Debugging Multi-Agent Systems Using Design Artifacts: The Case of Interaction Protocols

David Poutakidis
davpout@cs.rmit.edu.au

Lin Padgham
linpa@cs.rmit.edu.au

Michael Winikoff
winikoff@cs.rmit.edu.au

RMIT University
Melbourne, Australia

ABSTRACT

Debugging multi-agent systems (which are concurrent, distributed, and consist of complex components) is difficult, yet crucial. We propose that the debugging process can be improved by following an agent-oriented design methodology, and then using the design artifacts in the debugging phase. We present an example of this scheme which uses interaction protocols to debug agent interaction. Interaction protocols are specified using AUML and are translated to Petri nets. The debugger uses the Petri nets to monitor conversations and to provide precise and informative error messages when protocols aren't correctly followed by the agents.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging—*Testing tools, Debugging aids*; I.2.11 [**Artificial Intelligence**]: Distributed Artificial Intelligence— *Intelligent agents, Multiagent systems*; D.2.2 [**Software Engineering**]: Design Tools and Techniques—*Petri nets*

1. INTRODUCTION

Intelligent agents are an important technology. Based on foundational work in AI and Philosophy, agent technology has significant applications in a wide range of domains [9] and is seen by some as a natural successor to object-oriented programming [8].

As is sometimes forgotten, agents are software and developing agents is developing software [24]. An important phase of developing software is debugging it: it is suggested that debugging and testing may occupy between 25 and 50 percent of the total cost and time of system development with much of this time spent locating the cause of a problem [1, 25].

Multi-agent systems are inherently complex due to the non-deterministic behaviour that these systems can exhibit and the fact that the components of these systems may interact in flexible and sophisticated ways. Therefore the debugging process is more complex and more important than for traditional systems. Despite this, very little effort has been applied to developing appropriate debugging tools and techniques. Unfortunately, debugging multi-agent systems without good debugging tools is highly impractical.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
AAMAS'02, July 15-19, 2002, Bologna, Italy.
Copyright 2002 ACM 1-58113-480-0/02/0007 ...$5.00.

Current debugging tools mostly use information gathering and visualisation to present a graphical depiction of system behaviour to the programmer, so they can understand how the system and the agents are behaving and interacting. The focus is on the collection of information, usually agent messages, and the presentation to the user with filtering applied to the messages [14, 15, 11]. This work however, has not adequately addressed the difficulties of debugging multi-agent systems. Some limitations of current multi-agent debugging techniques are:

- Programmers are generally presented with too much information making it difficult to understand what is really happening in the system.
- Without a proper 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 difficult.

One way that we understand a system is by looking at the design documents produced in the pre-implementation stages of system development. During the design phase models of the various components of the system are built. These models are supported with diagrams ranging from static class diagrams to sequence or interaction diagrams [7]. These models represent the developers' description of the system and are used to explain how the system should operate, they are also used by programmers to facilitate the coding process.

Our central thesis is that 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.

We have identified a number of design artifacts that could be used to facilitate this process including protocols, interaction diagrams, scenario diagrams, capability diagrams, agent overview diagrams, plan descriptors, etc. In this paper we focus on debugging agent communication by using interaction protocols. Communication is an integral component of multi-agent systems, and debugging the communication that occurs between agents can be quite difficult (for example see [5]).

We begin by presenting an overview of interaction protocol design within an agent based software engineering methodology. We then describe (using a running example) how interaction protocols specified using AUML notation can be used to debug message exchanges between agents by translation to Petri nets and using the Petri nets to monitor conversations.

2. DESIGNING AGENT SYSTEMS

2.1 Agent Oriented Software Engineering

Software engineering techniques aimed at the development of agent based systems have recently emerged. Castro, Kolp and Mylopoulos [3] provide 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, these ideas are manifested in the Tropos methodology. Wooldridge, Jennings and Kinny [23] also provide a methodology that the analyst can follow from requirements to implementation and they also focus on human-like characteristics, in particular the BDI framework. However, they view requirements as an independent process rather than driving the design as in Tropos. Other work has identified the need for agent oriented software engineering [19, 22].

Over the last several years we have been developing an agent oriented methodology in collaboration with Agent Oriented Software[1] called *Prometheus* [18] Prometheus is a detailed and complete ("start-to-end") process for specifying, designing, and implementing intelligent agent systems. It is this methodology that we use in our investigation into the use of interaction protocols within a debugging agent.

