Citation: Poutakidis, D, Padgham, L and Winikoff, M 2003, 'An exploration of bugs and debugging in multi-agent systems', in Foundations of Intelligent Systems, Maebashi City, Japan, 28-31 October 2003.

An exploration of bugs and debugging in multi-agent systems

David Poutakidis, Lin Padgham, and Michael Winikoff

RMIT University, Melbourne, Victoria, Australia,
{davpout,linpa,winikoff}@cs.rmit.edu.au

Abstract. Debugging multi-agent systems, which are concurrent, distributed, and consist of complex components, is difficult, yet crucial. In earlier work we have proposed mechanisms whereby protocol specifications available from the design process can be used for monitoring the execution of the multi-agent system they describe. Protocol specifications can be used at run-time for reporting any discrepancies in interactions compared to that which was specified. In this paper we describe and categorise a range of bugs found in actual multi-agent systems developed by students in an Agent Oriented Programming and Design class. We then indicate how these bugs manifest to the debugging agent and what information it is able to provide to the user to assist in locating and diagnosing the problem.

1 Introduction

Based on foundational work in AI and Philosophy, *agent technology* has significant applications in a wide range of domains [5]. Agent based software engineering methodologies [1] have emerged to facilitate the development of agent based applications. These applications are often developed as multi-agent systems and are notoriously difficult to debug since they are distributed, concurrent, and complex. Under these circumstances good debugging tools are essential.

Despite this, little work has focused on debugging multi-agent systems. Most of the existing prototype or research tools rely on visualisation of the messages passing between agents, possibly with some filtering applied to these [4]. However these are often not useful as there is too much information for the developer to be able to detect that there may be a problem.

Exception detection and resolution and domain independent exception handling services [6] is a similar domain to debugging. However the techniques are not appropriate for multi-agent systems that are not the traditional problem solving or workflow systems seen in [6]. Many agent system do not have a concrete representation of tasks, resources and other properties required to make use of these exception handling services.

In an earlier paper [2] we proposed that protocol specifications, produced during the design stages of system development, could be used during debugging to monitor what message exchanges are expected and to locate bugs by detecting unexpected patterns of message exchange. We proposed a mechanism for taking protocol diagrams developed in Agent UML (AUML) [3], converting them to equivalent Petri nets, and then using

them to monitor execution and detect problems. The protocol diagrams being used were typical of those produced during the architectural design phase of the Prometheus design methodology for agent systems [1]. The key point is that instead of presenting messages to the programmer and relying on him or her to detect problems, the debugging agent proposed in [2] monitors conversations and detects problems by identifying messages that are not permissible in the conversation defined by the protocols.

In this paper we describe the results of applying the debugging agent from [2] to actual agent applications. Our aims are to:

- Determine what types of bugs occur in practice.
- Determine to what extent the debugging agent is able to assist with detecting these bugs.
- Apply these results to determine the usefulness of applying design documents in debugging multi-agent systems.

2 A description of the debugging agent

In Poutakidis et. al. [2] we proposed that the design documents (specifically interaction protocols) produced in pre-implementation could be used by a debugging system to provide run-time error detection and debugging support. It was suggested that the design documents could be converted into an internal representation that the debugging agent could use to compare against run-time execution. Execution that differed from what was defined by the design documents is reported to the developer.

Petri nets were used as an internal representation for protocols since they are a formal graphical modelling notation that can easily be interpreted by both the human user and a program. Importantly, Petri nets are also able to capture concurrency. We discussed a process for converting AUML interaction protocols to an internal Petri net form in [2].

The debugging agent takes as input a set of these Petri net protocols, one for each AUML interaction protocol. These Petri net protocols are used by the debugging agent to both monitor interactions for errors and to reason about how the errors may have occurred. This process is discussed in detail in [2].

Whenever a message is received the debugging agent needs to determine which protocol the message belongs to. The decision is facilitated by the parameters that agents are required to include with their messages. Messages that the debugging agent receives must include a *sender, receiver, message type,* and *conversation id.*

Briefly, for each incoming message the debugging agent determines which conversation the message belongs to and a token is then placed on the appropriate *message place*. The Petri Net is then fired to determine the next state of the conversation. If any message places have tokens after firing then that message was not expected and a bug is reported.

3 Debugging with the Debugging Agent

As part of our investigation into the types of bugs that occur in multi-agent systems and the methods that can be used to detect them we examined 15 multi-agent applications

and their associated bug logs. The bugs encontered in the application and the bug logs are presented. For each type of bug we explain what the bug is, what typical errors that programmers make lead to the bugs, any variations of the bug (where relevant), how we tested these bugs, and the results from the tests.

The following is a brief description of the types of bugs we found while inspecting a set of agent programs developed by third year software engineering students. After identifying the bugs we then introduced each of them into a single application. This application simulates an ambulance response service where patients are generated and ambulances are called on to handle requests by picking up patients and taking them to a hospital. The application consists of three agent types: one Controller agent named Controller, ten Ambulance agents named Unit-X, where X is a number between one and ten and an ExternalSimulator agent that is not discussed further in this paper.

Failure to send a message: An example of a failure to send a message that occurred in the test application involved the top level *Dispatcher* plan. This plan is responsible for assigning Ambulance agents to patients and then instructing the Ambulance agents, via a "PickUpPatient" message, to collect the patient and deliver them to a hospital.

There is a logic error that causes the plan to fail, the result is that no message is sent and the Dispatcher stops responding. This is a breach of the protocol however in this instance the debugging agent did not alert us to an error. This is however to be expected given the design of the debugging agent. The debugging agent focuses on debugging interaction and since the debugging agent did not receive any messages it can not know that the agent was supposed to begin a conversation. Therefore, a limitation of our debugging agent is that it is unable to detect a failure to send a message if the message is the first in the protocol.

