On the application of clustering techniques to support debugging large-scale multi-agent systems

Juan A. Botía, Juan M. Hernansáez and Antonio F. Gómez-Skarmeta

Departamento de Ingeniería de la Información y las Comunicaciones
Universidad de Murcia

Abstract. This work analyses the problematic of debugging a multi-agent system. It starts from the fact that MAS are a particular type of distributed systems in which the active entities are autonomous in the sense that behavior and knowledge of the whole system is distributed among agents. It situates the problem by firstly studying the classical approaches for conventional code debugging and also the techniques used in distributed systems in general. From this initial perspective, it tries to situate agent and multi-agent systems debugging. It finally proposes the use of conventional data mining tasks like clustering to, by summarising, help in debugging huge MAS.

1 Introduction

Nowadays, there is almost a total lack of tools to assist in the task of debugging and monitoring agent based distributed information systems in where the typical scenario of execution involves hundreds of thousands agents. In such cases, strong and flexible tools are needed to log and recover all necessary data and to analyze it by giving enoughly abstract views. These views should maintain the appropriate abstraction level because, in scenarios involving such a high number of agents, there is a clear necessity of summarizing to gain sight into what is really happening in the system. This paper proposes the use of data mining over agent messages to support this difficult task. Techniques exposed in this paper are implemented in the ACLAnalyser. This tool is described elsewhere [4] and you can find it at the JADE agents platform web page, in the form of an add-on.

The rest of the article is organized as follows. Section 2 introduces the general problem of debugging pieces of software and delimitates the particular issue of debugging agents. Section 3 is devoted to define the general framework we propose here, i.e. to use data mining on agent communication language messages to assist the developer in debugging a MAS (Multi-Agent System). Finally, section 4 outlines initial conclusions and points out future research.

2 Debugging software artefacts

Debugging and testing software artefacts is not easy task [22]. Moreover, programming errors may lead to get an information system down virtually all the

time, make services offered by a software company unavailable, make not dessired changes to valuable information and, in the worst case, produce wrong outputs.

In debugging software programs we find two different approaches. The first one consists in explicitly modifying the program being debugged to include checks about, for example, the values of critical variables. After that, the analysis process is done without having to execute the code, i.e. statically. This is the approach we may find, for example, in what is called *model checking* [5]. It consists in, given a code to check, to use a graph which represents the different states in which the code being checked may be found. After this graph is built, we use some search algorithm to explore all the possible states trying to find error states. For an example of a real system which systematically checks code on C and C++, please see [16]. A related approach is *program analysis*. It consists in analyzing the code, without having to execute it in order to detect, for example, deadlocks and data races. It detects problematic code regions in the source code, analyzing them and giving an accurate diagnosis about possible errors that may occur during runtime. Please, see [7] for an example of such debugging programs. The second approach consists in dynamically monitoring the program. This is what is called *dynamic checking* [24]. With a dynamic checker, the target code is modified in some way that it checks itself about invariants, and reports on possible violations of the invariants are used to follow the behavior and perform some diagnosis when necessary.

Considering this ideas, the following questions arise. Do model and dynamic checking have any applicability in the context of MAS programming? To what extent is conventional code, debugged by the above mentioned tools, related to that of a typical MAS? Starting from the considerations made in the last paragraph, we may deduce from the last paragraph that model and dynamic checking, may be applied to debug the source code of single agents, provided that they are coded in a language, let it be denoted with $\mathcal{L}$, and we have a checker for $\mathcal{L}$. In this case, we may detect data races and deadlocks in the internals of an agent. At the end of the day, this would be conventional debugging, i.e. it would be like debugging any other program written in $\mathcal{L}$. Notice that this simple analysis has been done on the basis that $\mathcal{L}$ is a general purpose language and not agent oriented like APRIL and JACK may be. An example of a general purpose language used to code agents is Java, as it is used in agent programming environments like JADE, for example. In this case, any existing debugger may be used to analyze the internals of any single agent. In the case that an agent oriented programming language is used, specific debugging techniques either pertaining to model or dynamic checking should be developed first. They should take into account typical agent programming elements like believes, goals, tasks, roles and so on. One good example of such approach is the Tracer tool [12]. It uses reverse engineering and a particular model ckecking to generate *relational graphs* which relate beliefs, intentions and actions. They are also used to generate explanations on actions.

