

ACLAnalyser: a Tool for Debugging Multi-agent Systems

Juan A. Botía and Alberto López-Acosta and Fernando G. Skarmeta
Universidad de Murcia

Abstract. Multi-agent systems are a special kind of distributed systems in which the main entities are autonomous, in a proactive sense. These systems are special because their unpredictability. Agents can spontaneously engage in complex interactions, guided by their own goals and intentions. When developing such kind of system, there are many problems the designer/programmer has to face. All these problems make virtually impossible to totally debug an enough complex multi-agent system. In this article we describe a debugging tool we have developed in our lab which pretends to alleviate the problems derived from distribution and unpredictability.

1 Introduction

To follow the global behavior of information systems which have control, data and processes all organized in a distributed fashion is a hard task [12, 3, 10]. Each entity of the system uses to have a local view or the organization and to integrate all limited views coming from each entity is a work which depends on the user (i.e. the programmer). Moreover, due to the complexity of interactions between many entities, a really effective visualization is even more important than in monolithic systems. The main reason for the high complexity on debugging these kind of systems is that the source of errors is two fold. In the one hand, each independent entity (i.e. agents) can be a source of functional errors. This kind of errors are derived from a bad design of the methods used to carry out an internal task. These errors are also typical in monolithic systems. In the other hand, in distributed systems there is also another typical source of errors, those derived from coordination and cooperation tasks. These kind of errors are even more subtle to discover, diagnose and solve. Moreover, multiagent systems are autonomous and their interactions can even be more complex and unpredictable than the typical processes we can find in *conventional* distributed systems.

In this paper we propose a tool devoted to assist on debugging both functional and coordination/cooperation related errors. From our perspective, the following are the requirements such a tool must accomplish with:

1. For each entity (i.e. agent) of the system, the tool must allow us to revise its behavior during the whole execution of a trial. It must keep a register of the actions (i.e. in the sense of communicative acts [1] in this case) along with the time they were executed and the appreciable effects they caused.
2. An important requirement is that the register needs a global time reference. In order to analyse an isolated interaction between two or more agents, it would be enough to keep a local time reference, just to order into a correct sequence all the exchanged messages. However, when the total systems has to be analysed, to obtain a global time stamp is crucial. In the case of having that global time reference, it is possible then to check if some actions had effects over others inside a global context.
3. The analysis tool must be able of not only registering but distinguishing all the different interactions carried out in the trial, also registering for each interaction, the protocol used, the agents implied, the time sequence of interactions and results for all agents. In order to correctly detect some nondesirable behaviors in agents implied at the interaction, the tool must be able of deciding if a sequence of messages, supposed to belong to a concrete interaction protocol, really adjust to it.
4. When hundreds or even thousands of agents are involved in a trial, other problems arise like, for example, how can we move through really populated groups of agents without losing the global perspective. Such kind of tool should also allow to perform some kind of zoom in/out operation, in a visual and comfortable manner for the debugger.
5. Also the global perspective of the trial would not be completed is some statistically descriptive measures where not used. The tool has to show statistics for each kind of interaction protocol, number of successful/unsuccessful conversation, statistics by kind of performative, and also graphs showing the number of messages sent through time.

Requirements 1 and 2 are addressed in section 2.1. Requirement 3 is addressed in section 2.2. The accomplishment of requirement 4 is explained in section 2.3 and some of the measures mentioned in requirement 5 are shown in section 3.

The rest of the article is structured as follows: section 2 will introduce the tool from an architectural and functional perspective. We will demonstrate the usefulness of the tool in section 3 with a working example. Finally, section 4 will summarize lessons learned and mention some future works.

2 The ACLAnalyser tool

The ACLAnalyser tool has been developed to analyse runs on the JADE (Java Agents Development Environment) platform. Hence, it is useful to debug FIPA compliant MAS [2]. It is coded in Java and available on request from the authors.

