Reimplementing a Multi-Agent System in Python

Jørgen Villadsen*, Andreas Schmidt Jensen, Mikko Berggren Ettienne, Steen Vester, Kenneth Balsiger Andersen, and Andreas Frøsig

Department of Informatics and Mathematical Modelling
Technical University of Denmark
Richard Petersens Plads, Building 321, DK-2800 Kongens Lyngby, Denmark
jv@imm.dtu.dk

Abstract. We provide a brief description of our Python-DTU system, including the overall design, the tools and the algorithms that we used in the Multi-Agent Programming Contest 2012, where the scenario was called Agents on Mars like in 2011. Our solution is an improvement of our Python-DTU system from last year. Our team ended in second place after winning at least one match against every opponent and we only lost to the winner of the tournament. We briefly describe our experiments with the Moise organizational model. Finally we propose a few areas of improvement, both with regards to our system and to the contest.

1 Introduction

This paper documents our work with the Python-DTU team which participated in the Multi-Agent Programming Contest 2012 [7]. We also participated in the contest in 2009 and 2010 as the Jason-DTU team [4,5], where we used the Jason platform [3], but this year we use just the programming language Python as we did in 2011 [6]. See `http://www.imm.dtu.dk/~jv/MAS` for an overview of our activities.

The scenario is based on the scenario from 2011 and has only been changed in a few ways. The most interesting change is the increase in number of agents from 10 to 20 agents per team.

Our focus for the 2012 version of the contest has been on reimplementing the system from 2011. Given that the scenario is very similar to that last year, we decided to look into ways of improving our system. We have been exploring the possibility of implementing an organization for the system using the Moise organizational model [1] as part of a two-student bachelor project.

The paper is organized as follows. In section 2 we discuss some of the ideas we have pursued. In section 3 we describe some of the facilities we have added in the improved system. Section 4 describes in detail our strategies and how the agents commit to goals. Finally, we conclude our work by discussing possible improvements of our system and the contest in section 5.

* Corresponding author.

2 System Analysis and Design

We chose to implement the system using Python as it is very fast and convenient to implement experimental systems in this language. Other useful features of Python are support of multiple programming paradigms, compact code and dynamic typing. We did not use any multi-agent programming languages because we wanted to have complete control of everything in the implementation. Last year we used Python 2 and we decided to upgrade to Python 3.

In order to make sure that our changes during the implementation phase improved our system, all new algorithms and architecture changes were tested against the older versions by comparing the data collected from the new statistics to see if the change made any differences.

2.1 Testing Moise

This year we wanted to try to implement some kind of organization for our system, so we made a substantial test implementation as part of a two-student bachelor project using the Moise organizational model [1]. We chose Moise because we have previous experience using it in combination with the Jason platform [3].

The Moise organizational model [1] is a formalism for organizational multi-agent systems where an organization is divided into three dimensions: structural, functional and deontic specification. The structural specification uses the concepts of *roles*, role *relations* and *groups* to build the individual, social and collective structural levels of an organization. Here, the roles an agent can enact are defined, and it is furthermore defined how roles are linked, e.g. by allowing agents enacting different roles to communicate. The collective level is specified using the notion of groups, in which it is determined which roles are allowed to be enacted and what links exists between agents both within internally in the group and with external agents. The functional specification specifies missions and plans using a so-called *social scheme* which is a goal decomposition tree that has as root the goal of that scheme. The responsibilities for each subgoal in a scheme are distributed in missions, which means that an agent choosing to commit to a mission effectively chooses to commit to the goals of that mission. The subgoals are created using the operators sequence, indicating that a goal is fulfilled when the sequence of subgoals are fulfilled, choice, in which a goal is fulfilled when a single subgoal is achieved, and parallelism, where all subgoals must be fulfilled, but no specific order is required. The deontic specification is the relation between the structural and functional specifications: it specifies on the individual level the permissions and obligations of a role on a mission. It makes it possible to specify that an agent enacting a certain role is obligated (or permitted) to commit to certain missions, and is therefore obligated (or permitted) to commit to the goals of that mission.

