SMADAS: a Cooperative Team for the Multi-Agent Programming Contest using Jason

Maicon Rafael Zatelli, Daniela Maria Uez, José Rodrigo Neri,
Tiago Luiz Schmitz, Jéssica Pauli de Castro Bonson, and Jomi Fred Hübner

Department of Automation and Systems Engineering
Federal University of Santa Catarina
CP 476, 88040-900 Florianópolis - SC - Brasil
{xsplyter,dani.uez,jrf.neri,tiagolschmitz,jpbonson}@gmail.com,
jomi@das.ufsc.br

Abstract. In this paper we describe the SMADAS system used for the Multi-Agent Programming Contest in 2012. This contest offers an useful context to evaluate tools, techniques, and languages for programming MAS. It is also a good opportunity to learn agent programming and test new features we are developing in our projects. Throughout the paper we highlight the main strategies of our team and comment on the advantages and disadvantages of our system as well as some improvements that still could be done. One important result from this experience regards the agent programming language we used, it provides suitable abstractions for the development of complex system and shows an increment in its maturity since no bugs was discovered this year.

1 Introduction

The empirical evaluation of proposals in the context of Multi-Agent Systems (MAS) is a quite complex task and the Multi-Agent Programming Contest [1, 3][1] offers an useful context for doing this evaluation. In particular, the latest Mars scenario has emphasised solutions based on cooperation, coordination, and decentralisation which are important topics for our research. This contest is thus selected as the environment to evaluate the proposals being developed by the authors in their master and Phd thesis. Among the authors, we have one PhD student, three master students, and one undergraduate student. The main approach is (*i*) to develop a *base* MAS for the contest, then (*ii*) the master and PhD students will change the base system using their corresponding proposals, and finally (*iii*) each proposal can be evaluated and compared against the base system. In this paper we report the development and the main features of this base team, called SMADAS (the acronym of our research group). Another objective for attending the contest is to improve the experience in developing MAS. Since most of the authors are just beginning on the domain, the concrete experience is important for their overall learning and maturity in critical analysis.

[1] http://multiagentcontest.org

2 System Analysis and Design

For the analysis of our systems, we adopted a prototype driven approach instead of a well known software engineering methodology because the problem seemed quite simple to solve and we had no experience with them. Thus we decided that it was better to use our time developing the system than learning a methodology.

Based on the agent contest scenario description, we divided the overall problem in sub-problems, each one analysed in detail: exploration, exploitation, attack and defense, buy, repair, and inspection. A team member was engaged with programming each strategy discussed on biweekly meetings. Forty five versions of the system were produced in this phase. These versions were tested and compared with the best teams from the last contest [6, 8, 7, 2] and also against our own versions of the system in order to select the most efficient one. In these preliminary tests, we identified some good strategies for the final implementation. To develop the SMADAS system, we spent about 500 hours, most of them testing the strategies.

The system has 20 agents of five types: repairer, saboteur, explorer, sentinel, and inspector. We considered two main distinct phases: exploration, in which the explorers identify all vertices and nodes in the map and find the best zones, and exploitation, where all agents try to conquest and defend these zones. During the match, if an agent senses a nearby enemy it calls a saboteur to attack it, and also if the agent is damaged it tries to find a repairer to be fixed.

Our agents are able to decide their own actions, however this autonomy produces some conflicting situations like two agents deciding to exploit different zones. These situations are solved using a centralized approach, which consists of a specific agent been responsible for the group decision. For example, one of the explorers defines the zones to exploit and one of the repairers defines the reparation order. Some conflicting situations are simply prevented by using a predefined priority order among the agents, where agents with higher priorities acts before agents with less priority.

The coordination among the agents is based on two communication mechanisms: blackboard and message exchanging. The blackboard is used to provide a global graph view to the agents, since some important information about the graph structure is synchronized in it. We decided to use a blackboard because the agents need an overall view of the scenario to be able to define the system exploitation strategy. The message exchanging is used to share information about the inspected enemies, the ally agent actions and damages, and about the map zones. The communication protocol used when a damaged agent needs to be repaired is shown in Fig. 1. It consists of the agent asking a repairer that contacts the other repairers to find out which one is the closest to the damaged agent. Then the other repairers inform their positions and the closest one is selected to repair the damaged agent. Thus, the selected repairer will send to the damaged agent the meeting path.