2.2 Interaction Protocols

One of the artifacts that is developed when following the Prometheus methodology is a set of interaction protocols (IPs) that capture the interaction patterns between agents.

The starting point for developing the interaction protocols are a set of use case scenarios which describe at a high level the steps the system will go through to achieve the desired functionality. For example, consider an online store selling books. One important use case involves a customer purchasing a book. This might be described as follows:
Use case: Sell (ID1)
Description: The customer buys a book.
Steps:

1. Customer places an order for a book.
2. Customer specifies delivery option.
3. Shop assistant checks customer's credit card details.
4. Shop assistant thanks customer and places order.

Additional fields in a use case specification (not shown here) include a context, description of information read and written, and variations on the use case.

Once the developer has determined which agent types will be used in the system, interaction diagrams can be developed based

[1]A company which markets the agent development software platform JACK^{TM} as well as agent solutions

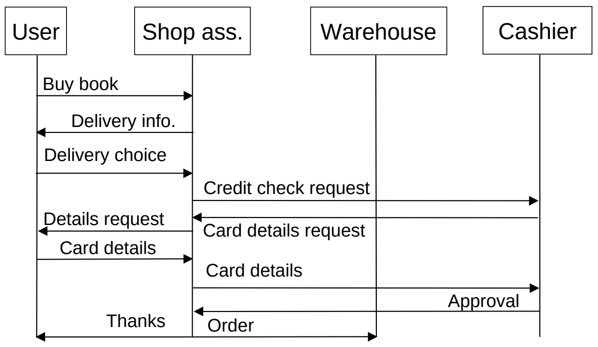


Figure 1: Interaction Diagram

on these use case scenarios, indicating the flow of messages (or interactions), between agents. Where a use case step performed by agent A is followed by a step performed by a different agent B the interaction diagram will have a message from A to B. Figure 1 shows an interaction diagram based on the use case above.

Interaction diagrams (like use case scenarios) show only one particular way the interaction may happen. Theoretically a full set of interaction diagrams could show all possibilities and would thus fully define the inter-agent interactions. However it would be tedious to develop a complete set of interaction diagrams and it would be difficult for the reader to comprehend the large number of such that would be generated. Instead we generalise sets of interaction diagrams into protocols by extending the notation with additional features; for example allowing a specification of choice points where interactions diverge, and allowing a specification of parallelism where temporal ordering can vary. This allows us to succinctly specify all possible sequences of interactions within an interaction pattern initiated for a particular purpose.

Figure 2 shows an example of a protocol[2] which is used as part of the interaction diagram in figure 1. A merchant asks a bank to authorise a credit card transaction. The bank replies asking for details which the merchant supplies (perhaps getting the user to supply them). The bank can respond with either approval (*approve*) or rejection (*reject-2*). In one case (where the card is known to be stolen perhaps) the bank will respond with both a *reject-1* message and a *fraud* message (perhaps asking the merchant to confiscate the card). Note that we do not require that the *fraud* and *reject-1* messages be sent in any particular order. We use the AUML [16] notation for specifying protocols. This notation has the advantage of being used (e.g. by FIPA [6]) and is close to UML, and thus presumably easy for a software professional to learn. AUML provides a range of features (in addition to those shown in figure 2) such as merging of messages, a non-exclusive choice, and the ability to specify sequencing of messages.

In order to build the protocol, the developer simply asks the question at each point in the interaction diagram (or developing protocol), whether this message necessarily comes after its predecessor, and whether there is any other response that could happen at this point. By doing this recursively the full protocol is eventually developed. Use cases and interaction diagrams can readily be checked to see that they are covered by the protocol developed.

Note that the protocol of figure 2 could be simplified (for exam-

[2]The labels (e.g. M1, B2) aren't part of the AUML notation, rather they are introduced later in the translation process; we give them here for convenience. The abbreviations for messages (e.g. CCR for "Credit check request") are also used later.

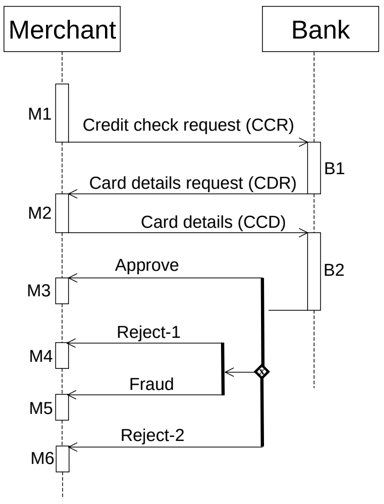


Figure 2: Interaction Protocol

ple by merging messages), however, we intentionally use the given version since it illustrates a range of features which are typical of larger more complex protocols. Note also that we do not handle the full notation as defined by FIPA [6]; rather we have selected the key constructs that allow us to adequately describe a great number of real world protocols.