But, would it be possible to apply model and dynamic checking at the inter-agent level? What are the particularities of MAS programming which makes it

different from any other concurrent programming discipline? For the rest of the article, we will limit the answer to that questions with the consideration of an specific, standard and widely used type of agents, FIPA agents. In principle, FIPA agents may be coded in any programming language. For example, FIPA agents from JADE and ZEUS platforms are coded by using Java and agents running at the APRIL Agent Platform are coded using the April agent programming language. This three agent platforms are FIPA (i.e. any agent of the three platforms interoperate with agents in the other two platforms). The only thing that makes all platforms similar is the ACL (Agent Communication Language) [9] they use. FIPA agents talk to each other using a predefined and standard set of communicative acts [6]. In consequence, general FIPA agents systems debugging has to be considered at this level only.

There are two specific reasons for restricting the discussion to FIPA agents. Firstly, we pretend to define a general approach for MAS debugging. We believe that to be general is to be useful for more people. However, there is an important number of different agent theories, architectures and languages. This is something that makes unaffordable a theory of general debugging with some guarantee of success. But, the approach is yet general if we take as a reference, the most widely used framework: FIPA standards are the most widely accepted references to develop agents and multi-agent systems. In this sense, if we focus our attention to FIPA agents, we keep the approach general in the sense that FIPA is the most widely used framework to develop software agents. Secondly, the FIPA ACL may be considered as a very stable and formally defined ACL as all performatives used in the communicative act library come with complete semantics [9].

3 Mining agent messages for debugging

FIPA messages are compound of an envelope and the message content. The envelope is a set of attribute-value pairs with the necessary information for correct delivery and conversation management, as agents structure their communication through conversations following concrete interaction protocols like, for example, the well known Contract Net protocol [20, 10]. When designing such kind of protocols, it is possible for the designer to incur in some errors. This kind of errors and techniques for detecting them find their roots in traditional concurrent processes [3] in which code is written with a number of critical sections and the modeling task is usually done with Petri nets [19]. To give an example of such errors, an agent a_i may fall into a deadlock when it keeps waiting for a resource r to be produced by agent a_j and, at the same time, a_j needs another r' to produce r and it is precissely a_i which should produce r'.

In the context of this paper, we are focusing in debugging systems whose design has been completely defined and the MAS has been coded in a concrete language. These kind of errors we mention in the last paragraph will be produced in the design phase when all the coordination protocols for agents interaction were totally specified. In this phase, it is usual to use either finite state automata

or Petri nets to specify the interaction protocols that agents use to interact [21, 1]. Hence, is a responsibility of the designer to produce a correct design or either detect these errors. The work of Poutakidis et al. goes in this direction [18, 17]. In Prometheus methodology for agent oriented software engineering, interaction diagrams are used to specify interaction protocols and them diagrams are transformed into petri nets which are used to follow dialogues between agents. Another interesting approach for interaction protocols verification consist in using a declarative language to express what is expected from each message, in terms of semantics [2]. This is then used to validate conversations.

So, what kind of systems are the target of ACL mining? This kind of data mining, as we see it, is useful when there is an important number of agents implied in the MAS, conforming an agent society. Moreover, such a MAS compounds a society of hundreds, thousands or even millions agents and an important number of them show a high communication activity (i.e. the number and size of messages exchanged is high during time). The whole picture of debugging, which shows what is the place reserved to ACL messages mining, appears depicted at figure 1.

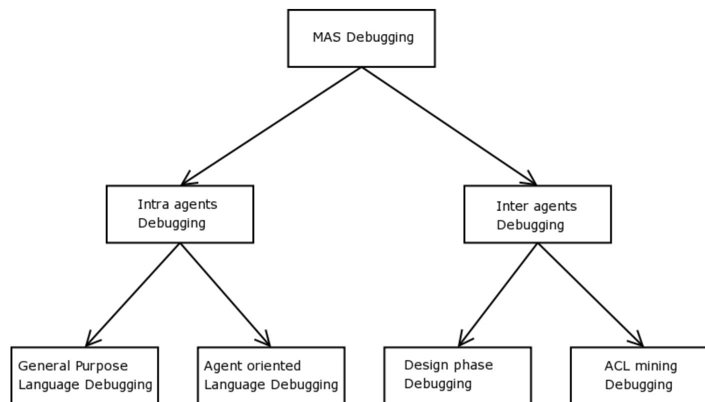

Fig. 1. The different debugging types when considering multi-agent systems

3.1 Source data

This section will formally define ACL messages mining, $ACLM^2$ for short. We will define what may be the data to be mined in this case and how it will be organized.