The SMADAS system is a truly MAS because the agents are autonomous, reactive, and proactive. They have autonomy to decide how and when to execute most of their actions, except the few conflicting situations explained before.

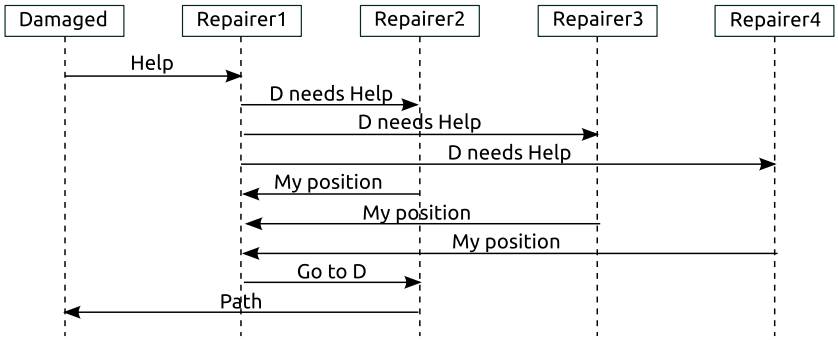


Fig. 1: A communication protocol used to define which repairer will repair a damaged agent. The damaged agent asks the `repairer1` for help, the `repairer1` then contacts the others repairers to find which one is the closest to the damaged agent. All repairer send their position and the `repairer1` elects the closest one. The selected repairer then sends the meet point to the damaged agent.

However, the agents also perform some actions in reaction to environment events, like the start of the step or a received message. Other reactive actions occurs when a saboteur attacks an enemy agent that is in the same vertex or when an agent runs away or defends itself from an enemy saboteur on the same vertex. Furthermore, the agents have a proactive behaviour, that shows up when they try to find a better vertex that improves the team score, contact the repairer when they are damaged, or look for enemies to attack.

3 Strategies

In our strategy both individual and group behavior are important. While the individual behavior is important when the agents are isolated in the map, the group behaviour is responsible for preventing redundant actions and for producing a coherent and cooperative global result. The agents are proactive in order to get achievement points and obtaining a good score. They also use their beliefs and the exchanged information to decide their next action.

As commented in the previous section, we consider two main strategies: exploration and exploitation. In the exploration phase the agents just explore the map and try to get as most achievement points as possible. After step 15, our agents go to a good zone to conquer it.

Since achievement points are important and they accumulate in each one of the 750 steps, it is desirable to obtain them as soon as possible. However, some achievements are more complicated to conquer after some time, hence they can be ignored. For example, it does not make sense to survey all edges in the graph, considering it takes a long time to be performed. Instead of it, our agents stay in a vertex getting more score by exploiting water wells. For the same reason we

are not interested on inspecting all opponent agents, thus our inspectors only inspect them when they are near.

After the exploration phase, the exploitation phase starts. One of our explorers reasons about which are the two best zones in the map to be exploited. Exploiting two zones is advantageous since the map is symmetric and it is particularly important against teams that keep only one zone. In order to do that, we used a modified version of the BFS algorithm, that is run for all vertex, summing their values until some depth. The vertex with the highest sum represents where the best zone is (zone 1). After it, the algorithm tries to find the second best vertex to set the second best zone (zone 2), which may have some intersection with the first one. This algorithm is not optimal because its result is always a circular shape, when the ideal choice often has a free shape.