3. DEBUGGING USING INTERACTION PROTOCOLS

We now present the overall design of a debugging system that monitors the interactions between agents. The key idea is that we monitor the exchange of messages between agents and check them against interaction protocols. Violations of the interaction protocols such as a failure to receive an expected message or receiving an unexpected message can then be automatically detected and precisely explained (e.g. "agent X received message m which was unexpected – the agent was participating in protocol P and was expecting either n or l").

We introduce a central debugging agent into a standard multi-agent system. This debugging agent is responsible for monitoring all interactions (messages sent) between the agents in the system. Therefore we require that carbon copies of any messages sent by agents also be sent to the debugging agent. Introducing the debugging agent has minimal impact on the behaviour of the system – if the system is highly timing dependent then the additional time taken to transmit a carbon copy of messages may affect the system's behaviour. In particular, the time difference between a message being transmitted and being received might be significant. For the moment, we assume that the system isn't timing dependent. Extending our method to deal with time-dependent systems can be done by separating message sending and message receiving into two different "events" that are significant to the debugging agent.

The actual representation of the message is not specified, however the debugging agent needs to be able to determine the following information: *sender, receiver, message type* (e.g. cfp, propose), and *conversation id*. The need for *sender, receiver* and *message type* should be clear, however the *conversation id* may need a little justification. The *conversation id* uniquely identifies each conversation within the multi-agent system. An extract from FIPA's proposed standard [6], explains the need for a *conversation id*:

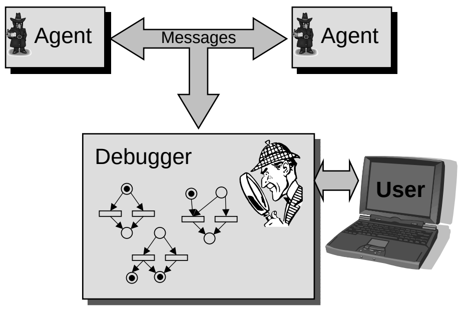


Figure 3: Debugging System Design

"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, with different agents, within different IPs. . . ." ([6], end of 2.1, IP = Interaction Protocol)

For the above reason we require that the agents are aware of the conversations that they are engaged in, therefore each message must belong to a particular conversation, hence the *conversation id*.

The debugging agent represents protocol instances internally as *Petri nets* [20]. We choose to do this, rather than use AUML, for two reasons. Firstly, Petri nets are a precise notation with clear formal semantics whereas AUML is not precisely defined. Secondly, by using a general underlying protocol representation which is decoupled from AUML we allow ourselves to use other protocol representations (including future versions of AUML) so long as they can be translated into Petri nets. Note that the AUML presentation of the protocols should be used for visualisation purposes rather than the translated Petri nets.

The overall architecture is depicted in figure 3. The debugging agent has a library of known protocols (represented as Petri nets). When a new conversation is started (i.e. the debugging agent received a message which isn't part of an existing conversation) the debugging agent instantiates all protocols which are capable of beginning with the received message[3]. There will be a set of protocol instances for each conversation. In many cases, where message types are unique to each protocol, the set of protocol instances for a given conversation will consist of a single protocol instance. We return to the debugging agent and how it tracks protocol execution using Petri nets in section 5.

4. REPRESENTING AND INTERPRETING PROTOCOLS

In this section we present a process for converting a protocol specified in (a subset of) AUML to an equivalent Petri net version. The translation process represents each state of the protocol and each message of the protocol as a unique place in the Petri net; the translation potentially introduces additional places.

We believe that Petri nets are an appropriate representation because they are able to capture concurrency and adequately deal with the need to be in multiple states simultaneously (for example M4 and M5 in figure 2); something which finite state machines are unable to represent. Also, they appear to be sufficiently powerful to

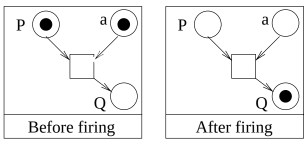
[3]Unless the protocol used is specified in the message, something that FIPA allows but does not require.

capture the necessary building blocks of protocols. The translation given is *local* in that each part of the AUML protocol corresponds directly to a part of the Petri net. Finally, 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.