We follow the approach of S-Moise$^+$, which is an open-source implementation of an organizational middleware that follows the Moise-model [2]. Among other things it consists of a special agent, the *organizational manager*, which maintains consistency in the organization, i.e. by making sure that a single agent cannot enact two incompatible roles at the same time. This is done by letting the agents communicate with the manager when they want to join a group, enact a role or commit to a mission. If any such event is a violation of the organizational specification, the organizational manager will not allow it.

The plan trees and social schemes of Moise have a large potential, due to the fact that they will make sure that the right amount of agents will work together toward the best goal. We have chosen to only plan for a single subgoal for each agent, because of the very dynamic nature and the size of the map and number of agents. This makes the plans sufficiently small for the agents to coordinate themselves using direct communication, which makes the plan trees unnecessary.

It might be possible to split the agents into smaller groups to perform more coordinated plans, like finding the opponent's zones etc., but we did not have the time to try to implement groups. In the end we decided not to use Moise as we found that the benefits did not outweigh the needed effort to get the computation under the time limit, due to the quite large communication overhead of the organizational manager.

2.2 Agent Behaviour

Our resulting system is a decentralized solution with a focus on time performance. The communication between the agents relies on shared data structures as this is a very fast way to communicate for the agents. The `Runner` class which coordinates communication is described in more detail in section 3.3.

Instead of letting the agents find goals based on their own knowledge alone they use the distributed knowledge of the entire team. This does add some communication which in some cases is unnecessary but in most cases the extra knowledge will produce better goals for the agents.

In each step each agent will find its preferred goals autonomously and assign each of them a *benefit* based on its own desires (i.e. the type of agent), how many steps are needed to reach the location and so on. In order to make sure that multiple agents will not commit to the same goal they communicate in order to find the most suitable agent for each goal. This is done using our auction-based agreement algorithm which will be discussed in more detail in section 4.3.

The agents in this contest are situated in an inaccessible environment which means that the world state can change without the agents noticing from step to step, e.g. if the opponent's agents move outside our agents' visibility range. Hence our agents should be very reactive to observable changes in the environment.

The agents are only proactive in a few situations. The most important one being the communication between a disabled agent and a repairer. They use their shared knowledge in order to decide which of the agents should take the last step and who should stay, so that they eventually are standing on the same vertex instead of simply switching positions. This is implemented by considering the current energy for each agent.

Some of our agents also attempt to be proactive by for example parrying if an opponent saboteur is on the same vertex. Furthermore, repairers will repair wounded agents since they are likely to be attacked again.

2.3 Random Generation of the Map

Last year all maps had one high-valued area, indicated by numbers on the vertices, as seen in figure 1. For this setting we developed an algorithm which places the agents in defensive positions inside the area in order to defend it. For more information we refer to the paper about our system from 2011 [6].

