by allowing a one to many association in the mapping procedure which allows agent instances to play multiple roles. Since role mappings are done at the conversation level there is no restriction placed on which roles an agent plays in different conversations. For example, an agent can play an initiator in one conversation and a participant in another.

From the role maps we can identify a number of role related error conditions which we will describe when we discuss the specific error types that the Interaction Monitor can identify.

5.4 Add Message and Fire Petri Nets (step 8)

After a conversation has been initialised, and the PPL contains at least one protocol, the currently dequeued message is added to each of the Petri Nets in the PPL. The message place matching the dequeued message is identified and a token is created and deposited onto the place. The Petri Net is fired to allow for any enabled transition to fire and the tokens are removed and deposited in the standard method for Petri Nets.

The addition of a token onto the message place in the Petri Net protocol can result in more than a single transition firing. For example, the merge operation after exiting a parallel region will result in a second transition firing to merge the tokens into a single state place. As such, we continue to fire the Petri Net until it is no longer live, meaning that there no longer exists any enabled transition\(^1\). With the requirement that a token is only generated on a message place when the Petri Net is no longer live we ensure that any possible transitions from one state to another are completed before any other message can enter the Petri Net.

Messages will be processed inside the Petri Net in this manner until either an error is detected (which we will discuss in section 5.5) or the Petri Net protocol terminates successfully. A conversa-

\(^1\)There are other definitions of liveness, however, this is suitable for our algorithm.
tion is said to have successfully terminated if all tokens in the net reside on a final state place. A final state place is any state place that is not an input to any transition, that is, the postset of the place is empty (Formally, \( x \) is a final place iff \( x\bullet = \emptyset \)). When one of the valid protocols terminates successfully the conversation is marked as successfully terminating indicating that no errors were detected in the protocol. Given that the PPL may contain other currently valid protocols it is possible that after one protocol terminates successfully there are other protocols that are not in error but also not complete. Any further messages received in a conversation that has an already valid protocol will still be processed inside the other relevant protocols, in the event that one of these protocols results in an error this will be indicated to the user.

### 5.4.1 Example of a Valid Conversation

The remaining steps in the monitoring procedure are concerned with the detection and reporting of errors. Before we discuss these errors and how they are detected we wish to present an example of a valid conversation to demonstrate how the Petri Net processes messages until the conversation successfully terminates.

Figure 5.2 shows how messages sent by agents engaging in a conversation following the Request protocol are modeled in the Petri Net version of the protocol. Figure (a) depicts the initial marking of the Petri Net after it has been initialised and the first message has been deposited on the request message place. This initial marking immediately enables the first transition, \( T1 \). Transition \( T1 \) fires, and as shown in figure (b) tokens are removed from state place \( A \) and message place request and a token placed on state place \( B \).

From this new state, either a refuse or an accept message is valid. Receiving either message will result in a token being generated and placed on the corresponding message place, causing either transition \( T2 \) or transition \( T3 \) to fire. Thus, receipt of the initial message initialises the conversation and advances the conversation into the state of receiving the next message.

When the next message in the conversation, the accept message, is received by the Interaction Monitor and dequeued it is added to the accept message place. Receipt of this message is shown in figure (c). Since this is a valid message Transition \( T3 \) is enabled and after the Petri Net is fired there remains a single token in the Petri Net on state place \( D \), as seen in figure (d). Finally, receipt of an inform-result message results in figure (e), which after firing the Petri Net transitions into the final
Figure 5.2: Modeling a conversation on the FIPA request protocol.
marking shown in figure (f). figure (f) shows a token deposited on state place $E$. Given that state place $E$ is one of the valid termination states this conversation is marked as successfully terminating (other valid termination states are $F$ and $G$).

5.5 Identifying Erroneous Interactions (step 9)

The reason for processing messages inside Petri Nets is to identify erroneous situations. In the previous example the conversation completed successfully, hence no errors were detected. However, if for example the wrong message was sent we would want to identify this and report it accordingly. Identifying information such as this is precisely how we use the Petri Net models of agent interactions to assist in debugging interactions. We can determine the state of a Petri Net protocol by considering the distribution of tokens over places. Inspection of the Petri Net can indicate, among other things, the current state of the conversation and the next valid message. This is an important property that we leverage to help identify errors, and provide useful debugging information. The various methods that we employ to identify errors will be the subject of this section.

5.5.1 Sending the Wrong Message

Sending the wrong message is characterised by an agent sending a message that is not valid. There are two main cases that we focus on: (A) the message not being valid in the protocol at all, and (B) the message not being valid at the current time (because of the previous messages that have been sent). Case (A) is detected by identifying that the message does not match any messages in any of the protocols in the PPL. Case (B) is identified when the message does not successfully fire any transitions in any protocol.

**Message not valid at all (A)**

Each protocol has a set of valid messages for that protocol. When we perform our transformation from AUML interaction protocol to the Petri Net version we create a convenient list specifying the messages that are valid for the protocol. During run-time messages are delivered to their respective conversations and are to be modeled in the Petri Net protocol. However, a message is only processed in a Petri Net protocol if the message place that matches the name of the message can be found.
For each protocol in the Possible Protocol List of the conversation for which the received message belongs we check the valid message list. If the message does not match any of the message names in the list then the message is not valid in the protocol at all. There is no way that this message can be added into the protocol since messages are inserted into the protocol via the matching message places. Therefore, if it is found that a message does not belong to a protocol the protocol is immediately in error.

When such an error is identified we mark that protocol as being in error and record relevant details about the error to enable reporting of the error if necessary. The reason we do not immediately report an error is because there may be multiple possible protocols contained in the Possible Protocols List.

As briefly explained earlier in this chapter, while there are multiple protocols in the Possible Protocols List the Interaction Monitor does not know which protocol the agents are engaging in. It determines this over time by processing the messages in each of the protocols. As protocols are invalidated due to an error, such as sending a message that does not belong to the protocol, they are marked as being in error and as long as there is at least one valid protocol remaining then there is no need to alert the user to an error.

The Possible Protocol List has an impact on how and when errors and other status information is reported. Each time the debugger receives a message for a specific conversation an analysis is done on each protocol within the Possible Protocols List to identify error situations. As long as there is a non-erroneous protocol in the Possible Protocols List, errors simply lead to the conclusion that the protocol with the error was not in fact the one that was being followed, and it is discarded from the list. Only when the Possible Protocol List contains no valid protocols can it be concluded that the interaction is not valid for any of the protocols in the Protocol Library.

The principle of marking error states for each protocol, but noting an actual error only when an error state is detected on the last protocol is followed for all errors. Note also that when error states of other protocols are detected in the same iteration as the final protocol in the PPL these are also reported as possible causes of the error.

A final comment concerning the Possible Protocols List is that in some implementations it would not be necessary. For example, the proposed FIPA standard for message content suggests the inclusion of a protocol name in the message header so that agents explicitly know which protocol they should be following. If the protocol name was to be included along with the message then we would
not need a list of possible protocols, but would instead store the single protocol being used.

**Message not currently valid (B)**

We now consider the case where the message is valid in the protocol but is not valid given the current state of the protocol. For this discussion we return the reader’s attention to the example in section 5.4 (figure 5.2), where we described modeling a valid conversation based on the request protocol.

We begin this discussion starting from the protocol state represented by figure 5.2(d). At this point the participant agent has agreed to perform the required task and the protocol is currently in state D. Consider what happens if the next message sent and subsequently received into the Petri Net is a refuse message. A token will be generated and the refuse message place will be located and a token deposited onto it. When the Petri Net fires no transition is enabled. For a transition to fire there would need to be a token on state place B, which would enable transition T2. The conversation has already transitioned beyond this state, hence, the addition of the message does not trigger any transitions.

We use the following logic to determine that this behaviour indicates an error in the conversation: When a token is deposited onto a message place the token represents the receipt of a message. If the message is valid, based on the the current state of the conversation, a transition should be enabled to advance the protocol into a subsequent valid state. If no transition is enabled the message must not have been valid. Therefore, if after adding the message token and checking for enabled transitions (firing the net), a token still resides on a message place it can be concluded that the message was sent at the wrong time. As discussed previously the protocol is marked as in error but the conversation only reports an error if no error-free protocols remain in the PPL.

In addition to identifying the error we can also identify the set of valid messages based on the current marking of the Petri Net. We identify the valid messages by locating any message place that shares a transition in its post-set with any state place that contains a token.
1 SPT is the set of state places that contain tokens;
2 MP is the set of message places;
3 $V$ is the set of valid messages, initially set to $\emptyset$;

4 \begin{verbatim}
for $x$ in SPT do
  for $y$ in MP do
    if $x \bullet \cap y \bullet \neq \emptyset$ then
      $V := V \cup \{y\}$
  end
end
\end{verbatim}

5.5.2 Failure to Send Message

Thus far, we have described error detection based on receipt of messages. The Interaction Monitor receives a message and determines if it is valid based on the current state of the conversation. This process will not identify conditions in which an agent has failed to act, rather than acted in the wrong way. In terms of interaction modeling, this problem is characterised by an agent not sending a message that it was expected to send.