2.1 The architecture

The architecture of the ACLAnalyser appears depicted at figure 1. In the figure, there are four main elements. These

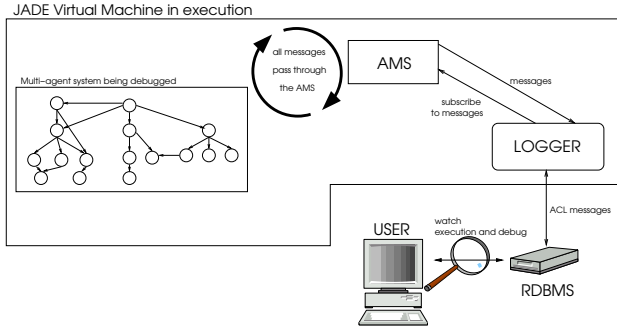


Figure 1. Architecture for the ACLAnalyser tool

are the JADE execution environment, a logger, a relational database and the user module.

The JADE environment represents the multi-agent system being debugged. It has to be executed and, in turn, it will generate the messages exchanged by all the agents of the system.

The logger is actually another JADE agent (i.e. a spy agent), which is not part of the MAS being debugged. In order to start using it, the JADE platform must be launched, along with the spy agent. This agent is really like the Sniffer which comes with the JADE distribution. Its purpose is to intercept all the messages exchanged between agents. However, instead of simply reproducing it like the Sniffer does, this spy agent has two main tasks: (1) to keep track of all the interactions being performed in the run currently executing, and (2) to store all messages in the third element of the architecture: a relational database. Communication between the spy agent and the database is done through a JDBC connection.

In figure 2 it appears the entity-relationship model used for the relational tables used in the database. The main element

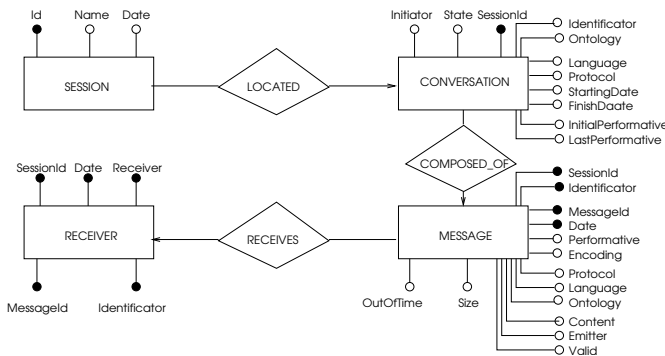


Figure 2. Entity-relationship model for the log of messages

of the database is a session. A session represents the set of

all messages generated in a run of the MAS being debugged. Obviously, a session can be composed of many different conversations. And a conversation corresponds to a concrete interaction between an initiator and one or more agents, but always following a concrete interaction protocol which can be a FIPA standard or user-defined. The use of a relational database to store all interactions provide with many benefits. One of them is that the system should be able of storing a virtually unlimited number of messages corresponding to complex interactions for the case of runs where thousands of agents are implied. Connection to the database from the Spy agent is done through a remote JDBC connection. This implies that the debugger has a minimal impact in terms of resources consumption as, for each message, the only thing to do is send it to the database which, in turn, can be located in a different machine than the one running the system being debugged. Another important benefit of using a relational database through a JDBC connection is that all the software, exception done for the Spy agent, can be reused for another FIPA compliant platform rather than JADE like, for example, FIPA-OS or ZEUS. The only additional work for integration for other platforms is to habilitate the mechanism in charge of sending all messages to the Spy agent. Moreover, the tool also offers the possibility of showing the effectivity of the debug process, because the database stores results for all the sessions performed during the whole process.

Finally we have the user module. This is in charge of presenting results to the user in order to help him keeping track of all events produced in the execution. This module is the most important part of the tool because it shows the user all the information needed to check if something went wrong in the execution of the MAS and, in this case, to make an appropriate diagnostic.

2.2 Interaction protocols

We have already mentioned that, in a run of the MAS being debugged, all the messages are organized into conversations, and all conversations belong to a session. A session is the compendium of all the messages exchanged in a run.