4.1 Petri Nets

We briefly introduce the Petri net notation (named after Carl Adam Petri). A Petri net [20] consists of places (depicted as circles) and transitions (depicted as rectangles) which are linked by arrows. Additionally, places may contain tokens (depicted by •) The placement of tokens on a net is its *marking* and executing ("firing") a Petri net consists of moving tokens around according to a simple rule; the places, transitions, and the links between them remain unchanged.

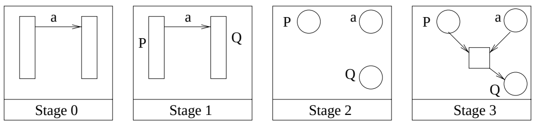
A transition in a Petri net is *enabled* if each incoming place (i.e. a place with an arrow going to the transition) has at least one token. An enabled transition can be *fired* by removing a token from each incoming place and placing a token on each outgoing place (i.e. each place with an arrow from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



4.2 Identifying and labelling places

The first stage of converting the protocol to an equivalent Petri net is to identify the places. In our Petri net protocols we have two types of places, corresponding to protocol states and to messages. The points on the agent lifeline that are connected by messages are converted to places. The messages themselves make up the second type of place and are labelled with the message type.

The figure below illustrates how the places are derived from the AUML representation. Stage 0 is the original interaction. It allows the agent on the left to send a message a to another agent. In Stage 1 we identify the protocol states and name them. Stage 2 converts each state and agent message into a place. Stage 3 involves taking the connections between the agent lifelines and adding the necessary transitions (possibly adding additional places as discussed below). The connector in this example is a single arrow and is converted to a single transition. The transition is joined to the places as shown below.

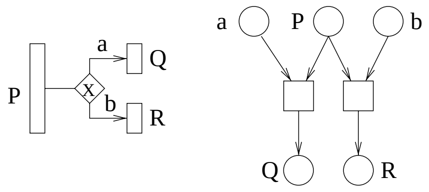


Note that the initial and final states of the protocol need to be identified. These can be easily extracted from the protocol specification, and translated places suitably marked. An initial state is one with no incoming messages. A final state is one with no outgoing messages. These are used to detect certain erroneous conditions.

4.3 Translating transitions

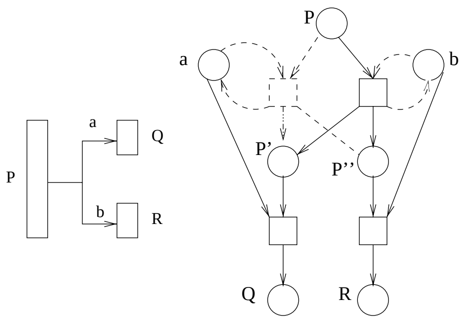
The transitions in a typical protocol will not be as simple as those shown above. Specifically, AUML allows for nested connectors. We present here a set of mappings from AUML messages to Petri nets. The underlying intuition is that a message is handled by adding a token to the corresponding message place, this should then enable the Petri net to fire transitions which simulate the changes in protocol state.

Selection (or exclusive or) as shown below provides the agent with a choice as to which message can be sent. We say that if the protocol is at state P it can send message a and will move into state Q, or it can send message b and move into state R (but not both). The Petri net version of the selection connector is presented alongside the AUML version. Note that we have the standard state places for P, Q and R. We also need the message places a and b. Therefore the selection connector adds a transition for each choice and the transition has input from the associated message and the current state P.

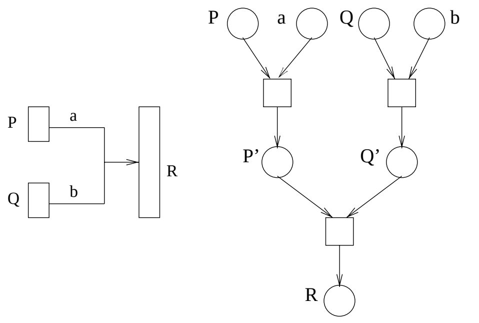


The parallelism connector as shown below requires that both message a and message b are sent when in state P, and the order is not specified. The translation (ignore the dotted transition and links for the moment) adds two extra places, P' and P'', these places essentially split the processing so that the agent can be in two states at any time (namely Q and R). This is precisely the behaviour we want when performing a parallel operation. The addition of these two places requires we add three transitions: from P to P' and P'', and from each of P' and P'' to Q and R respectively.