When the good zones are defined, an explorer organises the agents in two groups, one for zone 1 and another for zone 2. Each group has 10 members, with two agents of each type. The agents are then informed about the central vertex of its zone and how far they can go from it. The central vertex of an area is the one discovered in the exploration phase with the best sum. The distance they can go from it defines the border of the corresponding zone. After it, the agents are positioned in their zones. The non-saboteur agents take positions in vertices that have two neighbour vertices belonging to our team, but without anyone there. The saboteur agents scout their zones and attack opponents inside it, they also attack near enemy zones. We assume that if the enemy zone is not near, the opponent probably has a small zone and we do not need to attack them.

Table 1 shows the strategies and plans for each type of agent. There are plans with more steps (buy, repair, probe) and plans where the agents simply react (attack, parry, inspect, recharge, survey). We noticed that usually long-term plans are not a good idea, because the environment changes quickly. The strategies are explained in more details below.

- Buy: we concluded that it is be better to do not buy many things. We noticed it through tests between our MAS with a buying strategy where the agents buy more things against one where the agents just buy few things, and the second strategy won all matches in all simulations. Firstly the buying strategy consisted of only buying upgrades for the saboteurs: buy sabotage devices to have a strength equal to the highest enemy saboteur health value, and buy shields to have health one time greater than the highest enemy saboteur strength value. We did a second version of this strategy where just one saboteur (`Hulk`) buys upgrades, this had the benefit of decreasing our expenses while also making agent teams with a similar strategy waste money. Another improvement of the buying strategy was the addition of an agent named `Coach`, which received information about our enemies upgrades from the inspectors and used them to notice whether the enemy team is buying or not, if they were not buying anything this agent informs the agent `Hulk` to stop buying upgrades in the matches against this team and then save achievement points.

– Attack: the saboteurs always attack the opponent saboteurs first, and then the repairers. However, in the initial steps, attacking the explorers would be a good second option too, since it would be harder for the opponent team to explore the map. In order to prevent redundant attacks, there is a hierarchy defining which saboteur attacks first.
– Repair: the repair strategy consists of finding the closest available repairer to help a disabled agent, after it the repairer and the damaged agent move close to each other. If there are no available repairers the disabled agent moves to the closest repairer. If there is another closest disabled agent to repair or another repairer, they cancel the process and start it again with the closest agent.
– Parry: if there is an opponent saboteur in the same vertex that our agents, the formula $1/N$ defines the parrying probability, where N is the number of ally agents in the same vertex. This way we can prevent all agents from parrying the same saboteur. Our agents do not parry if there are more or the same number of ally saboteurs and opponent saboteurs, since the opponent probably will attack our saboteurs first. If an agent chooses not to parry, then it leaves the vertex.
– Probe: the explorers always probe the closest unprobed vertex and they repeat it until all vertices are probed. To avoid explorers probing the same vertex, there is a hierarchy which defines the explorers who act first.
– Inspect: the inspectors always inspects near enemies, the aim of inspection is to identify enemy saboteurs and to check if the opponent is using a buying strategy.
– Recharge: the agents always check if they have enough energy before doing an action, if they do not have or it is less than 2 points, then they recharge. They also recharge when they do not have any action to do.
– Survey: the agents only survey if there is an unsurveyed near edge. The sentinels are the main agents responsible for doing survey, but other agents do it too if they do not have anything to do in the step.

Action	Repairer	Saboteur	Explorer	Sentinel	Inspector
buy		x(Hulk)			
attack		x			
repair	x				
parry	x			x	
probe			x		
inspect					x
recharge	x	x	x	x	x
goto	x	x	x	x	x
survey	x	x	x	x	x

Table 1: Implemented strategies by agent type.

Finally, there are strategies to expand the team zone and to stop expanding. The goal of the first one is to conquer more vertices in the same zone: when an agent is participating in a zone occupation and it can go to another vertex without breaking the zone, it will do it. The second strategy stops the agents from expanding when they have a high score and to wait for the opponents reaction.