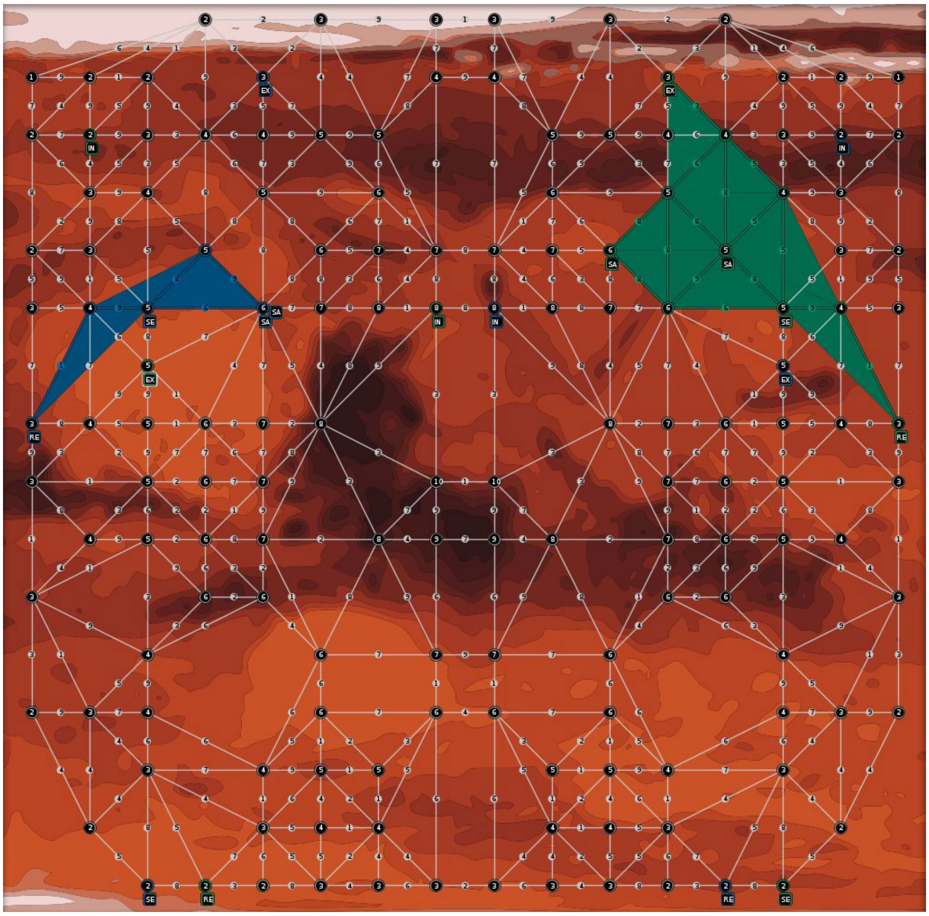

Fig. 1. An example of a map in the MAPC 2011

This year the map generation algorithm has been updated to create more than one high-valued area. An example of this can be seen on figure 2, where the size of a vertex represents its value. In some cases this lead to situations where our agents would protect a single good area even though it would be better to make smaller groups and have control over several areas. Therefore our previous solution would only be effective in special cases, so we have implemented a