In a framework based on observations, we need to consider the issue of observing something that does not happen. To enable this we need to redefine our semantics of observation, from those in which the observer waits forever for an event to occur, to one in which the observer waits for an event to occur over a specific time period. Under such conditions an observer can make assertions such as over time period $t$, event $e$ has not been observed to have occurred.

When developing an interaction protocol there is an expectation that if an agent receives a message it will reply with an appropriate message. Yet, although protocols can be specified with deadlines for message transmission this feature is often not used. The reason that time limits are not imposed on the replying agent is that there is an expectation that the agent will reply as soon as is practicable. Given that agents can engage in multiple dialogues, waiting for a reply in one conversation does not have an impact on executing another. There is, however, an expectation that the reply will be made in a reasonable time.

\footnote{Auction protocols are a notable exception, however only the initial \textit{propose} message has a deadline}
Identifying when a failure has occurred requires that we both determine a time frame for each message and devise a method for determining when the time frame has been exceeded. The first problem is one that should be considered by the developers of the system. If one wishes to have the support of identifying when a message is not sent, then, effort must be expended to determine the timing constraints on the interactions. Each protocol, or if desired, each protocol state must have a duration added. The time limit will indicate the duration that the Interaction Monitor will use to determine if a message has been received.

To support the detection of a failure to send a message we add timer support to the Interaction Monitor. The state places of the Petri Net protocols (representing the state the conversation is in) have a timer added such that whenever a token is deposited the timer is triggered. When a valid message for the current state of the conversation is received a transition is enabled and the token that was marking the state is removed and a new token is generated on another state place. The removal of the token from the state place stops the timer indicating that the message has advanced the state of the conversation. If a timer expires it is inferred that a message that should have been sent has not been sent, hence the error: failure to send a message.

In the event that a timer does expire the Interaction Monitor is able to report the current state of the conversation and can indicate the next valid messages. However, instead of reporting an error a warning is reported. We view the failure to send a message as a warning rather than an error because it is possible that too short a timer has been set. In the early stages of development the timing characteristics of the system may not be well understood and may require refinement. By reporting a warning we are able to alert the developers to possible problems for investigation.

In terms of processing messages into the protocols stored in the Possible Protocols List, those protocols that have been marked as a warning are handled differently to a protocol that has been marked as an error. When a conversation is considered to be in error no more messages are delivered to the Petri Net. In the case of a warning, messages are still delivered and processed by the Petri Net. If a message that was previously marked as not being received subsequently arrives then the status of the Petri Net will be changed to reflect that the conversation has returned to an active, valid state. Or in the case that the message was invalid, the protocol can then be marked with an error.

This default behaviour of considering failure to receive a message as a warning stems from the fact that we assume that the details of the timing constraints of the target system have not been considered
thoroughly, yet one would still like to know when messages are not received. If the timing constraints
are considered and it was desired that a failure to receive a message was reported as an error rather
than a warning then the Interaction Monitor could be easily modified to reflect this behaviour.

5.5.3 Role Related Errors

Having developed a procedure for mapping agent instances to roles we are in a position to identify
certain errors to do with the sending and receiving of messages. We had previously stated that the
rule for depositing a token into the net was to locate the message place that matched the received
message and generate a token on that place. However, once the message place has been located a
further check is performed to ensure that the agent that sent the message was permitted to send it.
This is the verification of the role assignments.

Using the role map we can identify two different errors. Sending a message to the wrong agent,
and the wrong agent sending a message. Both are identified in the same manner. After the first
message from each role has been sent any further messages must conform to the (possibly partial)
mapping established. When a message is received the sender and receiver are extracted from the
message and compared against the role map. If either of the two fields conflict with what is stored in
the mapping no token is generated for the Petri Net. Instead, the Petri Net is marked as being in error
for either sending a message to the wrong recipient, or the wrong agent sending a message.

5.6 Reporting Errors (step 10)

When an interaction is found to be in error it is necessary to communicate this fact to the developer.
Information about which protocol was being followed, which agent instances were mapped to which
roles, and the point at which a conversation diverged from the allowed behaviour need to be recorded
and presented. This information can be used to assist in locating the underlying error in the agent code
that has led to the error manifesting at the interaction level. To support the reporting of interactions
the Interaction Monitor uses a reporting module that directs the relevant information to the developer.

We have developed a simple prototype interface to display information about the status of active
and completed conversations that occur in the deployed multi-agent system. If a violation of a proto-
col is detected the Interaction Monitor will report to the interface describing the error conditions. To
assist in speeding up development of the prototype tool we currently use this same reporting interface
CHAPTER 5. MONITORING AND REPORTING ON INTERACTIONS

5.6.1 The Interaction Monitor Reporting Interface

The reporting interface provides the link between the Interaction Monitor and the developer. Each conversation is described on this interface in a separate panel. Figure 5.3 is a screen capture of the interface with a single conversation active. The conversation id is used as a reference to the panel. All messages that belong to conversation 1 will be directed to this conversation panel. This figure depicts the situation where a single conversation is active and a single message has been received. At the current point in time this conversation is valid.

The Interaction Monitor concurrently models multiple conversations. Each new conversation will result in a new tab being created. The tab that matches up with the latest incoming message will be given focus automatically. The reporting interface receives data from the Interaction Monitor indicating what message was sent, who sent the message and who it was sent to. The current possible protocols are also listed, and as more information is received, the list of possible protocols is updated (invalid protocols will be marked as invalid).

The next example, figure 5.4 shows a second conversation being started. There are now two conversation tabbed panels and the new conversation, conversation 2, has focus. Both conversations
are still valid. To return to another conversation you simply click on the tab for the conversation you want to view. This is necessary if a new message is received from a different conversation as you may lose focus from the conversation that you were watching. It should be noted that this interface is a very simple prototype used to support the experimental evaluation. The primary purpose of this interface is to provide the developer with the necessary information that the Interaction Monitor needs to communicate. Further work is needed to determine the most effective way of presenting the information to the developer. Our main concern is to ensure the developer is made aware of the presence of the bugs that are automatically identified. We have achieved this goal with the basic interface and therefore leave further enhancements to the user experience as future work.

5.6.2 Status Information Provided by the Debugger

Each of the conversation panels displays the status of the conversation by both a text description of the status and the icon on the conversation tab. The text area is responsible for displaying the messages received for each conversation. As long as the conversation is progressing correctly the text area will simply output each of the messages in the form.

\[ \text{AddTaskEv}(\text{Lenny} \rightarrow \text{Carl}) \]

Possible Protocols:
AddTask, AddTaskMoveTask

The first line indicates that AddTaskEv is the event that was received by the debugging toolkit, the sender was Lenny and the recipient Carl. This is a valid message hence no error message is indicated. The second two lines indicate the current possible protocols that the message could belong to. As long as there is at least one valid possible protocol no error will be generated. The protocol in this example is the AddTask protocol or the AddTaskMoveTask protocol. Given that this is the initial message for the conversation we show only this first message. Any further messages will be appended to the text area.

Errors

In the event that an error occurs, the tab of the conversation that is in error will change, a Red Cross will be added beside the conversation id to indicate that the conversation has failed. Figure 5.5 shows what the tab will look like after an erroneous message is received. In this example the conversation is in error because the message that was sent as the first message in the conversation is not a start message in any of the protocols within the system, hence an error is immediately reported.
CHAPTER 5. MONITORING AND REPORTING ON INTERACTIONS

Figure 5.6: A warning message is displayed and a warning icon is added to the conversation tab.

**Warnings**

The Interaction Monitor keeps track of the time elapsed since the last message is received in a conversation. If the time exceeds the developer defined duration then a warning can be reported. Figure 5.6 shows the warning message and the modified icon on the conversation tab. The warning indicates what messages were expected based on the current state of each protocol in the PPL.

Since this is only a warning, if a correct message is received after the conversation has been set to a warning then normal operation can continue. The conversation is switched back to active and the warning icon is removed.

**Successful completion of conversations**

When a conversation completes successfully the icon on the tab will change to a green tick as depicted in figure 5.7 and a message will be displayed indicating that the conversation has completed successfully. If any further messages are received with the conversation id of an already completed conversation they will cause the conversation to be in error. This is because any further messages would not be valid in the protocol as the conversation has already completed. Since we do not allow the reuse of conversation id’s there should be no new messages received by the Interaction Monitor.
with the conversation id of an already completed conversation.

In the current version of the reporting interface all previous conversation panels are retained. If the number of conversations becomes large it would not be practical to keep the successfully completed panels on the main interface. A first step would be to move the completed conversation panels to a secondary location.

5.6.3 Source Code Considerations

Although there is typically no need for a developer to explicitly direct messages to the Interaction Monitor there are considerations that need to be made to enable the use of the interaction monitoring support. For example, all of the agents in the system need to declare that they will be using the debugging tool. This is straightforward, the only input required by the developer is to indicate where the Interaction Monitor is located.