As the user module must represent any conversation produced in a run, the tool needs a way of representing interaction protocols and a method to match a sequence of messages with a concrete interaction protocol. Examples of FIPA interaction protocols are `fipa-request` [6], `fipa-query` [5] and the well known `fipa-contract-net` [11, 4].

In ACLAnalyser, an interaction protocol has an internal representation as a finite state automata. When a new conversation is initiated between agents, an automata is created for all the participants as they become known for the tool. In this way, the state of the automata represents the state of the conversation for the corresponding agent. When a new message is received by the spy agent, the roles for the sender and receiver of the message are obtained. Having these roles, state of each participant and the performative of the message, transitions are done for all automata.

Each instant of the run, all participants are in a concrete state. There are four kind of states. The first one is the initial state, let it be denoted with $q_{0,i}$, for the i -th participant. The second one is the final state, let it be denoted with $q_{f,i}$ for the i -th participant. If its automata is in this state, the conversa-

tion has ended for the participant i (i.e. if more messages are received by it, corresponding to the same interaction, these messages are incorrect). The third kind of state is the error state, let it be denoted with $q_{e,i}$ for the participant i . This state represents a situation in which the agent i either has passed a timeout before which it should have sent a message or received it. The fourth state is the so called *possible final*, let it be denoted with $q_{p,i}$ for the agent i . If an agent is in this state, it can be considered as a final one but, however, more messages of a concrete kind can still be received (sent) and the state would not be erroneous.

2.3 Zooming out

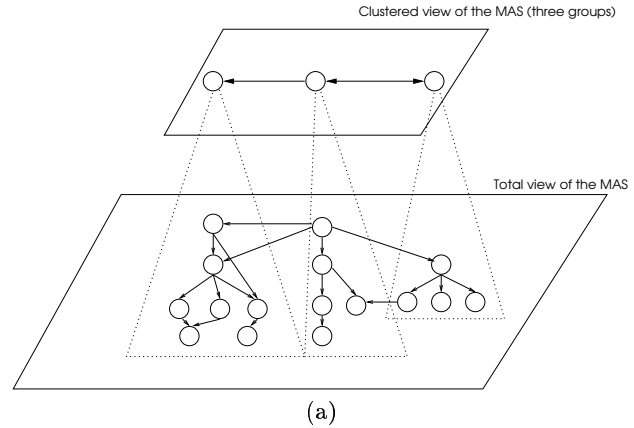
The ACLAnalyser tool can show a view consisting on a directed graph of all the agents implied in a run (i.e. a concrete session) or a graph of all the agents implied in a simple interaction (i.e. a conversation). In this graphical view, both nodes of the arc are agents and if, in the case that there is an arc from the agent i to the agent j , that means that agent i have sent one or more messages to agent j , and also that messages have been received by j . Also, each arc is labeled with the number of messages sent and the total number of bytes transferred. This basic view of the whole system has the following applications: detection of no communication, when expected, between two or more agents, detection of excessive number of bytes exchanged between two or more agents and detection of unbalanced execution configurations in which agents from a concrete group (or machine) show an amount of activity not compensated with the rest.

An interesting point of this kind of view of the system is that it is still useful when hundreds or even thousands of agents are implied, because it is possible to make a zoom out. It is possible to apply a clustering process on the messages to detect groupings of agents and present them by means of that.