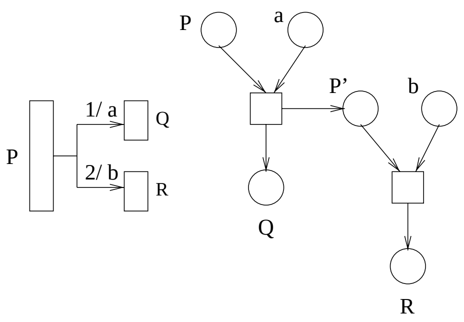
This translation (ignoring the dotted components) has a problem – the transition from P to P' and P'' is not caused by a received message. Although in this context it is safe to fire the transition, in other contexts (such as the interaction protocol in figure 2) this choice cannot be made up front. Effectively this translation introduces a "look-ahead". We fix this by requiring that the transition from P be triggered by a message – in this case either a or b. However, the transition from P to P' and P'' should not consume the triggering message and we achieve this by adding the message as an outgoing place of the transition. This improved translation (including the dotted components) has the correct behaviour and does not require any look-ahead. Note that the introduction of the additional places is necessary to capture the intention of the AUML notation (that the protocol be able to independently move into states Q and R) and does not reflect a limitation of the Petri net notation.



Synchronisation (below) restricts a protocol from continuing to the next state until it has sent both messages. The messages can come from different agents and the order is not specified. We have introduced two new states, P' and Q' to ensure the agent cannot move into state R until both messages are received. Note that this is the dual of parallelism (ignoring the dotted components).



Finally, the sequencing connector is similar to parallelism, but does specify an ordering of messages that needs to be followed. The *1/a* and *2/b* require that message a is sent first followed by message b. The translation is given below.



4.4 An Example

In figure 2 we presented a protocol specification for a credit card exchange between two agents. Using the process outlined in the previous two sections we convert this protocol into the Petri net equivalent so it can be used by the debugging agent. This protocol is a good example as it includes a number of the connectors we have discussed. Furthermore, the protocol includes a nesting of these connectors illustrating how non-trivial cases are handled by the translation scheme presented.

The translation creates places corresponding to each labelled protocol state and to each message. We then add transitions corresponding to the connectors. Figure 4 shows the completed Petri net. The transitions between M1 and B1, B1 and M2, M2 and B2 are straightforward. However B2 is connected to M3, M4, M5, M6 and there is a nesting of the connectors (see figure 2). We say that at B2 the Bank can send either an *Approve* message, a *Reject-2* message or *both* a *Reject-1* and a *Fraud* message (in any order). Our Petri net in figure 4 expresses these rules properly. It should be obvious that at each connector we have simply inserted the appropriate transition rule as detailed in section 4.3. For instance, at B2 we have the selection connector, therefore we have an input into each of the transition boxes. The parallel connector also takes inputs from B2 and it should be obvious that the bottom half of figure 4 is identical to the parallel transition depicted in section 4.3.

5. EXECUTION OF THE DEBUGGING AGENT

Having translated the protocol into an equivalent Petri net representation we now discuss how the Petri net representation can be used to track protocol states, and how errors (of various types) can be detected.

Recall that the debugging agent has a library of known protocols that it uses to detect errors. The debugging agent also keeps a list of current conversations, when a message is received it is added to the appropriate conversation and is processed. During a conversation the debugging agent does not know what protocol the agents