We must also consider how the Interaction Monitor will be able to understand the various message formats that could be used. We do not specify a requirement for the format of a message, however the Interaction Monitor needs to be able to determine the following information from the message:
sender, receiver, message type (e.g. cfp, propose), and conversation id. In fact, instead of sending a carbon copy of the raw message sending this limited information would be more suitable as the Interaction Monitor does not consider the message content. The method for extracting this required information from the original message will be dependent on the format of the messages used by the agents. Ideally any implementation platform that chooses to use the Interaction Monitor would provide the code for extracting the information. If this is done the Interaction Monitor does not need to know about the underlying message format. Alternatively, one could use a standard message format such as FIPA or KQML.

With respect to the use of a conversation id, although the FIPA standard requires a conversation id to be sent with each message it is not always the case that the implementation platform being used will natively support the definition of conversations. In such circumstances support can be added to the agent development process. This is a relatively simple exercise and we only impose the requirement that for any message received, messages sent in response to the message must contain the same conversation id as the received message. This ensures that once the initial message has been assigned a unique conversation id all messages sent in response can be identified as belonging to the same conversation.

When developing, or adapting, an agent application to support the definition of conversation id’s it is necessary that the initial message for each protocol is identified so that a new unique conversation id can be assigned for this message. This is a simple task that, as we have done for the JACK programming platform, can be wrapped up in a function such that the developer need only call a function to acquire a new conversation id when they wish their agent to initiate a conversation.

5.6.4 Computational Impact of the Interaction Monitor

Introducing the Interaction Monitor has minimal impact on the behaviour of the system. The overhead in relation to the number of messages transmitted in the system is for every message sent, a copy of the message is also sent to the Interaction Monitor. If the target system is highly timing dependent then the additional messages being transmitted and the additional time taken to transmit a carbon copy of messages may affect the system’s behaviour. This is a standard concern with adding debugging support to timing dependent systems. The benefit of the debugging support will need to be considered in terms of the possibility to adversely affect the execution of the system. However, given that multi-
agent systems are designed to function with often unknown numbers of agents, and hence unknown numbers of messages, this overhead should not prove a problem in any but the most sensitive systems.
Chapter 6

Evaluation

In this thesis we have demonstrated the applicability of using design documents to support debugging multi-agent systems. We showed how expected program behaviour can be extracted from design artifacts and used as a partial model of correct run-time execution. Violations of the models are automatically detected and explained. This chapter describes the evaluation methodology we applied, as well as the results and conclusions drawn.

The purpose of the evaluation is to measure the extent to which these debugging techniques support the debugging process. To facilitate the evaluation we developed a debugging tool that implements the debugging techniques proposed in this thesis. The debugging tool, or simply, the tool, is a functioning prototype debugger that can be inserted into the run-time environment of an agent system. We use the debugging tool to verify and further explore the capabilities and limitations of our proposed debugging framework by undertaking an empirical study that measured the extent to which debugging performance is improved by use of the debugging tool.

6.1 Empirical Evaluation

Identifying that a bug exists in code is only half of the problem when debugging. Locating and fixing the bug is equally important. Therefore, a thorough evaluation should include an empirical analysis of the benefits of the debugging tool. In this section we focus our attention on the problem of evaluating the degree to which the debugging support proposed in this thesis translates to an improvement in debugging performance. For this part of the evaluation we developed an empirical study and analysed
the results of a controlled experiment in which participants utilised the prototype debugging tool to assist in solving a set of debugging problems.

For the experiment we developed a functional multi-agent system, complete with design and debugging artifacts, to be used as the experiment testbed for evaluating our debugging approach. The format of the experiment was to have participants from two equivalent groups each attempt to resolve each of the debugging problems. Each group would debug half the debugging problems with the debugging tool and half without it. Following is a detailed discussion of the experiment and the results.

### 6.1.1 Experimental Objective

The objective of the experimental evaluation is to test the hypothesis that the debugging tool based on the design documents can be used for run-time error detection to assist the debugging process. Specifically, we wish to evaluate the degree to which such debugging support translates to an improvement in debugging performance. In the background section we discussed the three main stages that a user will proceed through during a debugging task. These are, firstly, identifying the bug by identifying that the program is not acting as expected. Secondly, locating the cause of the bug, by locating the part (or parts) of the source code that are responsible for the bug. And finally, fixing the bug by providing the necessary source code modifications to remove the bug.

Consequently we have 3 specific hypotheses for consideration:

- **H1:** If a user is supported in the debugging task by our debugging tool they will identify a bug more easily than would otherwise be possible if the debugging tool was not used.

- **H2:** If a user is supported in the debugging task by our debugging tool they will locate the cause of a bug more easily than would otherwise be possible if the debugging tool was not used.

- **H3:** If a user is supported in the debugging task by our debugging tool they will fix a bug more easily than would otherwise be possible if the debugging tool was not used.

For each of these hypotheses we address two levels of granularity. Firstly, was the bug successfully identified, located and fixed. This is a true or false relation used to test if use of the debugging tool results in more participants being successful in each of the three stages of debugging. The second
level of granularity takes into consideration the timing data for each of the three stages of debugging. With the timing data we were concerned with the length of time spent trying to identify, locate and fix a bug.

### 6.1.2 Experimental Plan

To evaluate the proposed hypotheses we developed a set of debugging tasks. The debugging task is based on identifying a set of bugs that have been added to a software application that we have developed for the experiment. We designed and developed a Multi-agent Personal Organiser and generated four versions of the application, each seeded with a single bug. Bugs three and four were of the interaction type as described in section 3.5 and bugs one and two were of the plan selection type as discussed in section 3.6. These four bugs comprised the debugging tasks that participants were required to debug, and are referred to as bug 1 through bug 4. Along with the application we provided source code and specific instructions for how to interact with the program and the debugging tool.

Each participant was asked to find and rectify (identify, locate, fix) one interaction bug and one plan selection bug with the debugging tool, and one of each without the tool. Participants were sorted and matched according to ability and were assigned to one of two groups, this ensured that each group had an even spread of novice, intermediate and expert participants. The benefit of this approach is that we can try to limit the effects of the variability inherent in the ability levels of each of the participants.

For the first two bugs group A used the debugging tool, hence was the experimental group and group B did not use the debugging tool, which made them the control group. The opposite held for the final two bugs, group B used the debugging tool whereas group A did not use the tool. This design also enables us to compare the performance of individual users since each user carried out half the debugging tasks with the debugging tool and the other half without it.

### 6.1.3 The Experimental Procedure

Each participant was provided with relevant instructions and the necessary resources to complete the experiment. These included

1. Instructions for taking the experiment
2. Documentation for the test application
3. Source code for the test application

4. Debugging tool

5. Data collection forms

The instructions consisted of a generic set of requirements and information describing the setup and general procedure for carrying out the experiment. Following the generic instructions were the specific instructions for carrying out the individual debugging tasks. The instructions for each group were identical in every way except for the instruction indicating whether the debugging tool should be used.

The instructions included a set of steps that the participants were expected to follow during their interaction with the test application. Each set of steps comprised the minimum test data that would ensure the system would encounter the planted bug. We chose to direct the testing activities of the participants rather than leave them to define their own test cases as there was a concern that if we did not direct the test input the participants may be unlikely to encounter the bugs in the limited time available. A maximum of 1 hour per bug was permitted to limit the experiment to approximately half a day per participant. This seemed to be the maximum length of time that many of the participants would be willing to spend on such an experiment.

By including the sequence of steps to trigger the bug we were able to ensure that the participants would focus their attention in the correct area of the program for each of the debugging tasks. However, this approach inevitably results in the participants quickly identifying the bugs. Consequently we were unable to fully evaluate our first hypothesis, that the debugging tool would help to identify bugs more easily. This is a limitation of the evaluation procedure but seemed necessary.

In addition to the instructions for carrying out the debugging tasks we also provided participants with a set of questions that they should answer as soon as they determined the answer. These questions formed part of the data collection form and were designed to direct the participants to explicitly consider each of the three debugging phases. In particular we were interested in the exact time taken to complete each of the phases. Following is a set of condensed questions, the first two questions relate to hypothesis one and the third and fourth questions relate to hypothesis two and three respectively.

1. Do you believe this code revision is functioning correctly?
2. Describe what you believe the problem (if any) is.

3. Identify the reason (location of the bug) the code is not functioning correctly.

4. Explain how the code can be fixed to remove the bug.

The bug data collection form also included space to record the current time of starting the debugging task, and subsequently for each of the questions that were answered. The instructions for carrying out the experiment for each of the groups and the bug data collection form are included as appendix A and appendix B respectively.