Let M be the set of all possible messages that can be exchanged between FIPA agents. An element of M, let it be denoted with m, may be defined as $m = (e, c)$ where the $e = \{v_1, v_2, \ldots, v_n\}$ refers to the envelope of the message by means of the variables representing each of its parameters such as v_i refers

to the i-th parameter and contains the value of that parameter in m. Besides, c represents the content of the message. We may also define a session in a run of a MAS, let it be denoted with S_k for the k-th run, as the total set of messages exchanged in that run. Moreover, S_k may be seen as a data set, compound by tuples of the form (e, c). Tuples in S_k contain categorical data (e.g. the sender and receiver of the message) and numerical data (e.g. the timeout to wait for the next message in the conversation). From now on, we can see each S_k as a relation that could be analysed and it would be possible to extract some knowledge from it, i.e. we can apply conventional data mining [8] techniques to study the activity of the multi-agent system being programmed. Hence, we define $ACLM^2$ as the process of applying conventional data mining techniques to data coming from sessions, with the purpose of locating anomalous behaviors in the execution of the MAS being debugged.

3.2 Data visualization

One typical data mining task is *complex data visualization* [23]. In this task, data is analysed to find adequate graphical representations which, at a first sight, are capable of representing information in a manner that may allow to obtain quick answers to questions made on source data. We believe that using data visualization in the multi-agent systems development process is useful. And this will be demonstrated through the rest of section 3.

One of these graphical representations is what is called an *agent communication graph*. It consists in a directed graph in which nodes are agents and node i is connected to node j when agent i has sent one or more messages to agent j and j received them correctly. More information may be added to the graph, for example, decorating each arc with the number of messages sent and the total number of bytes transferred. Some of the applications of this very basic view of the whole system are:

- detection of no communication, when expected, between a number of agents,
- detection of excessive amount of bytes exchanged between two or more agents and
- detection of unbalanced execution configurations in which agents from a specific group (or machine) show an amount of activity disproportionate to the rest.

We may find this kind of application view in systems like Zeus agent platform tools. This kind of information representation tools are very convenient when the number of agents in the system being debugged is small (i.e. less than one hundred agents, for example). However, they become useless when this number grows. In this scenarios, when multi-agent systems are really huge, we need more abstract representations. We will now illustrate the situation with a concrete example.

3.3 An example

This example will consist in using a coordination protocol to distributely decide which agent, among a group of one thousand, will be the leader. This algorithm can be found in [15], pag. 101 and the following is a possible pseudo-code:

```
maxId = ID;
send ID to all acquaintances
on reception of message J do
  if maxId < J then
    maxId = J;
    send J to all acquaintances;
  end
on message from each acquaintance received do
  if maxId = ID then return leader else return follower
```

where ID is an unique identifier which all agents have and which has been randomly set. The leader selection process ends when the agent with the highest ID assumes the leadership, and when this occurs, all agents know which ID is the highest one. An agent knowing that its ID is lower that at least one of the received IDs deduces that it is not the leader. If all IDs received by an agent are lower that its ID, it becomes the leader.

Let n be the number of total agents and m an integer, such as $m << n$. The m represents the number of different disjunct sets in which the n agents are organized. Lets suposse, for simplicity, that the number of agents in each subset is n/m. For each agent set, there will be a group coordinator. This agent has the rest of agents in the same group as acquaintances and these other agents have the coordinator as their unique acquaintance. With this arrangement, the agent communication graph will be a tree, in which the root node is the launcher (i.e. the agent which starts all the others and inform them to start executing) and its direct children are the group coordinators, each one having as children their respective acquaintances.

If we run this example with $m = 500$ and $n = 5$, we may obtain a communication graph like the one appearing in figure 2. This graph was obtained with the ACLAnalyser tool [4]. Quickly, we can conclude that this representation is useless because, in this case *too much information is not information*. However, still a similar graph may be useful in the form of a more abstract representation of the same scenario.