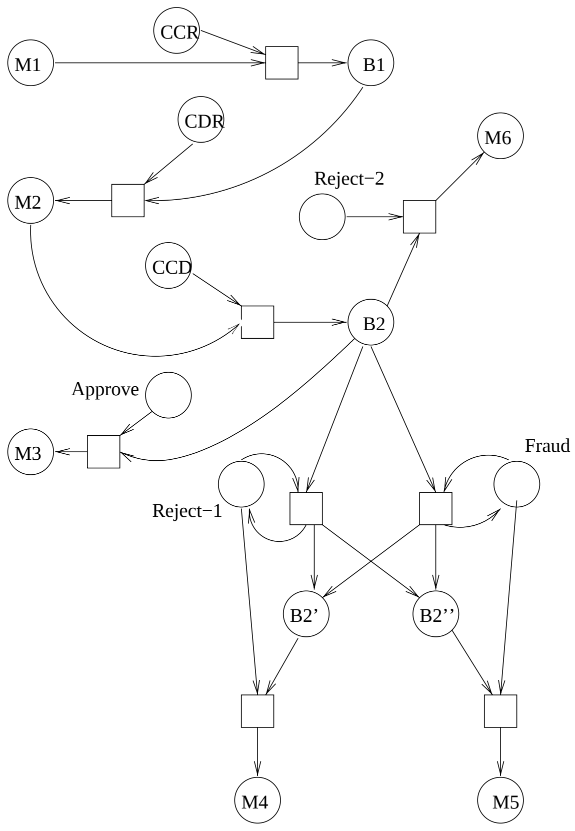


Figure 4: Credit Card Example

are following[4], therefore it keeps a list of *possible protocols*. These *possible protocols* are the protocols from the library that currently match the sequence of messages within a conversation. As the conversation progresses the *possible protocols* list is reduced whenever a message is received that causes an error in the individual protocol. At the end of a conversation it is expected that a single correct protocol remains in the *possible protocol* list.

The execution of the debugging agent consists of the following repeated cycle:

1. Receive a message m
2. If the message does not belong to any conversation (determined by examining the *conversation id*) create a new conversation by instantiating all matching protocol instances and initialising them (by placing a token on the initial state of each protocol instance)
3. If the message belongs to an existing conversation then add the message to each protocol instance in the conversation
4. Fire each Petri net within the conversation's *possible protocol* list until no more transitions are enabled
5. Examine the Petri nets for erroneous conditions (see below)

There are a number of different erroneous conditions that can be detected. One of these is an unexpected message which corresponds to an unprocessed message. If after firing a Petri net any

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

message place has tokens then that message was unexpected and the protocol instance should be deleted from the *possible protocol* list.

Another detectable erroneous condition is where a protocol is considered to have completed but there are tokens on non-final places.

Finally, if the set of protocol instances corresponding to a conversation becomes empty report an error. This could be caused by an entirely unknown message being received, in which case all entries in the *possible protocol* list would be removed in one cycle. For example, if the bank sent a *cfp* message.

Consider now the following sequence of messages between a merchant and a bank:

- Merchant to bank: credit card request (CCR)
    1. A new protocol instance is created and a token placed on M1.
    2. A token is placed on CCR
    3. The net is fired which removes the two tokens (on M1 and CCR) and places a token on B1
- Bank to merchant: credit details request (CDR)
    1. A token is placed on CDR
    2. The net fires removing tokens from CDR and B1, and placing a token on M2
- Merchant to bank: credit card details (CCD)
    1. A token is placed on CCD
    2. The net fires removing both tokens, and placing a token on B2
- Bank to merchant: approve
    1. A token is placed on the Approve place
    2. The net fires removing both tokens and placing a token on M3
- Bank to merchant: fraud
    1. A token is placed on the Fraud place
    2. No transitions are enabled. In particular, since there is no token on B2 no transition which uses *Fraud* can fire
    3. The debugging agent reports a bug: the fraud message was unexpected – the merchant wasn't expecting any further messages.

We have implemented a debugging agent using the algorithms presented and have tested it on the example above (for which it indicated an error at the appropriate place). We have also converted and tested several FIPA interaction protocols, including the call for proposal, request interaction and query interaction protocols. Our debugging agent was able to identify incorrect message exchange whenever it occurred. We have also performed testing to ensure that correct message exchanges are not erroneously flagged by the debugging agent.

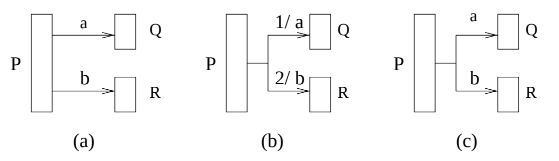


Figure 5: Different interpretations of an ambiguous interaction

6. DISCUSSION

Debugging multi-agent systems is challenging and good tool support is highly desirable. We believe that integrating debugging with the larger picture of agent oriented software engineering will enable us to move beyond the current generation of visualisation and debugging tools. Specifically, more precise information can be provided to the programmer by making use of design artifacts during debugging. In this paper we looked at the use of interaction protocols for debugging a multi-agent system and presented a design for a debugger which "eavesdrops" on agent conversations and is able to give precise and informative error messages when protocols aren't adhered to.

We noted that the underlying AUML notation was not a suitable representation for the debugging agent and so we defined a translation from (a subset of) AUML to Petri nets. This translation is of value in its own right since it gives the AUML subset precise formal semantics, and since it allows tools for checking properties of Petri nets (e.g. livelock, deadlock, etc.) to be applied to AUML protocols. Furthermore it provides a systematic general method for addressing the issue of detecting failures by showing how specifications can be automatically converted into monitoring components.