6.1.4 Test Application

The multi-agent test application that we developed for the experiment is a multi-user personal organiser with support for managing a user’s daily activities. A user can add tasks with properties such as duration, priority and deadlines. The application will automatically schedule a task in an appropriate time slot. Automatic scheduling of meetings between users is supported and handled in much the same way as scheduling tasks, with the addition that multiple users need to be consulted. Features such as automatic rescheduling of tasks and meetings are supported. The application was designed using the Prometheus methodology (Padgham & Winikoff 2004) and was implemented using the JACK Intelligent Agents programming platform (Busetta, Rönquist, Hodgson & Lucas 1998).

In developing the test application we had to consider the issue of system complexity. If the application were too simplistic it would be difficult to generalise the results to more complex systems. Yet, if the application was too complex then subjects may not be able to comprehend the system in the limited time they had available.

The resulting test application contains 5 agent types that make use of 55 plans to handle 63 event types. Many other supporting files, including source code for diary manipulation and GUI functionality are included. Each user of the personal organiser system is supported by a set of 5 agents. The user interacts directly with the GUI to schedule tasks that need to be completed or meetings to be held with other users of the system. For the purposes of the experiment we have set the number of users to three. An abstract view of the system with three users is shown in figure 6.1.

User documentation for the personal organiser system was provided to the participants. The documentation consists of approximately 40 pages of functional specifications and design documentation.
Figure 6.1: Abstract System Overview, 3 users.
including the design artifacts used as inputs to the debugging tool. The system relies heavily on agent interaction and as such we have developed a number of interaction protocols using AUML notation. The AUML protocols were converted to the internal representation as specified by the translation rules set out in chapter 4 and were added to the debugging tool for use in the experiments.

6.1.5 Participant Selection and Group Allocation

Participant selection and group allocation was of particular importance in the design of this experiment. Although the debugging techniques are independent of a specific underlying implementation platform we needed to select a programming environment to implement the test application in. As such, a requirement for participation in the experiment was a familiarity with the JACK programming platform. Members of the local agent community in Melbourne, Australia were contacted. An email was also sent to an international JACK programming mailing list. Finally the developers of the JACK programming platform were contacted for participation. In total 20 subjects completed the experiment. The majority, 17, were from Melbourne, Australia with only 3 subjects sourced from outside of Melbourne.

To ensure comparable levels of ability between groups, we administered a pre-experiment survey to measure the experience and ability of each participant with regard to programming generally and specifically programming on an agent platform such as JACK. The surveys were evaluated and the participants were assigned to one of three categories: beginner, intermediate and advanced. We then randomly allocated equal numbers of each category to the two groups. Group A used the debugging tool for bugs 1 and 2 but not for bugs 3 and 4, whereas Group B used the debugging tool for bugs 3 and 4 but not for bugs 1 and 2.

For reference we have included the pre-experiment survey that participants were asked to complete as appendix A.

6.1.6 Data Preparation

The data collection forms contained space for the responses to the questions that would indicate if the participant was able to solve the debugging problem. The questions on the data collection form offer a structured and consistent way for the participants to record their progress in resolving the bugs. Progress is recorded for each of the three main stages of debugging: identification of the bug,
identification of the cause of the bug, and identification of a solution to the bug. We graded each of the responses to these questions for each of the bug revisions to determine if the participant solved the debugging task.

We had identified that the grading of the responses could be somewhat subjective. This is because although we had developed descriptions and solutions to the bug problems it would be possible that the participants would present bug descriptions or solutions from a different viewpoint that was equally valid. Therefore, to limit the possibility of bias, grading was carried out without knowledge of which participant used the debugging tool for which debugging task. Only after the forms had been graded were the results added to the appropriate group.

The responses were graded on a scale of 1 to 3, a score of 1 indicated that the participant failed to provide a correct response, a score of 2 indicated that a partially correct response was provided and a score of 3 indicated that the participant responded correctly. After grading all the data collection forms it was noted that only 3 out of 120 responses received a score of 2 (a partially correct answer). To simplify the analysis of the data and to greatly improve the presentation of the results we decided to alter the grading by revoking the option of marking a response as partially correct. The grading was simplified so that a score of 3 is recorded as yes, indicating that the response was correct and a score of 1 is recorded as no indicating the response was not correct.

The three responses that received a score of 2, being partially correct, were first assigned to the yes category and the statistics were computed. Next we recomputed the statistics but this time the three responses were assigned to the no category. Given that there was no change to the effective significance we concluded that such a modification was appropriate.\(^1\) In the thesis we present those responses that were initially graded as partially correct in the no category.

In addition to the grading of the debugging tasks we were able to extract the time taken to complete each of the tasks. The bug data collection form included a field for recording the time when each of the three debugging sub tasks was completed. Given that we asked participants to record the actual time we end up with a cumulative time for each of the debugging phases. This means that the time to fix a bug includes the time taken to locate the cause of the bug and to identify that the bug exists in the first place.

\(^1\)By effective significance we mean that the change did not affect whether significance was above or below the threshold of \(p \leq 0.05\)
6.1.7 Statistical Tests

In evaluating our hypotheses we identified three suitable statistical tests. First, a chi-square test that measures the difference between two proportions. This test was used to evaluate the hypothesis that the number of debugging tasks resolved would increase if the debugging tool was used. For each of the three hypotheses we have two sets of data from the two groups. The chi-square test provides a measure of how different two groups are with respect to a treatment variable (use of the debugging tool). If the data from the two groups are close to equal then we can conclude that the debugging tool did not have an effect on the hypothesis under consideration. For each of the three debugging stages we propose the null hypothesis that there is no difference between the two groups. Rejecting the null hypothesis, based on the resulting test statistic, supports our initial suggestion that use of the debugging tool will result in an increase in performance.

The second set of tests were chosen to test the hypotheses relating to the time taken to complete a debugging task. We selected the Wilcoxon rank sum test. This test compares the median score for two sets of data to determine if the difference between the medians can be explained via a random sampling. If not, then it is likely that the variable being tested for has affected the positioning of the medians, indicating that the variable was responsible for the difference. The test is a non-parametric test that is suitable for data that is not consistent with a normal distribution. The data we received was not normally distributed which is not surprising given that there were large differences between the ability level of the participants.

There was a further reason for choosing the Wilcoxon rank sum test. Timing data is recorded for each of the debugging tasks. If a participant was able to solve a debugging task then they would record the time taken to solve it. Since there is a 1 hour time limit imposed on each of the bug versions the timing results range from 1 minute through to 60 minutes. If a participant was unable to solve the debugging task within the 1 hour time limit we want to be able to record that in our data since it was a common occurrence. The way that the Wilcoxon rank sum test computes its test statistic enables us to include the failed results. Instead of using the variations between the raw timing values the Wilcoxon rank sum test ranks the timing values. The logic behind the test is that if two sets of data are different there will be little overlap of the ranks between the two sets of data.

For example, if Group A was using the debugging tool we would expect that more participants would be ranked higher (by solving the debugging task quicker) than those from Group B not using
the debugging tool. If there is a great deal of overlap between the two groups then it is likely that the debugging tool does not have the expected effect. This ranking of data enables us to include those participants who failed to complete a debugging task by assigning them a time of 61 minutes. Since no participant that solved the debugging task can have a time of greater than 60 minutes we ensure that any participant that fails to complete will be ranked behind all others that completed.

The final test that we employed was a test on the effects of the debugging tool at the individual level. The previous tests are based on the results of the differences between groups of individuals. However, the design of the experiment also enables us to compare the performance of individuals when using the tool and when not using the tool. The test used is once again the Wilcoxon rank sum test, however, this time it is performed as a matched test. This means that there is a direct relationship between each row of timing data. Specifically, we are comparing the time taken for a participant to complete a (different) debugging task with the tool against the time taken for the same participant to complete a task without the debugging tool. We will describe the details of how this was done in the relevant section of the results.

### 6.2 Empirical Results

In this section we present the results of the debugging experiments We compare the results over each of the sub problems that occur during debugging: identify, locate and fix. For each of these problems we consider if the participant was successful as well as the time taken to resolve the bug (bug resolution speed). We analyse the difference between the two groups with respect to our hypothesis to identify the effect that our debugging tool has on debugging the test application.

#### 6.2.1 Identifying the Bug

**H1:** If a user is supported in the debugging task by our debugging tool they will identify a bug more easily than would otherwise be possible if the debugging tool was not used.

**Success**

Figure 6.2 compares the the number of bugs found by the group using the debugging tool against the number of bugs found by the group that did not use the debugging tool. Each group consisted
of 10 participants, therefore, a score of 10 indicates that all participants in the group were able to identify the bug. Over the four bug revisions only one participant from the group that did not use the debugging tool in revision 2, 3 and 4 failed to identify the bug.