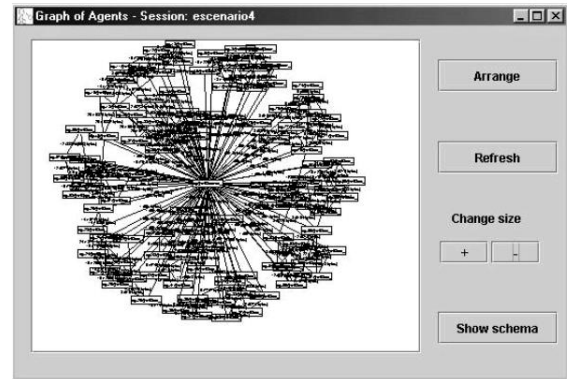
The clustering is based on the ROCK [7] clustering algorithm. Conventional clustering algorithms detect spontaneous groupings on data [8]. This grouping is based on a distance measure, typically the euclidean distance. This basic clustering works on continuous data, however we have categorical data (i.e. messages exchanged between agents in a MAS). That is the reason which moved us to use ROCK. This clustering algorithm works with boolean and categorical (i.e. symbols) data. Instead of using a distance, ROCK uses the concept of link. This term is, in turn, based also in the concept of neighbor. Two items of data are considered as neighbors if they share some degree of similarity, depending on the problem. The number of neighbors between two entities corresponds to their number of links. Once the links are calculated, groupings are determined between entities which share a high number of links. The number of groups detected is a configuration parameter of the algorithm. Hence, the lower the number, the higher the abstraction level we are using to look at the agents society.

With respect to this concrete problem, two agents will be considered as neighbors if they exchanged some messages. In this way, the zooming process is represented in the graph (a) at figure 3. There we can see, in the lower layer, the whole agents society and the links between the agents. We can easily check that there are three different groups of agents in terms

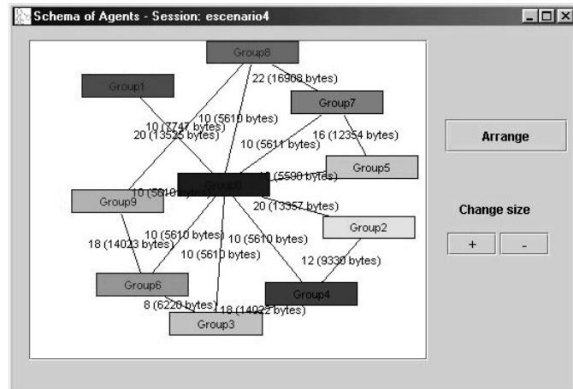
of neighborhood. Should we apply the ROCK clustering algorithm, with the number of groups set to 3, then we obtain the three clusters of the higher level. As a particular example of



(a)



(b)



(c)

Figure 3. Different levels of abstraction for a MAS through categorical clustering

using the tool, in graph (b) of the figure 3, it appears represented a graph corresponding to a hundred agents. Applying the zoom out, we can obtain the graph (c) of the same figure with ten groups, corresponding to the example in the next section.

3 A concrete example of use

We will show now the use of the tool with a concrete example. For that, all the agents will execute the task of deciding which agent will be the leader for the rest when this process does not need to be fair. This is a well known problem in distributed systems when spontaneous centralised coordination is needed between a group of entities. We will use the algorithm proposed in [9], pag. 101. The following is its pseudocode:

```

M = I;
send I to all neighbors
on reception of message J do
  if M < J then
    M = J;
    send J to all neighbors;
  end
on message from each neighbor received do
  if M = I then return leader else return follower

```

It assumes that all agents have a numeric ID and the selection process ends when the agent with the highest ID assumes the leadership (i.e. remember that it is not necessary for the process to be fair in any sense).

There will be a *launcher* agent, in charge of creating the rest of the agents which will compound a group of one hundred. Each agent will have a unique ID and an unique random number to be used for the leadership selection, both generated by the launcher agent.

For a first scenario, the agent i will know the agent $i - 1$ and the agent $i + 1$. Now, we define a new interaction protocol, called `protocolo-lider` using the window appearing at figure 4. In this figure, two different cells (i.e. rows in the

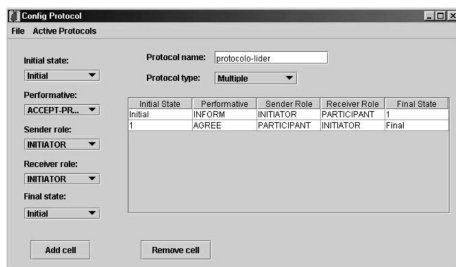


Figure 4. Definition of the `protocolo-lider` interaction protocol

table) appear. Each cell refers to a kind of message. The first column indicates the state of the automata from which the transition is generated when the message is sent. The second one, the kind of performative. The third and the fourth one who sends and who receives messages, respectively. Finally, the fifth column shows to where state the transition generated by sending the message goes. The protocol type is `multicast` because the initiator sends a message in multicast mode (i.e. to all its neighbors) and has to wait until all agree messages are received from the participants. The buttons on the left are used to fill the gaps in new cells. But still another interaction protocol is needed. This will be used for all the agents when it is clear which one is the leader and will be called `protocolo-resultado`. It will consist in an `inform` sent to the launcher. When all these messages are received, the final result will be shown to the user by the launcher.