4 Software Architecture

This section describes the technologies and frameworks that we used to develop our agents and how they are integrated. We used the EISMASSim framework [4] to communicate with the contest server, since the competition is built on Java MASSim platform and Java EISMASSim framework is distributed with the competition files. The programming language used to develop our agents is Jason (version 1.3.8) [5]. Its concept of BDI agents provided useful resources to build our agents, like plans and intentions, which allowed us to implement the strategies and to provide our agents with long-term goals. Another advantage of Jason is its interpreter that allow us to call Java methods, which simplifies the implementation of some algorithms and enables them to run faster. These methods are integrated with our Jason agents using internal actions. More specifically we implemented two algorithms as Java methods: Dijkstra algorithm to find the best path between vertices and Breadth-First Search algorithm to locate the best area in the graph.

A blackboard was used to share and build knowledge about the environment in the form of a graph. The process to update information in the graph has a high computational cost, lasting more than one step. Therefore, to avoid losing steps, the graph is updated and shared every three steps. The agent interaction is divided in two modes: agent-to-environment and agent-to-agent. In the first mode, in each step the EISMASSim framework receives an XML text from the server with the agents percepts, these percepts are then translated into Jason environment perceptions for our agents. This translation however does not happen when our team conquers the full map and the quantity of perception is so huge that the agents are not able to process them on time. In this case, perception is disabled and a default action (e.g. recharge) is sent back to the server. The actions of the agents are translated into text and sent to the server by EISMASSim. The Fig. 2 exemplifies how actions and percepts are exchanged. The agent-to-agent interaction uses Jason speech act based communication.

5 Results

We have tried to develop a system as complete as possible and we created several strategies for each system feature, like exploring, exploiting, buying, repairing, and attacking. Hence we developed many versions of the system, we exhaustively tested each one against the others to select the more efficient. We also tested our system during the contest test phase against the teams provided by the contest

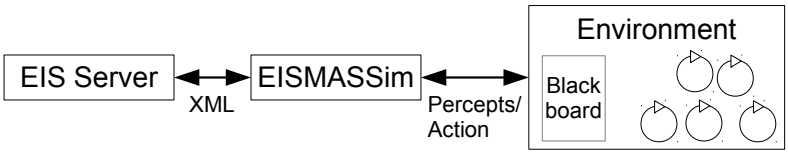
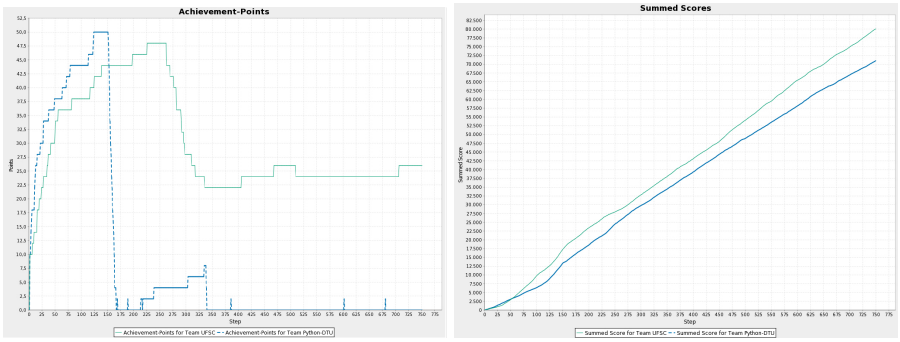


Fig. 2: Communication architecture.

organisation. This approach was our main advantage in the contest and one of the reasons we played eighteen matches against six different opponents and won seventeen. However our system has a worse performance when it confronts a passive system because it is not so offensive. If our agents are in a good map zone they do not bother about the opponent: they assume that the opponent is not in a good area. Also, our agents have no focus on defending a conquered zone and this explains the match we lost against Python-DTU during the contest.