3.4 Clustering agents for more abstraction

In this section, we will explain our approach to summarize complex communication graphs, produced in situations where huge MAS runs are represented. The key here is to find which are the most convenient graphical representations or views. Two useful views are the following:

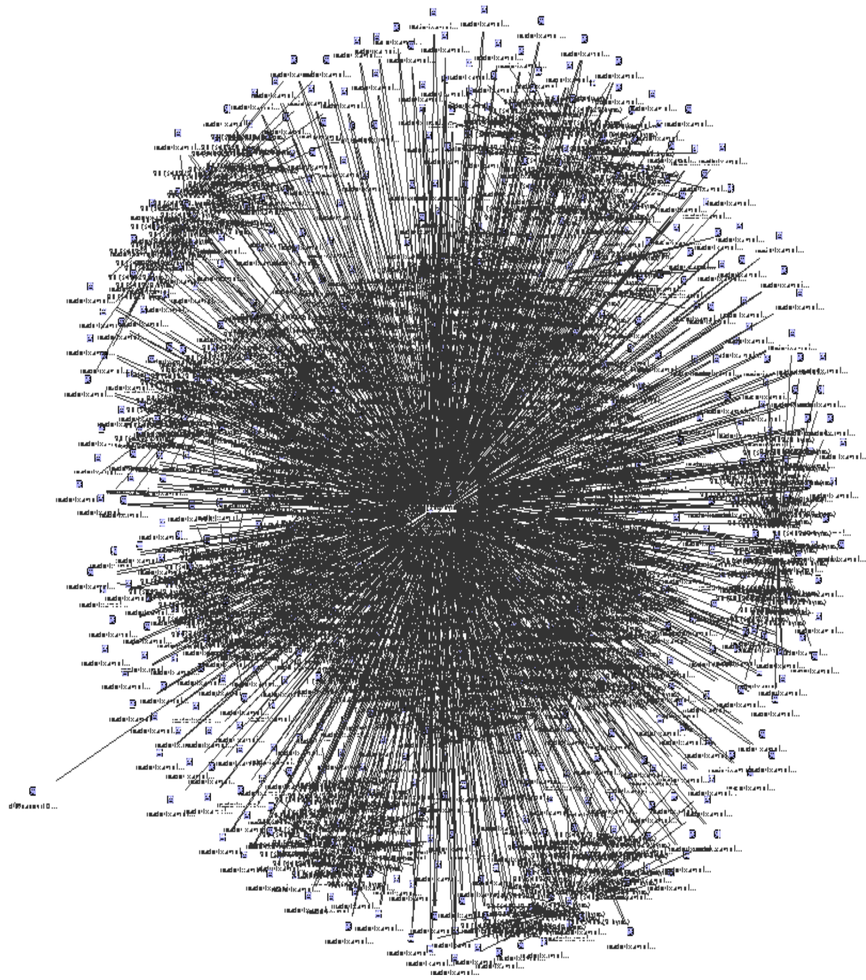

Fig. 2. A *communication graph* compound by all agents in the example

- similar communication activity view: a view in which agents are grouped in the same cluster if they communicate with the same agents. With this view we can group agents showing the same external behavior. Hence, they are similar.
- cooperation activity view: a view in which agents are grouped in the same cluster if they maintain a high communication activity between them. With this other view agents are arranged together because they cooperate.

Other useful views would include, for example, the *organizational view* in which, acquaintances are shown related in they maintain some kind of subordinate relation however, this kind of view is out of the scope of the paper.

Obtaining a similar communication activity view The rest of the section is devoted to explain how to obtain a similar communication activity view. We may perform a clustering over all messages exchanged in such a way that agents were grouped into clusters. Then, a group of k agents belonging to the same cluster would mean that these n agents have been maintaining a similar communication activity (i.e. they have been communicating with the same agents and consequently, they appear grouped).

The particular clustering process we have used here to illustrate the effectivity of $ACLM^2$ is based on the ROCK [11] clustering algorithm. Conventional clustering algorithms detect a priori unknown groupings on data [14]. 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). This fact brought us to apply ROCK clustering algorithm. 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 data points are considered as neighbors if they share some degree of similarity above a certain threshold. This similarity definition depends on each problem. Once neighbors are calculated, two data points have a link between them if they share a common neighbor. After links have been calculated, groupings are compound by data points highly linked (i.e. they all share a high number of neighbors). The number of groups created 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. What would be the definition of neighbor in this case? In this application of ROCK, two agents are neighbors if they share some degree of similarity. Hence, they are neighbors if they share the same links, which means that they separately communicate with the same agents.