It is clear that the debugging tool did not have any significant effect on a user’s ability to successfully identify the bug for any of the bug revisions. For completeness we performed a chi-square test focusing on the differences in the two proportions between the two groups (with debugging tool and without). A significant value for chi-square would indicate a significant difference between the two groups (Gravetter & Wallnau 1996). For Bug 1 there is no difference between the two groups. For Bugs 2 through 4, the p-value = 0.3049, since this is not less than 0.05 we conclude that the debugging tool did not have any effect on a participant’s ability to successfully identify bugs. This outcome is not surprising, the high success rate at finding the bug is because the instructions guided the user to the problem.
CHAPTER 6. EVALUATION

In the Speed sections of the results we are interested in identifying any observable differences between the time it takes to identify each of the bug revisions. Each bug revision is shown as a comparison between the group that used the debugging tool for the bug version (light shade of grey) and the group that did not use the debugging tool for the bug version (dark shade of grey). Since there was a one hour time limit imposed on the bug version the y-axis is marked from 0 to 61 minutes. We extend the axis by 1 minute to differentiate between participants that finished in exactly 1 hour and those that did not finish. The latter case is presented as taking 61 minutes to complete the task.

Figure 6.3: Identified timing data
Figure 6.3 shows the timing results for the four bug revisions presented as a box plot. A comparison is made between the time taken for the group that used the debugging tool to help identify the bug against the group that did not use the tool to help identify the bug.

Examining the first bug revision we see that there is very little difference between the times for the two groups. The median time to find the bug is identical, 10 minutes. Furthermore the spread of the data is similar between the two groups, with the exception of the two outlying data points for the group that used the tool to help identify the bug. The differences between the two groups appears insignificant which was confirmed with a Wilcoxon rank sum (p=0.4638). This result indicates that there is no statistically significant difference between the two groups for bug revision 1.

For bug revision 2 both groups have a similar median time to identify the bug, between 5 and 6 minutes. The group that did not use the tool is, however, slightly positively skewed indicating a larger variance between the results with a few participants taking up to 23 minutes to identify the bug. A Wilcoxon rank sum test indicates that this difference is also not significant (p=0.898). Bug revision 3 also recorded no significant difference (p=0.79).

The boxplots for bug revision 4 indicate that the median time to identify the bug for the group that used the tool was 14 minutes compared to a median time of 9 minutes for the group that did not use the tool. The Wilcoxon rank sum test indicates that this is not significant at the 95 percent level (p=0.06317). Although this is not quite significant it is close, and it is worth noting that it is in the opposite direction of that which would be expected. For bug 4, using the debugging tool to identify the bug seemed to be somewhat of a hindrance. We will discuss our analysis of this and other interesting observations at the closing of the results section. (section 6.2.5)

### 6.2.2 Locating the Cause of the Bug

**H2:** *If a user is supported in the debugging task by our debugging tool they will locate the cause of a bug more easily than would otherwise be possible if the debugging tool was not used.*

---

2 Outlying data points represent results that were statistically very far from the group’s median and do not fit in with the box plot.
Figure 6.4 shows the number of participants that successfully located the cause of the bug for each of the bug revisions. Again we are comparing the difference between the group that used the debugging tool against the group that did not use the debugging tool.

There is a clear difference between the proportions for both bug 1 and bug 4. Seven out of ten participants that used the debugging tool for bug 1 were able to locate the bug compared with only one out of ten for the group that did not use the tool. A similar effect was observed for bug 4 with eight out ten participants finding the bug with the aid of the debugging tool compared with zero from the group that did not use the debugging tool.

A chi-square test for the differences in the two proportions confirms this, for bug 1 (p=0.0062) and bug 4 (p=0.0003). These results clearly show that, for bug revisions 1 and 4, the group that used the debugging tool to help locate the bug performed markedly better than the group that did not use the debugging tool. The proportion data for bug 2 and bug 3 also show differences, but these do not appear to be significant. For bug 2, nine out of ten participants successfully located the bug using...
the tool compared with eight out of ten for the group that did not use the tool (p=0.5312) There was a slightly greater difference between the two groups for bug revision 3, however, this difference was also not significant (p=0.1213).

**Speed**

Figure 6.5 shows the box plots for each of the 4 bug revisions concerning the time spent trying to locate the cause of the bug. As previously mentioned, it should be noted that the time recorded here is cumulative from the time the experiment was started for the bug revision. As such it includes the
time taken to identify the presence of the bug.

As with the analysis of identifying the bugs we observe that using the debugging tool clearly shows an improvement in performance for bugs 1 and 4. The median times for the group that used the debugging tool for bug 1 and bug 4 are 38 minutes and 41 minutes respectively. Since only 1 participant from the group that did not use the debugging tool was able to locate the cause of the bug within the 1 hour time limit the box plots are entirely at the 61 minute mark.

A Wilcoxon rank sum test confirms our suspicion, for bug 1 (p=0.0027), and for bug 4 (p=0.0003). It is clear that the debugging tool was a significant factor in the time taken to locate the bug for these two bug versions. From these results we can conclude that the difference between the median time to locate bug 1 was at least 22 minutes (38 minutes vs 60(+1) minutes). For bug 4 it was at least 19 minutes. This is quite a substantial amount and we can speculate that if the one hour time limit was not imposed the real difference could be greater.

The results for bug 2 reveal less of a difference between the two groups. The majority of participants from the group that used the debugger were able to locate the bug between the 11 minute and 39 minute mark. In comparison, the majority of participants from the group that didn’t use the debugger located the bug between the 29 minute mark and 45 minute mark. The group that used the debugging tool had a median time to completion of 23 minutes compared with the group that did not use the tool having a median time to completion of 37.5 minutes. The p-value for the Wilcoxon rank sum test is 0.0683. This is not quite significant at the 95 percent level, however, the statistic is very close.

The box plots for bug version 3 are very different from each other, and suggest a considerable performance difference between the two groups. The data for the group that used the debugging tool is clustered around a median time of 19 minutes. There is little variance in times for this group with only 1 participant finishing with a time uncharacteristic of the rest of the group (41 mins). On the other side, the plot of the data for the group that did not use the debugging tool is a lot more varied. The median time to locate the bug was almost 40 minutes, which is only 1 minute less than the single worst time for the comparison group. In addition, only two participants from the group that did not use the debugger finished quicker than the median time for the group that used the tool. These differences translate to more than a 20 minute increase in time for the group that did not use the debugging tool and the test statistic recorded is significant (p= 0.0059).
6.2.3 Fixing the Bug

H3: If a user is supported in the debugging task by our debugging tool they will fix a bug more easily than would otherwise be possible if the debugging tool was not used.

Success

Figure 6.6 shows the proportion of participants that were able to successfully fix the bug within what time was left of the 1 hour time limit. From this graph we note that the number of successful responses has dropped significantly from the previous case of locating the cause. The total number of successes for fixing the bug, over both groups and over all bugs, was 41 out of a possible 80. This is 7 fewer than the previous phase in which 48 participants managed to complete the task of locating the bug (3 fewer with the tool and 4 fewer without the tool).

As for the previous debugging phase, there is a clear difference between the two groups for bug 1. Seven out of a possible 10 participants from the group that used the debugging tool managed to fix the bug. In comparison none of the participants from the group that did not use the debugger were
able to fix the bug within the 1 hour time limit.\footnote{Given that only 1 participant found the bug there was only 1 participant who attempted to fix the bug.} A chi-square test for the difference among the two proportions confirms this significance (p=0.0010).

The results for bug 2 do not indicate any significant difference between the two groups. A single correct response separated the two groups, and the difference is not significant (p=0.5312).

The final two bug versions indicated a positive effect of using the debugging tool with more participants from the group that used the tool able to determine the correct fix for these bugs. Figure 6.6 shows a significant eight to three ratio in favor of the group that used the tool for bug 3 (p=0.0246) and six to zero for bug 4 (p=0.0034).

**Speed**

As in the previous phase there is strong evidence that the debugging tool was a significant advantage for bug 1 and bug 4. No participant was able to fix the bug for either of these two bug versions if they were in the group that did not have the tool. The median time for completing the task is therefore artificially set at 61 minutes, while the median time taken for the group that used the tool was 45.5 minutes for bug 1 and 47.5 minutes for bug 4. These differences are statistically significant with both recording a p-value of 0.0015.

For bug 2 there is a large difference between the median times to provide the fix. With a median time of 24 minutes for the group that used the debugging tool against 45 minutes for the group that did not there appears to be a significant difference. There is, however, a large amount of variation in the recorded times, which the Wilcoxon test takes into consideration. This results in the difference being insignificant (p=0.1192).

Bug 3 shows a similar trend. The difference between the median time to provide a fix for the bug is the same as for bug 2, 21 minutes. There are, however, slight differences in the distribution of times that make this bug version more statistically significant than the previous one. A Wilcoxon rank sum test indicates that the observed difference between the median times is significant at the 95 percent level (p=0.0497).
6.2.4 Within Subject Analysis

It is likely that a certain amount of variability exists because of the varying abilities of each of the participants. A within subject analysis is an effective method for handling the variability between subjects and can shed some further light on the effect of the debugging tool on an individual’s performance. The general process of evaluation is to have each participant complete a task under two different circumstances (typically described as a control task and a treatment task). Scores are recorded for each task and the difference between the scores for the two tasks is obtained. If the difference is close to zero then we can conclude that the treatment variable (the debugging tool) had no effect.
on the outcome. However, if a large difference is measured then there is evidence that the treatment condition did effect the outcome.