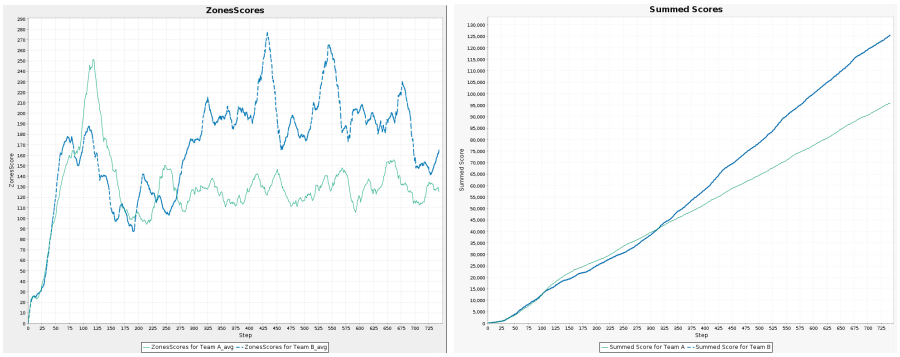
Two main strategies were responsible for the good performance of our system: the buying and exploitation strategies. The buying strategy was decisive because it forced our opponents to reinforce their agents spending a lot of their money. In a match against Python-DTU during the tests phase, for example, we conquered a small area but we won because we had more money. Fig. 3a shows the achievement points from this match. In the step 175 the Python-DTU (in blue) spent most of their money strengthening their agents and SMADAS (in green) spent only a part of its money. In the last 400 steps, from the step 350 to the 750, we had about 23 achievement points in each step, summing 9200 achievement points. In the end, this difference allowed us to win this match, as shown in Fig. 3b.



(a) (b)

Fig. 3: From the step 350, SMADAS-UFSC (in green) has more achievement points than Python-DTU (in blue) (a). This difference has decided the match for SMADAS-UFSC system (b).

Our exploitation strategy chooses two good zones in the map. It was efficient because usually the opponents are concerned about finding and conquering just one good zone. Thus while part of our agents are under attack in one of these zones, the other part are scoring in another zone. This strategy earns less points in each step, because our agents are divided in two smaller zones, but it has better results against an offensive opponent. Fig. 4 shows a comparison from our system performance using these two exploiting strategies. The system in green tries to conquer one single zone and the blue system looks for two zones. The blue system has fewer points at the beginning because it gets two smaller zones. However after some steps where the green system loses many points disputing a single zone, the blue system has one fixed zone scoring without any attack. This strategy was decisive in the match against the AiWYX system.



(a) (b)

Fig. 4: The green system tries to conquer one single zone and the blue system looks for two zones. The blue system finishes the match with a highest score because it keeps scoring in a zone without disputing it with opponents.

6 Conclusion

Participating in the contest was a worthy experience for all the team, we learned a lot about MAS developing and about the tools and languages we used. The contest result, where our team got the first place, is due both to the dedication on developing the strategies described in this papers and to the tools we used. For instance, the Jason programming language supports agent programming with abstract concepts like plans, beliefs, and goals which are suitable for the problem and very expressive. Different from previous participations in the contest where several bugs in Jason were discovered and fixed [9], we did not identify any bug in Jason this year, which shows the maturity of this language. Although we can evaluate the used tools positively in general, some features are still missing.

For example, it was very difficult to change, refactor, and debug the agents code since we have 5504 lines of Jason code and 20 agent instances running concurrently. The tools provided by Jason for debugging, like the sniffer and the mind inspector, are too specific and focused on the details. It is a hard task to identify a bug by looking at thousand of mind samples or message traces. High level abstractions and tools are required to help the debugging of complex MAS.

There is still a room for improvements in our system both in the strategies and the tools. Some of the improvements will be investigated in the authors' master and PhD thesis where proposals will be compared against the version of the system described in this paper. One particular drawback of the system is to be focused only on the agent aspect, all the code is "agent programming". More global aspects should be considered, for instance by organisation and interaction programming as first class abstractions. For that, new models and tools need to be developed.

For the current scenario of the contest, we would propose two improvements. (*i*) Inform opponent's score. It would allow participants to design strategies based on the current match result, rising more confrontations. (*ii*) Leave the graph less connected to increase the use of edges.