We will simulate two different errors. The first one is a coordination error, and it will consist in killing the agent with 20 ID after some seconds alive. After running the simulation, the launcher will show an error window because it did not receive all the necessary `inform` messages to compound the final result after a timeout. Once the error is produced, we can use the general statistics window produced (it appears at figure 5). Two different data are marked with a circle. The

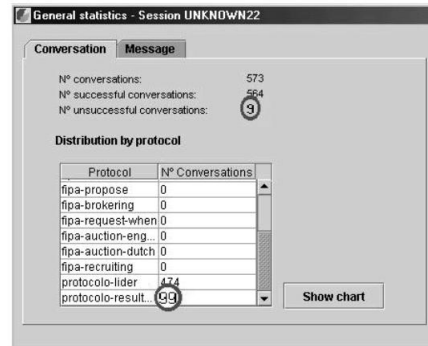


Figure 5. General statistics window for the coordination error scenario

one in the lower part of the window indicates that a total of 99 conversations following the `protocolo-resultado` protocol were performed when the value should have been 100. Also, in the upper part of the window we can see that 9 conversations which did not totally follow their corresponding interaction protocol. How can we determine which agent did not send the `inform` message to the launcher? In an scenario with a few agents we could use the MAS communication graph (see figure 3, graph (b)). But this is not the case here. Fortunately, there is an alternate manner. We can search for the unsuccessful conversations of session UNKNOWN22 by using the conversations search window of the tool and obtain the list of conversations appearing at the window of figure 6. Having a glimpse at the

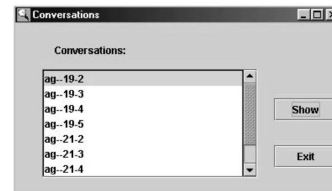


Figure 6. List of uncorrect conversations for the session UNKNOWN22 used in the coordination error scenario

window, we can see that all unsuccessful conversations were performed by agents 19 and 21. We can get more in detail in, for example, the first conversation labeled with `ag-19-2` and obtain the window on the left of figure 7. In which it can be seen that the first and last performative is a single `inform`. From which we can conclude that the agent 20 did not respond with the required `agree`. We can also confirm that by the sequence diagram we can obtain by clicking at

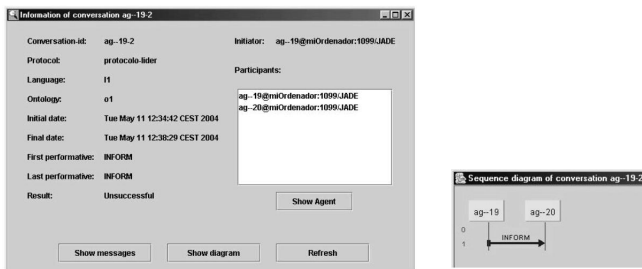


Figure 7. A detailed view of conversation `ag-19-2`, on the left, and the sequence diagram on the right

the `Show Diagram` button. It also appears at figure 7 on the right. We can also have a look at the rest of the unsuccessful conversations and conclude the same.