Although the translation presented addresses many of the more common AUML constructs, there are a number of AUML constructs which are not handled. In some cases extending the translation is a simple matter, in other cases the extension is more involved. For example, AUML's multiplicity connector (denoted by a message followed by an asterisk) means that the message can be sent an arbitrary number of times and can be accommodated by an extension of our translation fairly easily. On the other hand, preconditions require additional information that is not easily extracted from the messages: the debugging agent cannot know when an arbitrary condition becomes true since it doesn't have access to the internal beliefs of other agents.

One problem that we encountered in our work with AUML is the incompleteness of the specification. The definitions in (e.g.) [6] lack precise semantics and are ambiguous in a number of places. For example, figure 5(a) shows what we believe to be a different way of representing sequencing. We are assuming that message a must be sent before message b (equivalent to figure 5(b)) However, another plausible interpretation of figure 5(a) is that there is no ordering constraint implied (i.e. equivalent to figure 5(c)). By using the formal semantics of the Petri net we are removing the ambiguity from much of the AUML notation. A potential drawback with translating a design specification that has weak semantics into one with stronger semantics is that it relies on making assumptions that may not be intended by the original developers. We avoid this impact by only providing translations for constructs that are unambiguous. Future work will involve interaction with both the FIPA and AUML communities to clarify and strengthen their specifications so that we may provide a more complete set of translations.

A final important component of AUML which we have looked into but have not yet formalised in this paper is that of cardinality of messages or more simply, more than two agents engaging in a conversation. This is typical of auction protocols where there is an initiating agent that requests a bid from a number of participating agents. To effectively debug such a conversation we need to include all of the participant within a single Petri net. This introduces the issue of tracking which message belongs to which agent. A possible solution to this is to use coloured Petri nets. Each agent pair, i.e. initiator and participant, is represented by a coloured[5] token and any message they send results in a token of this same colour being placed in the associated message place. To ensure messages match up with tokens we add some logic into the transitions that does a simple check to see if all incoming tokens are of the same colour.

6.1 Related Work

It has been argued that Multi-Agent systems merely represent a specific form of distributed systems [17]. Several methods have been developed to assist in the debugging of distributed systems: recording a history of execution for analysis or replay [10], animating the execution of a system at runtime by providing a visual representation of the program [2], race detection algorithms to facilitate the detection of simultaneous access to shared resources, model checkers and declarative debuggers to automate the process of verifying the expected behaviour of execution [21, 13].

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 specific to agent systems that require specific attention. Traditional distributed systems support distributed information and processes whereas multi-agent systems address distributed tasks. 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.

The current state of multi-agent debugging has not thoroughly addressed the issues specific to agents. The most common approach to date has been to borrow visualisation techniques and customise them to the needs of multi-agent systems. Visualising system behaviour is primarily concerned with providing a visual representation of the message exchange between the agents in a multi-agent system [14, 11, 15]. This method does not provide rich enough debugging information and does not address the issue of knowing what information is necessary for debugging. The result is that the programmer is presented with too much information and experiences information overload reducing the effectiveness of the visualisation technique. Liedekerke and Avouris [11] 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 an attempt to provide useful debugging information that could be absorbed effectively by a programmer Nwana et al [15] provide debugging tools based on multiple views of the computation. The intention is that by combining results from different views the programmer will be better able to identify incorrect system behaviour. Providing different views does limit the information flow to some degree, however, it does not overcome the problem.

Although existing visualisation debuggers provide essential infrastructure, they do not solve the underlying issue: how to effectively filter the information presented to the programmer, so as to avoid overloading her with too much information while not missing out on vital information which would aid in diagnosing and locating errors.

[5]Not literally, tokens are coloured in the sense of carrying additional information beyond their existence, not in the sense of being red, blue, or yellow.

We are not the first to suggest the use of Petri nets in specifying agent communication (see e.g. [4, 12]). Nowostawski et al. [12] also use their petri nets for debugging purposes, however the details of this are not explained and there is no systematic approach for developing the protocols. This is where our approach differs, we have provided a method for translation from AUML (which is more likely to be used by practising software engineers who are familiar with UML) to Petri nets that should enable developers to construct, or convert their own protocols into an equivalent Petri net that can be used for debugging within our debugging agent.