References

1. Tristan Behrens, Mehdi Dastani, Jürgen Dix, Michael Köster, and Peter Novák. The multi-agent programming contest from 20052010. *Annals of Mathematics and Artificial Intelligence*, 59:277–311, 2010.
2. Tristan Behrens, Michael Köster, Federico Schlesinger, Jürgen Dix, and Jomi Hübner. The multi-agent programming contest 2011: A résumé. In Louise Dennis, Olivier Boissier, and Rafael Bordini, editors, *Programming Multi-Agent Systems*, volume 7217 of *LNCS*, pages 155–172. Springer, 2012.
3. Tristan Behrens, Michael Köster, Federico Schlesinger, Jrgen Dix, and Jomi Hübner. The multi-agent programming contest 2011: A résumé. In *Programming Multi-Agent Systems*, volume 7217 of *Lecture Notes in Computer Science*, pages 155–172. Springer, 2012.
4. Tristan M. Behrens, Koen V. Hindriks, and Jürgen Dix. Towards an environment interface standard for agent platforms. *Annals of Mathematics and Artificial Intelligence*, 61(4):261–295, April 2011.
5. Rafael H. Bordini, Michael Wooldridge, and Jomi Fred Hübner. *Programming Multi-Agent Systems in AgentSpeak using Jason*. John Wiley & Sons, 2007.
6. Dominic Carr, Sean Russell, Balazs Pete, G. O'Hare, and Rem Collier. Bogtrotters in space. In Louise Dennis, Olivier Boissier, and Rafael Bordini, editors, *Programming Multi-Agent Systems*, volume 7217 of *LNCS*, pages 197–207. Springer, 2012.
7. Marc Dekker, Pieter Hameete, Michiel Hegemans, Sebastiaan Leysen, Joris van den Oever, Jeff Smits, and Koen Hindriks. Hactarv2: An agent team strategy based on implicit coordination. In Louise Dennis, Olivier Boissier, and Rafael Bordini, editors, *Programming Multi-Agent Systems*, volume 7217 of *LNCS*, pages 173–184. Springer, 2012.
8. Mikko Ettienne, Steen Vester, and Jrgen Villadsen. Implementing a multi-agent system in python with an auction-based agreement approach. In Louise Dennis,

Olivier Boissier, and Rafael Bordini, editors, *Programming Multi-Agent Systems*, volume 7217 of *LNCS*, pages 185–196. Springer, 2012.
9. Jomi Fred Hübner and Rafael Heitor Bordini. Using agent- and organisation-oriented programming to develop a team of agents for a competitive game. *Annals of Mathematics and Artificial Intelligence*, 59(3-4):351–372, 2010.

Short Answers

A Introduction

1. What was the motivation to participate in the contest?
   A: Evaluate the result of our master and PhD thesis.
2. What is the (brief) history of the team? (MAS course project, thesis evaluation, ...)
   A: Our team was formed by members from the Multi-Agent Systems research group (called SMADAS) at Federal University of Santa Catarina (UFSC).
3. What is the name of your team?
   A: Our team's name is SMADAS-UFSC.
4. How many developers and designers did you have? At what level of education are your team members?
   A: Our team has six developers and everyone was involved with the system design. We have one PhD, one PhD student, three masters students and one undergraduate student.
5. From which field of research do you come from? Which work is related?
   A: All team members work with Multi-Agent Systems and Artificial Intelligence.

B System Analysis and Design

1. Did you use a Multi-Agent programming languages? Please justify your answer.
   A: We used the Jason language because all members are familiar with it.
2. If some Multi-Agent system methodology such as Prometheus, O-MaSE, or Tropos was used, how did you use it? If you did not, please justify.
   A: We did not use any software engineering methodology because the problem seemed quite simple to solve and we had no experience with such methodologies.
3. Is the solution based on the centralisation of coordination/information on a specific agent? Conversely if you plan a decentralised solution, which strategy do you plan to use?
   A: The system information is decentralised: each agent has all available information about the enemies and the graph. The coordination is centralised in a few cases, to solve some conflicting situations, like defining which agent should be repaired first or what is the best zone to exploit.
4. What is the communication strategy and how complex is it?
   A: The agents use two mechanisms for communication: a blackboard and message exchanging. Some communication protocols are composed by one single message sent by an agents to others (e.g., when an enemy is inspected or when an agent report its action). Other protocols use more messages, for example when a damaged agent request a repair, nine messages are sent among the damaged agent and the repairers.