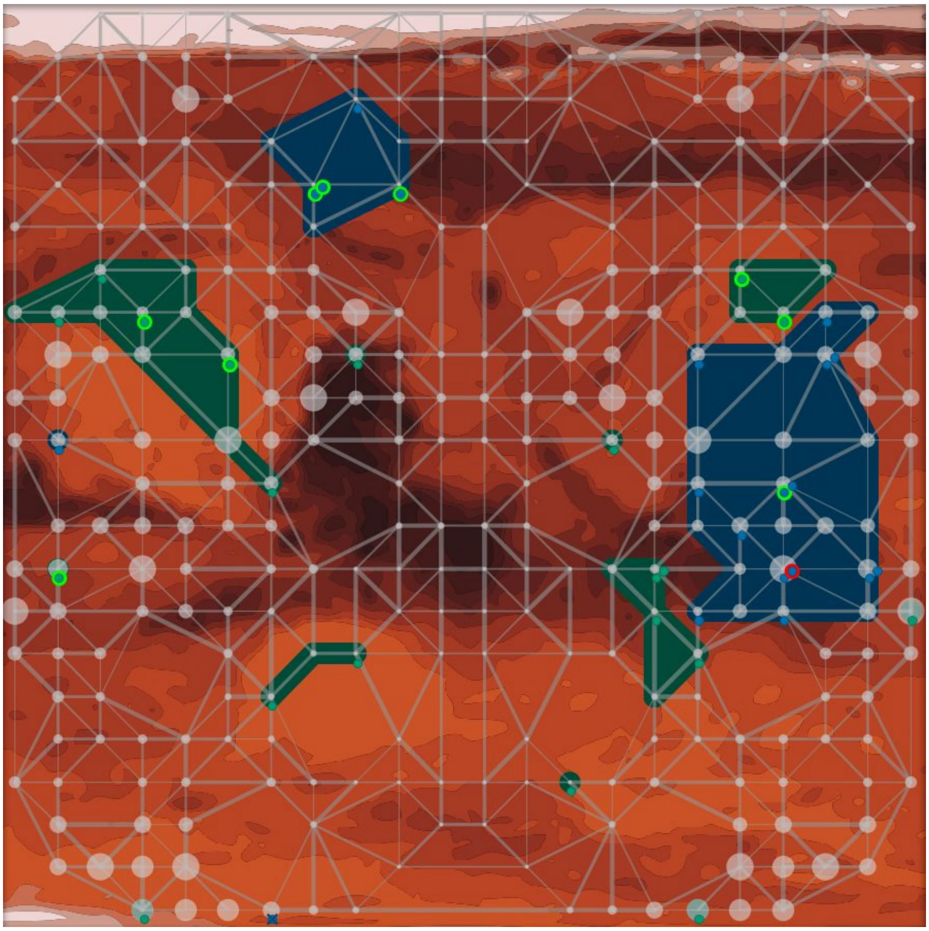

Fig. 2. An example of a map in the MAPC 2012

new algorithm which takes multiple areas into consideration. The new solution is actually much simpler and it works well for both maps with multiple areas and maps with a single, high-valued area. In section 4.2 we describe the main properties of this algorithm.

3 Software Architecture

The software architecture, including the auction-based agreement approach, is thoroughly described in the paper about our system from 2011 [6] and will only be described briefly here. The rest of this section will describe a few minor facilities added this year.

3.1 Considerations

The competition is built on the Java MASSim-platform and EISMASSim framework which makes it easy to implement a system quickly without spending time on server communication and protocols. However, we did not utilize this framework but chose to implement our system in Python exclusively to have better control and complete knowledge about the implementation. Another solution based on EISMASSim, ActiveMQ and the Java implementation of Python, called Jython, was implemented as well. This solution was discarded due to performance issues. We also considered using a multi-agent framework such as Jason, but due to prior experiences, we thought that the benefits where outweighed by the increased complexity and thus chose to implement our own framework. We chose Python as we think it is in many ways superior with respect to development speed and succinctness compared to Java, C#, C++ and other languages that we have experience with. Furthermore Python supports multiple programming paradigms, including the functional, which has quite effective for this setting.

Last year we used a decentralized solution where the agents shared their percepts through a shared data structures but each kept their own copy of the graph representing the environment. The increase in the number of agents and the size of the maps for this year's competition, forced us to rethink and reimplement the percept sharing. To efficiently handle the increased amount of information, all agents share a single instance of the graph. To avoid deadlocks, percepts that lead to updates in this graph are handled with synchronized queues which allow safe exchange of data between multiple threads.

3.2 Testing Using Flags

A lot of testing was required for verifying that our system was improved compared to our previous system, so we needed an easy way to select which algorithms to use. In order to be able to run several instances of the program, we decided to create program arguments, or flags, for the system. In the beginning we had a configuration file in which we set flags. This was not a very practical way to do it as we had to have multiple configuration files in order to run more instances of the program. These flags make it possible to specify which algorithms the system should use. The help page for our multi-agent system where the different flags are described is shown below:

```
$ python ./bagent.py -h
usage: bagent.py [-h] [-b] [-d] [-a] [-w] [-l] [-v {0,1,2}] {a,b,Python-DTU}

positional arguments:
  {a,b,Python-DTU}      agent name prefix

optional arguments:
  -h, --help            show this help message and exit
  -b, --buy             make the agents shop for upgrades
  -d, --dummy           dummy agents
  -a, --attack          do attack
```

```
-w, --weak_opp         attack EXP and INS in the start of the simulation
-l, --load_pickle      load vertices from pickled data
-v {0,1,2}, --verbosity {0,1,2}
```

The flags are used to start multiple instances of the system using different strategies. For example we can test whether it is better to use our buying strategy by starting the server and then start two instances of the system where the flag `-b` was passed to one of them. This was used to test whether it was beneficial to use our heuristics, but as we found that this was not the case we have removed them from the system.