For this analysis we will concentrate on bug 2 and bug 3, and only in the second two phases. We do this because we have already noted a considerable and statistically significant advantage of using the debugging tool for bugs 1 and 4, and we see know advantage to using the debugging tool for the first phase. We perform a matched pair evaluation with the same data as already recorded. Group A comprises participants 1 though 10 and these participants used the debugging tool for bug 2 but not for bug 3. Group B comprises participants 11 through 20 and did not use the debugging tool for bug 2 but did use it for bug 3.

**Locating the cause of the bug**

Figure 6.8 is a graphical representation of the differences between the times for each participant. Group A is represented by the light shading of grey and group B by the dark shading of grey. This graph shows how much quicker each participant was able to locate each of the bugs. For example, participant 1 was observed to have a 30 minute speed increase in locating bug 2 (using the tool) over Bug 3 (not using the tool). Similarly, participant 12, from group B, was observed to have a 15 minute speed increase in locating bug 3 (using the tool) over Bug 2 (not using the tool).

The graph shows an increase in performance for most subjects when the debugging tool was used to locate the bug. However, there are three exceptions. Participant 6, 10 and 13 had a performance decrease of approximately 7, 16 and 1 minutes respectively. Performing a matched pair Wilcoxon rank sum test confirms our hypothesis that in general the debugging tool results in bugs being located more quickly (p=0.00987).

**Providing a fix for the bug**

Figure 6.9 shows the difference that the debugging tool had for each participant in fixing the bugs. Again this graph clearly shows that the majority of the participants were able to fix the bug quicker when using the tool than without. The matched pair test indicates that this is indeed a significant difference (p=0.00987).
6.2.5 Discussion

The results that we have obtained clearly demonstrate that using the debugging tool resulted in much better performance than when no tool was used. Figure 6.10 provides an overview of the results in terms of the overall success rate at identifying, locating and fixing the bug. This graph presents an aggregated summary of success for each of the three debugging phases.

In terms of identifying that a bug exists we found that in this experiment the debugging tool was not a significant factor in the participant being able to identify the bug. We had earlier identified that the design of the experiment was such that users were given specific instructions for how to trigger the bugs which meant that both groups should identify the bugs without much trouble.

We had not, however, predicted that the debugging tool might slow users down while trying to
identify that a bug exists in the code. This seemed to be the case for bug version 4. Although not statistically significant it is worth looking into. It has been observed in the testing and debugging of functional programs that the style of interruption of a debugging tool can affect the debugging activity (Robertson, Prabhakararao, Burnett, Cook, Ruthruff, Beckwith & Phalgune 2004). It is possible that during our experiment participants were focusing on the debugging tool after entering each command and querying the output of the debugging tool instead of focusing on the behaviour of the program that was being tested. We speculate that the ability of the debugging tool to automatically identify bugs would be far more successful under non-experimental conditions where the users are not directing the test data to a specific bug trigger, as was necessary for this experiment.

In terms of overall successes in locating the cause and fixing the bugs, we identified that using
the debugging tool afforded the participant a considerable advantage. This was most noticeable with bugs 1 and 4, which are in our opinion the more difficult bugs to resolve, but was also to some degree significant for bugs 2 and 3. To further explore the effectiveness of the debugging tool for these two bugs we spent more time analysing the timing data for these bugs and found that there was significant evidence that the time taken to resolve the bug was reduced for all tests except for locating the cause of bug 3.

We noted that there was substantial variability in the results for bug revision 2 and 3 and therefore tried to limit the effects of the variability by performing a matched pair test. This was done and we showed that there was indeed a significant advantage to using the debugging tool when we compared the performance within each individual’s results. Interestingly there were a few participants that performed better on bug 2 or 3 when they were not using the debugging tool. After analysing these individuals we noted that they were ranked as intermediate or expert. One possible explanation for them performing better without the tool is that both bugs were easy for them given their skill level. Using the debugging tool may have interrupted their normal debugging activities and slowed them down on what, for them, was a simple bug to locate. If we consider the results of these participants
with the more difficult bugs included then the difference between using the tool and not using the tool is reduced. A final comment is that there were several other participants rated at the intermediate or advanced level for which the debugging tool was found to be an advantage and over the entire experiment the tool was shown to be statistically significant in reducing debugging time.
Chapter 7

Conclusion and Future Work

In this thesis we explored the problems associated with debugging multi-agent systems and have provided a framework for debugging based on using design documents to automatically identify errors in developed systems. We have demonstrated a proof of concept that design documents can help in debugging, using two design artifacts: protocol specifications and parts of a Prometheus event descriptor specification.

We have developed a debugging framework and described the process of converting design artifacts, adding them to a debugging tool and explaining how the debugging tool is integrated to the development of a system. The framework is implemented as a prototype tool and this developed tool was used to assess the effectiveness of our debugging technique via an empirical evaluation. The evaluation showed that this debugging approach was effective in helping to resolve bugs in multi-agent systems.

An important part of using a design document as a debugging aid is in determining how the design document can be represented internally to a debugging tool to allow for effective comparison against observed behaviour. Interactions are an important aspect of multi-agent systems that are difficult to debug and have been the focus of a number of other debugging approaches and tool developments. We provide a detailed mechanism for using AUML interaction protocols in debugging interactions. Our transformations from AUML to Petri Nets to realise the debugging artifacts are general purpose and local so they can be used to convert most arbitrary AUML protocols.

The main research contributions for this thesis can be summarised as follows:

- Proof of concept of debugging approach.
• Debugging framework.

• Mechanism for translating AUML protocols to Petri Nets

7.1 Proof of Concept of Debugging Approach

This thesis clearly demonstrates the efficacy of using design artifacts as a suitable technique for debugging agent systems. We have demonstrated that by identifying suitable properties of interest from design artifacts, and converting them to debugging artifacts we can provide useful debugging support.

We initially identified the event descriptor from the Prometheus methodology as a candidate debugging artifact. We demonstrated how we can extract information about plan selection rules based on the properties defined in the event. An example of such a property is defining an event as having full coverage, which requires that when the event fires it will always have a plan applicable to handle it. We also identify the plan overlap property which describes that when an event fires multiple plans may be applicable.

We show how we extract and encode these properties and then at run-time monitor the system to determine if the observed behaviour diverges from that specified by the design. If the system diverges from this model then we can conclude that an error has occurred.

The debugging framework has been implemented in a debugging prototype and we added event monitoring as well as protocol monitoring components. The prototype was used in the evaluation of our debugging approach. The evaluation is based on an empirical study that measured the extent to which debugging performance is improved by use of the debugging tool. In order to conduct the experimental evaluation we developed a moderately large and complex meeting scheduler that we used as our test-bed application. We then systematically inserted bugs into the application which we provided to our experimental subjects to assess debugging performance. The application included suitable design artifacts which were translated using our translation procedure and added to our debugging prototype.

The results that we obtained clearly demonstrate that using the debugging tool translates to an improvement in debugging performance. We gathered data from twenty participants over four debugging tasks. Each debugging task was measured over the three aspects of debugging: identifying,
locating and fixing a bug. In terms of overall successes in locating the cause and fixing the bugs we identified that using the debugging tool afforded the participant a considerable advantage.

7.2 Translation of AUML Protocols to Petri Nets

AUML is commonly used to specify agent interaction protocols. The notation provides good support for specifying and communicating protocols to developers but is not designed to be directly executed. In contrast to this is the Petri Net notation which has been used by a number of people for execution based interaction specification. However, large or complex interaction specifications are typically difficult to follow in Petri Nets and thus they are unsuitable as the basis for design documentation. We provided detailed translation rules that allow a protocol specified in AUML to be represented as a Petri Net. The resulting Petri Net could then be used by our debugging tool to compare run-time execution against the protocol.

The AUML protocol specification enables complex protocols to be developed using a set of structured components such as loops and alternatives. In developing a mechanism for translating such protocols we needed to identify and provide translation rules for each of the AUML components. Our approach was to convert the individual AUML fragments into Petri Net fragments and then join the fragments together to provide the resulting Petri Net interaction protocol. This involves identifying the possible states in a protocol and applying our derived transformation rules. The transformations are general purpose and local so they can be used to convert most arbitrary AUML protocols. After describing the rules we showed some examples of how to convert complex AUML protocols to equivalent Petri Net versions which could then be used in our debugging tool.

7.3 A Domain Independent Framework for Debugging

We have developed a framework in which the debugging techniques proposed can be incorporated into agent system development. Our approach was to use a monitoring agent external to the developed agent system. To access the debugging capabilities of the monitoring agent a system developer need only provide a simple interface to the monitoring agent. Our framework is extensible and new debugging artifacts can be added to the debugging system as they are developed. The debugging framework details the general procedure for identifying, selecting and transforming design artifacts.
It shows how the converted artifacts fit into wider system development, including such things as how source code should be automatically modified to interact with the debugging framework. At run-time the system is monitored using the debugging artifacts and when a diversion from specified behaviour is identified a report is sent to an external reporting interface.