The only thing that rests now is to prepare the session to be mined, S_k, by means applying a data transformation in order to generate an appropriate data set for ROCK clustering. This transformation may be defined as the following set

$$T_{rock}(S_k) = \{(a_i, a_j)|\ a_i \text{ sent some } m \in S_k \text{ to } a_j\},$$

where a_i and a_j are agents which participated in S_k. Then we would obtain a S_k^{rock} from $T(S_k)$.

If we perform the ROCK clustering on S_k^{rock}, we will obtain a graph like that appearing in figure 3. This functionality appears also in the ACLAnalyser.

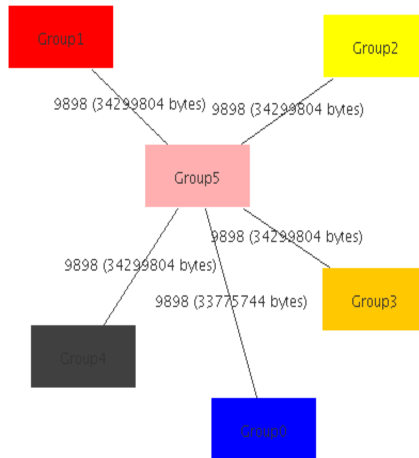

Fig. 3. A *similar communication activity view* in which all agents are grouped into six different clusters.

Notice that this similar communication activity view represented there is much more informative that the graph of figure 2. The first thing to notice is that the graph has a star topology and that there are six different clusters. Notice that each arc comes with a number showing how many messages are exchanged between agents in the two connected clusters and the size in bytes. If we look inside the cluster labeled with `Group5` (ACLAnalyser allows to click on each cluster to show a list of each agent belonging to it) , we find inside only the leaders of each one of the five groups and also de launcher agent (i.e. the one that executes all agents and then waits for responses on who is the leader from each group coordinator). The rest of the clusters have subordinate agents inside.

Obtaining a cooperation activity view The cooperation activity view shows how agents in the same cluster maintain a high communication level. In whose case, it is possible for a developer to discover social arrangements of agents showing cooperation clouds.

The approach to obtain such a view is similar to the one explained above but the clustering algorithm is not the same. Here, we may use a k-means clustering algorithm [13] which arranges data points into clusters but it locates a centroid in each cluster in such a way that this centroid is the point to what the distance from the rest of the points of the clusters is minium, on average. This clustering algorithm is the most well known grouping technique.

In order to correctly apply the algorithm, we need to find the appropriate data transformation. Given that S_k is the sesion we will mine, we transform it into a S_i^{km} which represents the mentioned cooperation activity view. The idea is that two agents are more near (in the sense of the distance used between data points at clustering) if they exchange more messages. Then, two tuples of S_i^{km} should represent two different agents and no more than one tuple should represent a single agent. Let us suposse that in the S_k run, we have m agents, $\{a_1, a_2, \ldots, a_m\}$. Then, in this case, the transformation of S_k should be defined as

$$T_{km}(S_k) = \{(b_{1,i}, \ldots, b_{m,i}) | 1 \leq i \leq m \text{ and } b_{j,i} \equiv \text{ number of bytes sent from } a_j \text{ to } a_i\}.$$

The kmeans clustering needs a distance metric so we may also define the *communication activity distance*, let it be denoted with $d_{i,j}$ between agents i and j, as

$$d_{i,j} = \frac{1}{1 + t_{a_i}(a_j) + t_{a_j}(a_i)},$$

where $t_{a_i} = (b_{1,i}, \ldots, b_{m,i}) \in T_{km}(S_k)$ and $t_{a_i}(a_j)$ refers to the $j - th$ value of tuple t_{a_i}. Now, it is possible to apply the conventional k-means clustering algorithm.

4 Conclusions and Future Work

In this article we have shown how data mining can be applied to debug highly populated MAS. We have identified the different kinds of debug tasks which may be performed over a MAS, depending on the agent level (i.e. inter and intra agent). We have concluded that data mining may be applied on ACL messages to discover, for example, unknown arrangements of acquaintances in very populated agent societies. We have only used a single data mining task which is clustering. We have shown that, depending on the particular transformations applied to the data obtained in a single run, it is possible to obtain different kinds of groupings of agents in the systems which would help in the debugging process.

Ongoing works include exploring other possible transformations to be applied to S_k to obtain new and useful views. We are also exploring the application of other data well known and widely used mining tasks like classification, regression or association rules mining.