3.3 Code Structure and Files

We briefly describe the main classes and files:

global_vars.py: We have all our global variables in this file. They are mainly used to make the implementation more dynamic and easier to maintain.

comm.py: This is the file where we have implemented the `Agent` class and the procedures used to communicate with the server. The `Communicator` class is implemented as processes such that all the agents can send and receive messages at the same time. The logic of the agents are implemented in the util.py and algorithms.py files.

bagent.py: This is where the main program is started and where the flags are parsed. It is also in this file that our `Runner` class is implemented. The `Runner` class starts and lets the agents do their calculations in a sequential fashion.

algorithms.py: Most interesting of our algorithms are implemented in this file, including:
- The *greedy zone control* which will be discussed in section 4.2.
- The *get goals algorithm* called by each agent. This algorithm is discussed in more detail in the paper about our system from 2011 [6].
- The *best-first search* used by each agent in order to find specialized goals according to their type.

util.py: We have implemented our graph representation of the map in this file. The file also includes a timer which was used to find bottlenecks in our code.

4 Strategies, Details and Statistics

In the competition each step of each achievement is exponentially harder to reach than the previous, thus our agents need a way to change their goals as the simulation progresses. We describe our strategy for getting achievements in section 4.1 and our zone control strategy in section 4.2. We describe how the agents decide what to do in section 4.3 and finally how communication works in section 4.4.

4.1 Getting Achievements

In the beginning every agent will work towards achieving as many type specific goals as possible in a more or less disorganized fashion, e.g. the inspector will inspect every opponent it sees.

We do this to achieve as many achievements as possible as fast as possible. We tried implementing different heuristics to improve the first part of the strategy. We considered the following heuristics:

Survey Heuristic: The agents always survey the vertex with the most outgoing edges if the steps needed to reach the vertex are the same (figure 3). The idea is to get survey achievements faster, but it turned out that even though we got the first few achievements faster, the last ones were achieved a lot later using this heuristic, so we did not use it.

Probe Heuristic: The agents probe the vertex with the highest valued neighbours (figure 4). This worked very well in the scenario from 2011, but in the 2012 scenario it can be more beneficial to first find a lot of potentially high valued areas which can be probed later. This can be achieved using a random walk, which will reduce the time in each area increasing the chance that the agents might find more areas in less steps. We chose not to use the probe heuristic since a random walk was more successful.

Attack Vulnerable Opponents: This heuristic is only applied in the first 80 steps (a simulation has 750 steps). We prefer to attack agents that cannot parry, as this will get us more successful attacks. Furthermore, as added value this will also lead to fewer successful parries for the opponent. This turned out to give us a slight advantage in the beginning of the simulation, so we chose to use it.

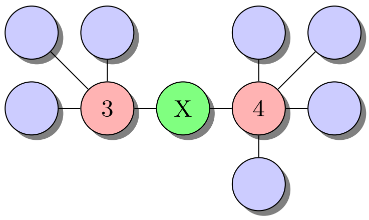

Fig. 3. Illustration of the heuristic values our agents would get trying to survey, standing on the green vertex. The vertex to the left has a heuristic value of 3 because it has three outgoing edges, whereas the one on the right has a slightly better heuristic value of 4.

After a certain number of steps the agents will proceed to the zone control part of our strategy. The sentinel is the only agent surveying after step 30. The explorers keep probing until step 150 and will probe in our target area for the next 50 steps to make sure we control as many vertices as possible. Afterwards they will follow the zone control strategy. All other agents begin zone control after step 150.