5. How are the following agent features considered/implemented: *autonomy*, *proactiveness*, *reactiveness*?
   A: The agents are autonomous, reactive, and proactive. They have autonomy to decide how and when to execute their actions, they react to environment events and new messages, and are proactive while looking for a better vertex.
6. Is the team a truly **multi**-agent system or rather a centralised system in disguise?
   A: The tasks of the team are decentralised among the agents which need to coordinate themselves to produce a coherent global behaviour.
7. How much time (man hours) have you invested (approximately) for implementing your team?
   A: We expended about 500 hours developing the system.
8. Did you discuss the design and strategies of you agent team with other developers? To which extend did your test your agents playing with other teams.
   A: We did not discuss the design or strategy with other teams before the contest.

C Software Architecture

1. Which programming language did you use to implement the Multi-Agent system?
   A: The language used for programming our agents is Jason 1.3.8 [5].
2. How have you mapped the designed architecture (both Multi-Agent and individual agent architectures) to programming codes, i.e., how did you implement specific agent-oriented concepts and designed artifacts using the programming language?
   A: The BDI concepts provided by the Jason language are the building blocks to develop our strategies.
3. Which development platforms and tools are used? How much time did you invest in learning those?
   A: We used Eclipse platform with Jason 1.3.8 plug-in. These tools were known by all team members then we spend just few hours learning new features.
4. Which runtime platforms and tools (e.g. Jade, AgentScape, simply Java, ...) are used? How much time did you invest in learning those?
   A: We used EISMASSim framework to communicate with the environment and spent about 50 hours to learn it. For communication among the agents, we used Jason centralised infrastructure.
5. What features were missing in your language choice that would have facilitated your development task?
   A: The Jason language has almost all features we needed to program our agents. However, for some algorithms, we preferred Java because it is faster.
6. Which algorithms are used/implemented?
   A: We used two traditional algorithms for graphs: Dijkstra and breadth-first search.

7. How did you distribute the agents on several machines? And if you did not please justify why.
   A: The agents were conceived to execute in the same machine to simplify blackboard programming, which uses shared memory. Future versions of the system will use distributed blackboards.
8. To which extend is the reasoning of your agents synchronized with the receive-percepts/send-action cycle?
   A: The synchronisation with the environment is given by the reasoning cycle of Jason, where the first step includes the perception and the last the action.
9. What part of the development was most difficult/complex? What kind of problems have you found and how are they solved?
   A: A blackboard has been used to share and build the knowledge about the environment. The process to update information in the graph has a high computational cost, lasting more than one step. Therefore, to avoid losing steps, the graph is updated and shared every three steps.
10. How many lines of code did you write for your software?
    A: We have 7885 lines of code, 5504 written in Jason and 2381 written in Java.

D Strategies, Details and Statistics

1. What is the main strategy of your team?
   A: We conceived our system strategy in two main phases: exploration, in which the explorers identify all vertices and nodes in the map and the best zones, and exploitation, where all agents try to conquest and defend these zones.
2. How does the overall team work together? (coordination, information sharing, ...)
   A: Our agents exchange information to coordinate their activities.
3. How do your agents analyse the topology of the map? And how do they exploit their findings?
   A: Some important information about the graph structure is shared and synchronized in the blackboard and it is used by the agents to move through the map. Despite this, agents do not use any information about topology to make the decisions.
4. How do your agents communicate with the server?
   A: We use the EISMASSim framework to communicate with the server. External actions and usual perception are used by the agents to interact with the EISMASSim.
5. How do you implement the roles of the agents? Which strategies do the different roles implement?
   A: The implemented strategies for each agent type is shown in Table 1.
6. How do you find good zones? How do you estimate the value of zones?
   A: The system uses a modified version of the BFS algorithm to find the best zones in the map. It is run for all vertices, summing their values until some

depth. The vertex with the highest sum represents where the best zone is (zone 1). After it, the algorithm tries to find the second best vertex to set the second best zone (zone 2).