Acknowledgements

Supported by the Spanish Ministry of Education and Science through the Research Project TIN-2005-08501-C03-02 and also by the ENCUENTRO (00511/PI/04) research project of the Seneca Foundation with the CARM.

References

1. S. Abdelwahed and W. M. Wonham. Blocking detection in discrete event systems. In *Proceeding of the American Control Conference*, pages 1673–1678, Denver, Colorado, 2003.
2. Marco Alberti, Marco Gavanelli, Evelina Lamma, Paola Mello, and Paolo Torroni. Specification and verification of agent interaction using social integrity constraints. *Electronic Notes in Theoretical Computer Science*, 85(2), 2004.
3. Gregory R. Andrews. *Concurrent Programming. Principles and Practice.* Addison-Wesley, 1991.
4. Juan A. Botía, Juan M. Hernansáez, and Antonio F. Gómez-Skarmeta. Towards an approach for debugging mas through the analysis of acl messages. *Computer Systems Science and Engineering*, 20, July 2005.
5. E.M. Clarke, O. Grumber, and D. Peled. *Model Checking.* MIT Press, 1999.
6. P. R. Cohen and H. J. Levesque. Communicative actions for artificial agents. In *Proceedings of the First International Conference on Multi-Agent Systems (ICMAS-95)*, pages 65–72, San Francisco, CA, June 1995.
7. Dawson Engler and Ken Ashcraft. Racerx: Effective, static detection of race conditions and deadlocks. In *Proceedings of the SOSP*, 2003.
8. Usama Fayyad, Gregory Piatetsky-Shapiro, and Padhraic Smyth. Data Mining and Its Applications: A General Overview. In Jiawei Han Evangelos Simoudis and Usama Fayyad, editors, *The Second International Conference on Knowledge Discovery & Data Mining.* AAAI Press, August 1996.
9. Foundation for Intelligent Physical Agents. FIPA Communicative Act Library Specification. SC00037, 2002.
10. Foundation for Intelligent Physical Agents. FIPA Contract Net Interaction Protocol Specification. SC00030, 2002.
11. Sudipto Guha, Rajeev Rastogi, and Kyuseok Shim. ROCK: A robust clustering algorithm for categorical attributes. *Information Systems*, 25(5):345–366, 2000, (citeseer.nj.nec.com/guha00rock.html).
12. D. N. Lam and K. S. Barber. Comprehending agent software. In *AAMAS '05: Proceedings of the fourth international joint conference on Autonomous agents and multiagent systems*, pages 586–593, New York, NY, USA, 2005. ACM Press.
13. J. B. MacQueen. Some methods for classification and analysis of multivariate observations. In *5-th Berkeley Symposium on Mathematical Statistics and Probability*, Berkeley, 1967. University of California Press.
14. Tom M. Mitchell. *Machine Learning.* McGraw-Hill, 1997.
15. Jorg P. Muller. *The Design of Intelligent Agents. A Layered Approach*, volume 1117 of *Lecture Notes in Artificial Intelligence.* Springer, 1996.
16. Madanlal Musuvathi, David Y.W. Park, Andy Chou, Dawson R. Engler, and David L. Dill. Cmc: A pragmatic approach to model checking real code. In *Proceedings of the OSDI*, 2002.

17. David Poutakidis, Lin Padgham, and Michael Winikoff. An exploration of bugs and debugging in multi-agent systems.
18. 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.
19. Wolfgang Reisig. *Petri Nets, An Introduction*. Springer-Verlag, Berlin, 1985.
20. 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.
21. Agnieszka Wegrzyn, Andrei Karatkevich, and Jacek Bieganowski. Detection of deadlocks and traps in petri nets by means of thelen's prime implicant method. *International Journal of Applied Mathematics and Computer Science*, 14(1):113–121, 2004.
22. James A. Whittaker. What is software testing? and why it is so hard? *IEEE Software*, pages 70–79, January 2000.
23. Graham Wills and Daniel Keim. Data visualization for domain exploration. In *Handbook of Data Mining and Knowledge Discovery*, pages 226–232. Oxford University Press, 2002.
24. Pin Zhou, Fen Qin, Wei Liu, Yuanyuan Zhou, and Josep Torrellas. iwatcher: Simple, general architectural support for software debugging. *IEEE Micro*, pages 50–56, November 2004.