The first message in a protocol is a special case and although we would like to be able to detect failure when it occurs in this situation, at present we cannot. However the debugging agent is capable of detecting an error if the failure is from any message other than the first message. To test this we modified the test application so that the failure would occur after the conversation had begun. In this example the debugging agent indicated that there was a warning, not an error. Since the lack of a message is detected using a timeout it is not possible to guarantee that the message will not arrive at some point in the future.

Uninitialised agents: When an agent sends a message to another agent that agent must be in the position to receive the message. If the intended recipient has not yet been initialised it will not receive the message and the protocol will not be able to complete.

In our test application when we introduced an error that resulted in one of the agents sending a message to another that had not yet had time to initialise an error message was generated. It is interesting to note that the error message is the same type of error as *failure to send a message*. Although the cause of the bugs are different, the debugging agent cannot tell the difference between the two. The debugging agent is able to determine that an error has occurred but can offer little more advice. This is due to the fact that by sending a message to an uninitialised agent you are guaranteeing that a response will not be returned, hence the bug is presented in the same way.

Sending a message to the wrong recipient: We have seen how the debugging agent deals with the sending of a message to agents that don't yet exist, or will never exist. We

now present the results of sending a message to the wrong recipient where the recipient actually exists but was not the intended recipient..

A message is addressed to the wrong agent, it should be addressed to the Controller agent but is instead addressed to Unit-4 who is a valid agent but the debugging agent does not have it as one of the agents in the role map. Only Unit-1 and the Controller agent are valid agents in this conversation. The token is rejected from the message place and an error message indicating that the agent is not the intended recipient is generated. We are given specific information as to which protocol was being followed and which agent was supposed to receive the message.

Sending the wrong message: In the case that the message does not exist in the protocol and the case that the message is valid in the protocol but not at the time it was sent the debugging agent is able to determine that an error occurred. The test application was modified so that after Unit-1 received a "PickUpPatient" request it would send the wrong message in reply. The debugging agent receives a copy of the wrong message, places it in the associated message place and then fires the net. After the net is fired a token remains in the message place indicating that an error has occurred. The debugging agent informs us that the wrong message was sent and details of the status of the conversation are presented.

Sending the same message multiple times: This can result in unintended consequences and unless the design of the protocol allows for it it should be considered an error. We instructed our agents to send the same message twice in immediate succession. The debugging agent is able to determine that an error has occurred but it does not specify that the error was that the same message was received twice. It is obvious how to extend the debugging agent to handle this as the necessary information is available. The sequence of messages that have been received is known and the debugging agent could check to see if the last two messages are the same.

4 Conclusion

We identified a range of bugs and demonstrated how the debugging agent was able to assist in detecting them. Figure 1 summarises the different bug types that we found, and how well the debugging agent is able to assist in detecting them. As can be seen, the debugging agent is able to detect most of the bugs that we found and in many cases gives precise feedback that assists in localising the cause of the bug.

We then summarised the design of a debugging agent (presented in [2]). Having identified a range of bugs, and described a debugging agent, we then selected a particular application and showed how the debugging agent was able to assist in detecting a range of bugs, both actual, and seeded (based on bugs that were found in other applications). Figure 1 summarises the different bug types that we found, and how well the debugging agent is able to assist in detecting them and in localising their cause. As can be seen, the debugging agent is able to detect most of the bugs that we found and in many cases gives precise feedback that assists in localising the cause of the bug.

The debugging agent is an improvement over existing debuggers in that it doesn't rely on the programmer to interpret information and detect bugs. Rather, it diagnoses bugs itself based on design information. *Our results show that design documents and*

Bug Type	*Debugging Agent*
Uninitialised Agent:	★
Failure to send:	
first message in conversation	✘
not first message	✔
Wrong recipient:	
recipient non-existent	★
recipient exists and first message	★
recipient exists and not first message	✔
Message sent multiple times:	✔
Wrong message sent:	✔

Fig. 1. Bug types and how well the debugging agent handled them. A "✔" or "★" indicates that the debugging agent can handle the bug. A "✔" means that the agent also gave a precise error message, a "★" means that the error message was not precise, but still useful. A "✘" indicates that the bug is not handled.

system models, specifically in this case interaction protocols, can be successfully and usefully applied to debugging agent systems. Future work includes extending to design models other than interaction protocols and extending the debugger to look at events posted *within* agents, in addition to messages *between* agents.

References

1. L. Padgham and M. Winikoff. Prometheus: A Methodology for Developing Intelligent Agents. Proceedings of the Third International Workshop on Agent-Oriented Software Engineering (AOSE), 2002.
2. D. Poutakidis and L. Padgham and M. Winikoff. Debugging Multi-Agent Systems using Design Artifacts: The case of Interaction Protocols. First International Joint Conference on Autonomous Agents and Multi-Agent Systems (AAMAS), 2002.
3. Foundation for Intelligent Physical Agents (FIPA). FIPA Interaction Protocol Library Specification. Document number XC00025D, version 2001/01/29, www.fipa.org
4. M. Liedekerke and N. Avouris. Debugging multi-agent systems. Information and Software Technology, 37(2), pg 102-112.
5. N.R. Jennings and M.J. Wooldridge. Agent Technology: Foundations, Applications, and Markets. Chapter 1 pages 3-28, 1998.
6. Mark Klein and Chrysanthos Dellarocas Exception Handling in Agent Systems. Third International Conference on Autonomous Agents, 1999.