7. How do you conquer zones? How do you defend zones if attacked? Do you attack zones?
   A: With the zones defined, each agent is informed about the central vertex of its zone and how far they can travel inside it. The distance they can travel is the shortest path, in number of edges, between the central vertex and the target vertex. To defend these zones, the saboteurs attack all opponents inside the zone or in nearby vertices. The other agents stay in a vertex that has two neighbour vertices that belongs to our system. It is assumed that if the enemy zone is not near, the opponent likely has a small zone and then our agents do not try to attack it.
8. Can your agents change their behaviour during runtime? If so, what triggers the changes?
   A: If the opponent does not have any buying strategy, the `Hulk` agent changes its behaviour and it stops buying upgrades. Besides it, in the start of the match the saboteurs attack the enemies, but after some steps they change their behaviour to attack the enemies.
9. What algorithm(s) do you use for agent path planning?
   A: We used Dijkstra to path planning.
10. How do you make use of the buying-mechanism?
    A: It was defined the minimum that the agents have to buy in order to make the enemy expend its money. In particular, we have *one* agent (named `Hulk`) that focus on buying and inducing all the opponents to also buy and spend their money.
11. How important are achievements for your overall strategy?
    A: The achievement points are quite important since they accumulate each step. It is desirable to get the maximum of achievement points as soon as possible, but some achievements are hard to get. For example, our system does not surveys all edges and they do not inspect all opponents because it takes a long time and it is better to keep the agents in the best vertices, getting water wells score.
12. Do your agents have an explicit mental state?
    A: The agents have their beliefs and use them to reason about their next action.
13. How do your agents communicate? And what do they communicate?
    A: Our agents communicate indirectly by using the blackboard and directly by message exchanging.
14. How do you organize your agents? Do you use e.g. hierarchies? Is your organization implicit or explicit?
    A: There is an explicit pre-defined hierarchy to prevent redundant actions: agents with higher priority decide before the others.
15. Is most of your agents behavior emergent on an individual or team level?
    A: In our strategy both individual and group behaviour are important. The individual behaviour is important when the agents are isolated in the map

trying to get achievement points. The group behaviour is responsible for preventing redundant actions and conquering zones, for example.

16. If your agents perform some planning, how many steps do they plan ahead?
    A: We do not use planning, all plans are previously programmed based on the strategies.
17. If you have a perceive-think-act cycle, how is it synchronized with the server?
    A: We use the EISMAssim framework [4] to synchronize the agent actions to the server.

E Conclusion

1. What have you learned from the participation in the contest?
   A: We learned a lot about MAS developing and about the tools and languages we used.
2. Which are the strong and weak points of the team?
   A: Our strongest point is that we created several strategies for each system feature and tested them against each other to select the more efficient ones. Our weakness is that our system is not so offensive. Another problem is that our agents does not focus on defending their own zone.
3. How suitable was the chosen programming language, methodology, tools, and algorithms?
   A: The Jason programming language was quite mature and suitable for the agent programming. However, we still need tools for programming and debugging at a higher level of abstraction.
4. What can be improved in the contest for next year?
   A: We can improve our system both in the strategies and the tools. Our system is focused only on the agent aspect and more global aspects should be considered.
5. Why did your team perform as it did? Why did the other teams perform better/worse than you did?
   A: Our system performed well because we focused on extensively testing all strategies.
6. Which other research fields might be interested in the Multi-Agent Programming Contest?
   A: We think that some parts of the problem can be solved by optimisation techniques, which we plan to use in future versions of the systems.
7. How can the current scenario be optimized? How would those optimizations pay off?
   A: We propose two improvements. (*i*) Inform opponent's score. It would allow participants to design strategies based on the current match result, rising more confrontations. (*ii*) Leave the graph less connected to increase the use of edges.