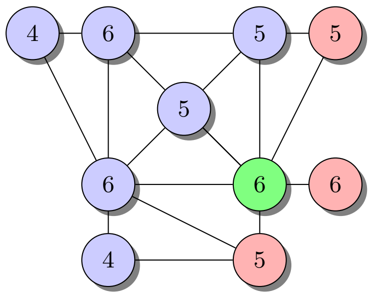

Fig. 4. Illustration of the heuristic values our agents would get trying to probe, standing on the green vertex where the blue ones are owned with the given value. The heuristic value of the red vertices are calculated by taking the mean of the known neighbouring vertices.

4.2 Zone Control

The zone control part of our strategy uses a very simple, but surprisingly effective, greedy algorithm. The algorithm works by first choosing the node with the highest value, and then by choosing a potential neighbour node. The potential value of choosing that node is then calculated as the value of the node plus the sum of all the neighbours which, according to the graph coloring algorithm [7], will be owned if the potential node is chosen. For each agent, the algorithm will choose the best node according to some parts of the graph coloring algorithm. If a vertex has not been probed the algorithm will use the value 1. This way we take some of the area coloring algorithm from the contest into consideration and as it is an inaccessible environment this is the best we could achieve.

This algorithm will to some extent choose the optimal area or several areas which are still fairly easy to maintain, even though our choices are limited by our (partial) knowledge of the map and the missing parts of the area coloring algorithm.

During the zone control part every type of agent has a specific job.

- *Repairers and saboteurs* do not directly participate in the zone control, instead they are trying to defend and maintain the zone.
- *Inspectors* keep inspecting from their given expand node, because the opponents might have bought something which we need to make a counter move against.
- *Explorers* will probe unprobed vertices within the target zone. When all vertices are probed they are assigned a vertex by the zone control strategy.
- The *sentinels* will stay on a vertex assigned by the zone control strategy and will parry if some of the opponent's saboteurs move to the sentinels position.

The last important change in the state of mind of the agents is that after step 150 the saboteurs start buying. They buy exactly enough extra health so that they will not get disabled by a single attack from an opponent saboteur that has not upgraded his strength. Furthermore we buy enough strength to disable any opponent saboteur in a single attack by buying strength for all our saboteurs every time we inspect the opponent saboteurs and find that it has more health than all other inspected saboteurs. This buying strategy is chosen in hope of dominating the map which will make it possible to gain control of the zone we want. The advantage is that we only try to out-buy in one specific field, thus we are unlikely to use all our achievement points. As this is a quite aggressive buying strategy we had to wait to step 150 to have enough achievement points to execute it.

4.3 Making Decisions

The agents need a consistent way of figuring out what to do. We do this by letting every agent find the nearest goals according to their type. They do this by using a modified best-first search (BFS) which returns a set of goals. To make sure that every agent always has at least one goal the BFS returns as many goals as we have agents. This is a very agent-centered procedure meaning the agents simply commit to the goal with the highest benefit, instead of coordinating any bigger schemes. However, since the goals are more or less dependent on each other there is some implicit coordination. For example the repairers will often follow the saboteurs as these search for opponents and thus more often will share a vertex with an opponent saboteur and get disabled.

To decide which goal to pursue the agents use an auction algorithm. Every agent can bid on the goals they want to commit to and will eventually be assigned the one they are best suited for. This results in a good solution, which however might not be optimal. For further details we refer to the paper about our system from 2011 [6].

Even though our planner calculates a few turns ahead the agents recalculate every turn. We do this to adapt to newly discovered obstacles and facts, such as an opponent saboteur or the fact that the agent has been disabled. The agents will not end up walking back and forth as their previous goal will now be one step closer, thus the benefit of the goal has increased. If another goal becomes more valuable it means that it is a better goal than the one the agent was pursuing, thus changing the commitment makes sense, so we do not lose anything on recalculating each turn.

4.4 Communication

Communication and sharing of information is extremely important in any multi-agent system. In our system every percept received by the agents are stored in a shared data structure so that all agents have access to the complete distributed knowledge of the team at all times.