We can get more details of the error by using the view of the 20 agent, appearing at figure 8. We can check there that this

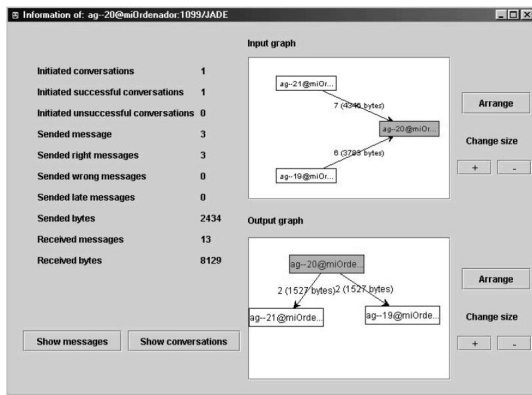
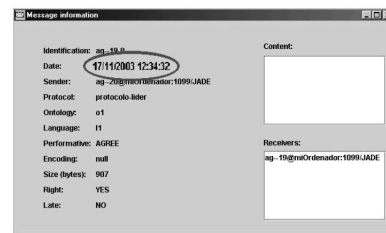


Figure 8. Detailed view of the activity of `ag-20`

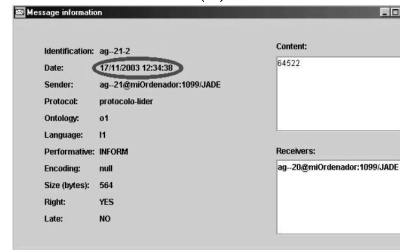
agent did not send any message to the launcher, by having a look at window `Output graph`. Moreover, we can check by using the `Show messages` button, when this agent sent its last message (with label (a) at figure 9) and when it receive the first unanswered one (with label (b) at figure 9). From this we can conclude that `ag-20` died some time between 12:34:32 and 12:34:38.

4 Conclusions and future works

In this paper, some of the many functionalities of the ACLAnalyser has been shown. Its strengths are the use of a relational database which allows storage of any number of complex multi-agent interactions, views of the MAS at different abstractions levels by using categorical clustering, and a complete and versatile window system which allows accessing any information available for a run. This, as a whole, compound a powerful debugging system for complex multiagent systems. Future works include to investigate the application of OLAP (On-line Analytical Processing) techniques for visu-



(a)



(b)

Figure 9. Last message sent by `ag-20` with label (a) and first unanswered message with label (b).

alizing the database and a more coupled integration with the execution system (i.e. JADE platform).

REFERENCES

- [1] J.L. Austin. *How to Do Things with Words*. Oxford University Press, 1962.
- [2] J Dale and E. Mamdani. Open standards for interoperating agent-based systems. *Software Focus*, 2(1), 2001.
- [3] David W. Flater. Debugging agent interactions: a case study. In *Selected Areas in Cryptography*, pages 107–114, 2001.
- [4] Foundation for Intelligent Physical Agents. FIPA contract net interaction protocol specification. Technical report, FIPA, August 2001.
- [5] Foundation for Intelligent Physical Agents. FIPA query interaction protocol specification. Technical report, FIPA, August 2001.
- [6] Foundation for Intelligent Physical Agents. FIPA request interaction protocol specification. Technical report, FIPA, August 2001.
- [7] Sudipto Guha, Rajeev Rastogi, and Kyuseok Shim. ROCK: A robust clustering algorithm for categorical attributes. *Information Systems*, 25(5):345–366, 2000, (cite-seer.nj.nec.com/guha00rock.html).
- [8] Tom M. Mitchell. *Machine Learning*. McGraw-Hill, 1997.
- [9] Jorg P. Muller. *The Design of Intelligent Agents. A Layered Approach*, volume 1117 of *Lecture Notes in Artificial Intelligence*. Springer, 1996.
- [10] David Poutakidis, Lin Padgham, and Michael Winikoff. Debugging multi-agent systems using design artifacts: The case of interaction protocols. In *AAMAS'02*, Bologna, Italy, July 2002.
- [11] Reid R. Smith. The contract net protocol: High-level communication and control in a distributed problem solver. In A. H. Bond and L. Gasser, editors, *Readings in Distributed Artificial Intelligence*, pages 357–366. Morgan Kaufmann Publishers, Los Altos, CA, 1988.
- [12] Marc H. Van Liederke and Nicholas M. Avouris. Debugging multi-agent systems. *Information and Software Technology*, 37(2):103–112, 1995.