### 7.4 Future Work

There are many agent design artifacts that have not been considered in this thesis. Some of these probably have the potential to be exploited to provide a greater coverage of the debugging search space, and could be incorporated into a debugging toolkit, such as the prototype tool that was developed in this thesis. In addition to developing translation rules for other design artifacts there are some aspects of the AUML specification where additional translation rules could be developed.

For a debugger to be maximally useful the user interface is a critical aspect and clearly there is substantial additional work to be done in this area. Further development areas would be to integrate the debugging tool into an agent based design tool such as the Prometheus Design Tool.

In this thesis we considered agent systems that are composed of a reasonably small number of agents (approximately 20). We have not addressed the issues that may arise with trying to debug very large systems, with hundreds or even thousands of agents. In such systems the number of messages may be too great for a single debugging agent to handle. If this is found to be the case it may be necessary to develop a distributed debugging environment with multiple debugging agents. Managing the interactions between debugging agents and determining how to group agents would be necessary.

A final comment is concerned with the assumption we make with respect to guaranteed message delivery from the agents to the debugging agent. If in practice we cannot make the assumption that the underlying network infrastructure will guarantee message delivery then we will need to look at how this will effect the functioning of the system. It may be necessary to explore techniques that deal with message failure or we may need to incorporate some other methods to guarantee delivery.

Although much interesting work remains to be done this thesis has established the viability and effectiveness of the approach to using design artifacts in the debugging of multi-agent systems.
Appendix A

Pre-experiment Survey
Debugging MAS Pre-Experiment survey.

Thank you for taking the time to participate in this experiment, we appreciate your time and hope that our work will be useful to the wider agent community. Please take a few minutes to fill in the following survey so we can better utilize the data we collect from the experiments.

1) What best describes your experience with JACK?
   ( ) Worked on the development of the JACK engine/kernel
   ( ) Developed JACK application(s) for commercial purposes
   ( ) Developed JACK application(s) as part of a post doc or Ph.D.
   ( ) Developed JACK application(s) as part of an honors project or summer studentship
   ( ) Developed JACK application(s) as part of a university subject.
   ( ) Tried to write a few applications but didn't really finish any
   ( ) Have only completed the JACK tutorials.
   ( ) Other, please describe:

2) Approximately how many projects have you worked on that involved programming in JACK?
   (You may provide a range if you are not sure)

3) Please rate your grasp of the JAVA programming language on the following scale.
   (circle the number that most closely represents your skill level)

   10  9  8  7  6  5  4  3  2  1  0
   Excellent  Average  Poor

4) Please rate your programming ability on the following scale.
   (circle the number that most closely represents your skill level)

   10  9  8  7  6  5  4  3  2  1  0
   Excellent  Average  Poor

Figure A.1: Pre-experiment Survey (Page 1)
5) Have you ever used a debugging aid to help track down a bug?

( ) Yes  ( ) No

6) If you answered “Yes” to question 5, please specify how often you use these tool(s)
   ( ) Whenever I code
   ( ) Often
   ( ) Only when I encounter a difficult bug
   ( ) Rarely
   ( ) Never

Please list up to 5 debugging tools that you use or have used to help debug code

1
……………………………………………………………………………………………………

2
……………………………………………………………………………………………………

3
……………………………………………………………………………………………………

4
……………………………………………………………………………………………………

5
……………………………………………………………………………………………………

7) If I am made aware of a bug I
   ( ) almost always find the problem quickly
   ( ) almost always find the problem, however it often takes a long time
   ( ) sometimes find the problem quickly
   ( ) sometimes find the problem, however it often takes a long time
   ( ) rarely find the problem

8) How much time do you spend working with code that you have not written yourself?
   (tick a single response that best describes your situation)
   ( ) more or less daily
   ( ) I have worked on a number of projects that required me to work with others code
   ( ) I have only worked on a few projects that required this
   ( ) rarely or never
   ( ) Other, please describe,

……………………………………………………………………………………………………

……………………………………………………………………………………………………
9) How much experience have you had with Agent Oriented Software Engineering methodologies, such as Prometheus?
   (tick a single response that best describes your situation)
   ( ) Designed a medium to large scale application with an AOSE methodology
   ( ) Designed a small application following an AOSE methodology
   ( ) Know very little about any particular methodology
   ( ) Other, please describe,
Appendix B

Instructions for experiment
Instructions for Completing the Debugging Experiment

Overview:

Thank you very much for taking the time to participate in this experiment. We have been working on developing tools to help debug multi-agent systems and your participation will help us with this goal.

The experiment basically consists of you interacting with 4 versions of a JACK application that we have developed. The rest of this document details the procedure that you should follow to do this as well as all the necessary software, documentation and forms you will need to fill out.

In order for you to complete the experiment you need to ensure that you have your environment set up correctly and that you have all the required materials. Written materials, such as Documentation and Data Collection forms are provided in Hard Copy format. The source code and other required software is provided on CD.

Checklist:

Please ensure that you have the following materials and that your environment settings are set as specified below.

1) Development Software  2) Test Application Software (MAPO)
3) MAPO Documentation  4) Bug Data Collection Forms
5) Instructions For Using the Debugging Toolkit

Development Software Requirements:

- Required JACK version: **4.1h**
- Required JAVA Version: **1.4.2_03-b02** (or any other that compiles!)

It is assumed that you have an appropriate licence to use JACK 4.1h however if you are unable to get access to this please let me know. JACK 4.1h is installed on RMIT University computer science servers.

Test Application Software Requirements:

You have access to 4 versions of the test application; referred to as MAPO (Multi-Agent Personal Organiser), these versions can be found on the Experiment Resources CD in the zip file mapo.zip (or I may have told you where to download them from!)

The zip file, once you extract it, should create 4 directories under the base directory for each of the code revisions. This directory structure should expand to: `<base-directory>/rev1`
<base-directory>/rev2
<base-directory>/rev3
<base-directory>/rev4
Where <base-directory> is the directory where you want to run the experiments.

**Important:** Do not view or interact with any of this code until you begin the experiments, except perhaps, to verify that the code has been unpacked correctly

**MAPO Documentation:**

The design documents that have been created in the process of developing MAPO have been provided so that you may understand how the program is supposed to function. During the experiments you may need to use the design document to determine if the program is functioning correctly or to help you diagnose any problems that you have found. The design document is labelled:

\[(M)ulti \ (A)gent \ (P)ersonal \ (O)rganiser)\]
\[(MAPO)\]
\[Version 1.0\]

**NOTE:** Before you begin the experimental procedure you should read up-to and including all of section 3. You may do this now or after you finish reading this document.

**Bug Data Collection Forms**

The Bug Data Collection forms are used to record your interaction and findings for each of the code revisions. You should have 8 forms labelled “**BUG DATA COLLECTION FORM**”. You only need to fill out a single form for each of the 4 code revisions. I have provided you with a few spare forms in case you need to rewrite anything. Also, if you need to, you can get them from the Resource CD.

**Instructions For Using The Debugging Toolkit**

To facilitate the debugging process you will be using a toolkit that we have developed to help identify bugs in Multi agent systems. We have provided a short document explaining the diagnostic information that the toolkit can provide. Instructions on how to start the debugging toolkit will be provided in the next section.

**NOTE:** You are free to read this document now or use it as a reference during the experimental procedure.

**WHAT TO DO NEXT**

After completing the checklists, setting up the JAVA and JACK if required and reading any required materials you should next read through the **Experimental Procedure** document, then you will be ready to run the experiments.
Experimental Procedure

The experiment requires you to perform debugging activities on 4 versions of MAPO. In each version you will be asked to interact with that version and determine if the version is functioning correctly. If it is not functioning correctly you need to describe what the problem is. You will also be required to identify where the problem exists and will be asked to provide a solution. More information is provided on the BUG DATA COLLECTION FORM.

You will be required to follow the same steps for each of the 4 versions of the software. You are only permitted to spend up to 60 minutes on each of the code revisions, if you do run out of time please just mark that on the BUG DATA COLLECTION FORM and continue on with the next code revision.

The basic steps are outlined below; you should read over these steps before actually starting the experiments.

The most important step of each iteration of the experiment is the first step, so please be sure to make sure you do it.

1) Switch to the directory with the next code revision you are up to, i.e. base-directory/rev1/
2) Compile the code (this is done by typing: compile)
3(a) Take a blank BUG DATA COLLECTION FORM,
3(b) Add the date and your name at the top of the page.
3(c) Fill out Question 1 and Question 2.
3(d) If you have not already done so, read the rest of the questions on the form so that you know what is expected of you during the experiment. (However don’t fill out past Question 2 just yet)
4) Follow the instructions on “Instructions for code revision N” where N refers to the code revision you are currently working on.
5) Fill out the BUG DATA COLLECTION FORM as you proceed.
6) END of experiment for this code revision.
6(a) Make sure you close down the debugging toolkit.
6(b) Please start again from point 1 with the next code revision.