Actual communication in our system only happens when the agents are deciding what to do. When they are figuring out what to do the auction-based agreement algorithm is used on conflicting goals and thus two agents will never pursue the same goal.

5 Conclusion

In the process of reimplementing and improving the Python-DTU multi-agent system we have analysed the changes to the competition and used our findings to design and implement better algorithms for the increasingly complex tasks. We have considered imposing an explicit organization upon the agents, and for this purpuse we experimented with the Moise organizational model. While it had some advantages, such as the being able to ensure that the right amount of agents work together toward a certain goal using by use of roles and plan trees, we decided not to use Moise in the final version of our system, as its benefits did not outweigh the communication overhead caused by the organizational manager in the organizational middleware, S-moise$^+$.

All improvements to the algorithms are quite simple, but are nevertheless effective at reaching their goals. The simplicity and specialized approach is probably one of our strengths, as it makes it easy to implement special cases when certain improvements of the algorithms were necessary. Having aggressive saboteurs was also an advantage as it lead to the opponents being disabled often, which in turn gave us a larger zone score. Our greatest weakness was that our uncompromising attempt to have the strongest saboteurs could be countered by buying enough health on a single saboteur to make us use most of our achievement points for improving all of our saboteurs. This could lead to a large difference in step score gained from achievement points each step.

The many advanced programming constructs in Python, e.g. lambda functions, list comprehensions and filters made it possible to implement algorithms very efficiently.

One thing we have noticed during the competition is that it does not seem to pay off to buy anything other than health and strength. This meant that a lot of teams had more or less the same strategies. We think it could be interesting if many kinds of strategies could be sufficiently effective so that we might see the teams following different strategies. One idea could be to introduce ranged attacks which could be achievable through upgrades and should be limited by visibility range. This could allow for some other strategies, since the agents need to figure out where to hit the opponent a few steps in the future and how to avoid getting hit themselves. Furthermore, the teams will need to use their inspectors even more to find out whether or not to avoid possible ranged attacks from the opponent.

References

1. Hübner, J.F., Sichman, J.S., Bossier, O.: A Model for the Structural, Functional, and Deontic Specification of Organizations in Multiagent Systems. In: Bittencourt, G., Ramalho, G. (eds.) SBIA 2002. LNCS (LNAI), vol. 2507, pp. 118–128. Springer, Heidelberg (2002)
2. Hübner, J.F., Sichman, J.S., Boissier, O.: S-moise$^+$: A Middleware for Developing Organised Multi-Agent Systems. In: Boissier, O., Padget, J., Dignum, V., Lindemann, G., Matson, E., Ossowski, S., Sichman, J.S., Vázquez-Salceda, J. (eds.) ANIREM 2005 and OOOP 2005. LNCS (LNAI), vol. 3913, pp. 64–78. Springer, Heidelberg (2006)

3. Bordini, R.H., Hübner, J.F., Wooldridge, M.: Programming Multi-Agent Systems in AgentSpeak Using Jason. John Wiley & Sons (2007)
4. Boss, N.S., Jensen, A.S., Villadsen, J.: Building Multi-Agent Systems Using *Jason*. Annals of Mathematics and Artificial Intelligence 59, 373–388 (2010)
5. Vester, S., Boss, N.S., Jensen, A.S., Villadsen, J.: Improving Multi-Agent Systems Using *Jason*. Annals of Mathematics and Artificial Intelligence 61, 297–307 (2011)
6. Ettienne, M.B., Vester, S., Villadsen, J.: Implementing a Multi-Agent System in Python with an Auction-Based Agreement Approach. In: Dennis, L.A., Boissier, O., Bordini, R.H. (eds.) ProMAS 2011. LNCS, vol. 7217, pp. 185–196. Springer, Heidelberg (2012)
7. Behrens, T., Köster, M., Schlesinger, F., Dix, J., Hübner, J.: Multi-Agent Programming Contest — Scenario Description — 2012 Edition (2012), `http://www.multiagentcontest.org/`