6.2 Future Work

The first item of future work is to integrate the debugging agent into a number of multi-agent environments and assess it's ability to detect incorrect interaction and provide useful debugging information in a variety of environments. We will also be continuing our work on providing translations for as many of the AUML constructs as possible, as stated earlier this will require interaction with the FIPA and AUML communities.

Note that although we have discussed checking agent interaction against interaction protocols, the use of sets of interaction protocol instances (*possible protocols*) enables the debugging agent to also check agent messages against interaction diagrams. Although in this case a violation should be viewed as a warning rather than as an error, since interaction diagrams are not intended to be complete.

Interaction protocols are but one instance of a design artifact that can be beneficially used in debugging. We are investigating how other design artifacts found in agent oriented methodologies such as Prometheus [18] can be used for debugging multi-agent systems.

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

7. REFERENCES

[1] B. W. Boehm. Software engineering economics. *Prentice Hall, Englewood Cliffs, NJ*, 1981.

[2] Bernd Bruegge, Tim Gottschalk, and Bin Luo. A framework for dynamic program analyzers. *Published in Conference on Object-Oriented Programming Systems, Languages, and Applications. OOPSLA*, pages 65–82, 1993.

[3] J. Castro, M. Kolp, and J. Mylopoulos. A requirements-driven development methodology. *In Proc of the 13th International Conference on Advanced Information Systems Engineering CAiSE Interlaken, Switzerland*, 2001.

[4] R. Scott Cost, Ye Chen, Tim Finin, Yannis Labrou, and Yun Peng. Using colored petri nets for conversation modeling. In *Workshop on Agent Communication Languages at the Sixteenth International Joint Conference on Artificial Intelligence (IJCAI-99)*, 1999. Available from *http://www.csee.umbc.edu/∼jklabrou/*.

[5] David Flater. Debugging agent interactions: a case study. In *Proceedings of the 16th ACM Symposium on Applied Computing (SAC2001)*, pages 107–114. ACM Press, 2001.

[6] Foundation for Intelligent Physical Agents (FIPA). FIPA interaction protocol library specification. Available from *www.fipa.org*, 2001. Document number XC00025D, version 2001/01/29.

[7] Martic Fowler and Kendall Scott. UML distilled. *Addison-Wesley. ISBN 0-201-32563-2*, 1997.

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

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

[10] T. LeBlanc, J. Mellor-Crummey, and R. Fowler. Analyzing parallel program executions using multiple views. *Journal of Parallel and Distributed Computing*, 9:203–217, 1990.

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

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

[13] Lee Naish. A declarative debugging scheme. *Journal of Functional and Logic Programming*, 1997(3), April 1997.

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

[15] Hyacinth S Nwana, Divine T. Ndumu, Lyndon C. Lee, and Jaron C. Collis. ZEUS: a toolkit and approach for building distributed multi-agent systems. In *Proceedings of the Third International Conference on Autonomous Agents (Agents'99)*, pages 360–361, Seattle, WA, USA, 1999. ACM Press.

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

[17] G. O'Hare and M. Wooldridge. A software engineering perspective on multi-agent systems design: Experience in the development of MADE. *Distributed Artificial Intelligence: Theory and Praxis*, pages 109–127, 1992.

[18] Lin Padgham and Michael Winikoff. A methodology for agent oriented software design. Technical Report TR-01-2, School of Computer Science and Information Technology, RMIT University, 2001.

[19] Charles Petrie. Agent-based software engineering. *Invited talk: Proc. PAAM 2000, Manchester, April, 2000, published in Agent-Oriented Software Engineering, Eds P Ciancarini and M. Wooldridge, Lecture Notes in AI 1957, Springer-Verlag*, pages 58–76, 2001.

[20] Wolfgang Reisig. *Petri Nets: An Introduction*. EATCS Monographs on Theoretical Computer Science. Springer-Verlag, 1985. ISBN 0-387-13723-8.

[21] Reinhard Schwarz and Friedemann Mattern. Detecting causal relationships in distributed computations: In search of the holy grail. *Distributed Computing*, 7(3):149–174, 1994.

[22] M. Wooldridge and P. Ciancarini. Agent-oriented software engineering: The state of the art. *In P. Ciancarini and M. Wooldridge, editors, Agent-Oriented Software Engineering. Springer-Verlag Lecture Notes in AI*, 1957, 2000.

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

[24] Michael Wooldridge and Nicholas R. Jennings. Pitfalls of agent-oriented development. In K. P. Sycara and M. Wooldridge, editors, *Agents'98: Proceedings of the Second International Conference on Autonomous Agents*. ACM Press, 1998.

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