NOTES:

- You may modify and recompile the source code at any time during the experiment. If you do recompile make sure you use the compile command, as this program will add back some necessary components for debugging.
- You may use any debugging technique you wish.
- DO NOT use any sort of diff command on the different code revisions, that would be cheating and the results would then be of no use to me.
- The debugger directory should not be accessed.
- The backend directory may be accessed but we are not testing for bugs in that directory, only bugs in the agent code are being tested for.
Instructions for code revision 1

Compilation needs to be done in Unix/Linux as I have used some perl scripts to add code to the agents. However you can run (and I recommend you do) the application + debugger from Windows.

i) Switch to the directory /rev1

ii) Compile the source using: compile

iii) Fill out the BUG DATA COLLECTION FORM upto and including Question 2

You are now ready to begin the experiment.

You are free to view or add code to MAPO at any time

You are free to restart the experiment at any time

You are free to perform actions other than those specified below. The tasks specified below are the minimum you should complete that will trigger a bug if one exists.

You should fill out the relevant sections of the BUG DATA COLLECTION FORM whenever the need arises, i.e. as soon the form requires you to write a response.

1) Open a terminal and start the debugger: java debugger.Main (this will pop up a debugging Frame) 

2) You will need a single terminal for the application, change directory to /rev1  

3) Type: start1.bat (this will start an organiser for user John)

4) Review the current diary entries and the task entries. You may want to navigate to the following day to see if there are any tasks scheduled there.

5) Add a task with the following details:
   Description = Clean my desk
   Due Date = End of Week
   Duration = 2 hours
   Priority = 2

   Ensure that the Task has been scheduled where expected, based on the rules the task is scheduled in the next available slot that can hold the duration. Therefore it should be scheduled at 13:00 hours.

6) Add a task with the following details:
   Description = Read chapter 3
   Due Date = ASAP
   Duration = 1 hour
   Priority = 3

7) Add a task with the following details:
   Description = Cancel Credit Card
   Before = 21/4/04 at 12:00
   Duration = 1 hour
   Priority = 2
Instructions for code revision 2

You have a maximum of 1 hour to complete this problem, if you run out of time please provide some indication of where you think the problem is.

i) Switch to the directory /rev2
ii) Compile the source using: compile
iii) Fill out the BUG DATA COLLECTION FORM upto and including Question 2

You are now ready to begin the experiment.
You are free to view or add code to MAPO at any time
You are free to restart the experiment at any time
You are free to perform actions other than those specified below. The tasks specified below are the minimum you should complete that will trigger a bug if one exists.
You should fill out the relevant sections of the BUG DATA COLLECTION FORM whenever the need arises, i.e. as soon the form requires you to write a response.

0) Make sure the debugger is not still running!
1) Open a terminal and start the debugger: java debugger.Main (this will pop up a debugging Frame)
2) You will need a single terminals for this code revision, change directory to /rev2
3) Type: start1.bat (this will start an organiser for user John)
4) Add a task with the following details:
   Description = Clean my desk
   Due Date = End of Week
   Duration = 2 hours
   Priority = 2
5) Add a task with the following details:
   Description = Read chapter 3
   Due Date = ASAP
   Duration = 1 hour
   Priority = 3
6) Add a task with the following details:
   Description = Dream of a sandy beach
   Due Date = End of day
   Duration = 1 hour
   Priority = 1

Figure B.5: Instructions for Completing Experiment (Page 5)
Instructions for code revision 3

Compilation needs to be done in Unix/Linux as I have used some perl scripts to add code to the agents. However you can run (and I recommend you do) the application + debugger from Windows.

i) Switch to the directory /rev3
ii) Compile the source using: compile
iii) Fill out the BUG DATA COLLECTION FORM upto and including Question 2

You are now ready to begin the experiment.
You are free to view or add code to MAPO at any time
You are free to restart the experiment at any time
You are free to perform actions other than those specified below. The tasks specified below are the minimum you should complete that will trigger a bug if one exists.
You should fill out the relevant sections of the BUG DATA COLLECTION FORM whenever the need arises, i.e. as soon the form requires you to write a response.

0) Make sure the debugger is not still running!
1) Open a terminal and start the debugger: java debugger.Main (this will pop up a debugging Frame)
2) You will need a single terminals for this code revision, change directory to /rev3
3) Type: start1.bat (this will start an organiser for user John)
4) Add a task with the following details:
   Description = Bake a cake
   Due Date   = ASAP
   Duration   = 1 hour
   Priority   = 2
5) Add a task with the following details:
   Description = Take a nap
   Due Date   = End of Day
   Duration   = 4 hour
   Priority   = 2
6) Add a task with the following details:
   Description = Go jogging
   Due Date   = End of Week
   Duration   = 1 hour
   Priority   = 3
7) Add a task with the following details
   Description = Clean desk
   Due Date   = End of Day
   Duration   = 1 hour
   Priority   = 3
Instructions for code revision 4

Compilation needs to be done in Unix/Linux as I have used some perl scripts to add code to the agents. However you can run (and I recommend you do) the application + debugger from Windows.

i) Switch to the directory /rev4
ii) Compile the source using: compile
iii) Fill out the BUG DATA COLLECTION FORM upto and including Question 2

You are now ready to begin the experiment.
You are free to view or add code to MAPO at any time
You are free to restart the experiment at any time
You are free to perform actions other than those specified below. The tasks specified below are the minimum you should complete that will trigger a bug if one exists.
You should fill out the relevant sections of the BUG DATA COLLECTION FORM whenever the need arises, i.e. as soon the form requires you to write a response.

0) Make sure the debugger is not still running!
1) Open a terminal and start the debugger: java debugger.Main (this will pop up a debugging Frame)
2) You will need 3 terminals for this code revision, change directory to /rev4 in each
3) Type: start1.bat (this will start an organiser for user John)
4) Type: start2.bat (this will start an organiser for user Sam)
5) Type: start3.bat (this will start an organiser for user Dave)
6) Switch to user: Dave and add a meeting with the following details:
   Description = Talk about possible PhD topics
   Due Date = ASAP
   Priority = 3
7) Switch to user Sam and add a task with the following details:
   Description = Do shopping
   Due Date = End of Day
   Duration = 2 hour
   Priority = 2
6) Add a task with the following details:
   Description = Go jogging
   Due Date = Before 21/4/4 at 12:00am
   Duration = 1 hour
   Priority = 2
7) Switch to user John and add a task with the following details
   Description = Clean desk
   Due Date = End of Day
   Duration = 1 hour
   Priority = 3

Figure B.7: Instructions for Completing Experiment (Page 7)
Appendix C

Bug Data Collection Form
APPENDIX C. BUG DATA COLLECTION FORM

Date: ____________       Name:

BUG DATA COLLECTION FORM

Instructions:
Questions 1 and 2 are to be filled out immediately BEFORE you begin to interact with the code. Once this has been done you are to interact with the code and fill in the remaining questions 
as soon as you determine the answers, don’t leave filling out the form until the end of your interactions! If you need more space to answer a question please turn this page over.

(Q.1) Code Revision Number: __________

(Q.2) Current Time __________
(I.e. the time just before starting the experiment for this code revision)

(Q.3). Do you believe this code revision is functioning correctly?
Code functioning correctly.
Code is NOT functioning correctly.

(Q.4). Current Time __________
(I.e. the time you determined the answer to 3 above)

(Q.5). Please describe what you believe the problem (if any) is. The description should be in terms of incorrect behaviour experienced by the user. For example, “User added a task however the task was not added to the diary.” You can be as detailed as you need and can refer to the documentation if you wish. You may also wish to provide your initial thoughts on what you think the problem is.
(more space available for your answer on the back of the page)

(Q.6). Current Time __________
(I.e. the time after writing the explanation)

(Q.7). Please describe what you believe the problem is in terms of the actual code. For example, “agent X was supposed to send Message M to agent Y however message M was never sent. Message M was not sent because the plan that was supposed to send Message M failed”. If you are able to indicate the exact cause, such as the offending line(s) or statements please do so. Providing a fix will also be very useful.
(more space available for your answer on the back of the page)

(Q.8). Current Time __________
(I.e. the time immediately after describing the problem in Question 7 above)

Figure C.1: Data Collection Form
Bibliography


Bates, P. (1987), EBBA Modelling Tool a.k.a Event Definition Language, Technical report, Department of Computer Science University of Massachusetts, Amherst, MA, USA.


Cabac, L. (2003), Modeling agent interaction protocols with AUML diagrams and Petri Nets, Master’s dissertation, University of Hamburg Department of Computer Science.


Tambe, M. (1996b), Teamwork in real-world, dynamic environments, Technical Report ISI/RS-96-446, USC Information Sciences Institute, Marina del Rey, California, USA.


