Research Lab "Multi-Agent Programming Contest 2014"

Final Report

Artur Daudrich[⋆], Sergey Dedukh[⋄], Manuel Mittler[⊙], Michael Ruster[◦], Michael Sewell[†], and Yuan Sun[▲]

University of Koblenz-Landau, Koblenz Campus

Table of Contents

1 Motivation and Background† 1
1.1 The "Agents on Mars" scenario† 1
1.2 The *MAKo* (Multi-Agents Koblenz) team† 1
2 Scientific Background and Fundamentals 2
2.1 MAPC: Contest and Scenario 2
2.2 Agent Programming Concepts 2
2.2.1 BDI. 2
2.2.2 Formal Methods. 8
2.2.3 Negotiation and Argumentation. 12
2.2.4 Agent Societies. 15
2.3 Agent Programming Languages 15
2.3.1 Situation Calculus. 15
2.3.2 GOLOG. 17
2.3.3 FLUX. 18
2.3.4 Jadex. 20
2.3.5 AgentSpeak(L). 26
2.3.6 Jason. 28
2.3.7 Choice of a programming language. 30
3 Team Organisation and Collaboration Tools 31
4 Architectural (?) Structure 33
4.1 Agents 33
4.2 Simulation Phases 33
5 Algorithms and Strategies 33
5.1 General Strategy Overview 33
5.2 Agent Specific Strategies† 34
5.3 Exploration 36
5.3.1 Cartographer Agent 36
5.3.2 Distance-Vector Routing Protocol 38
5.3.3 JavaMap 43
5.4 Repairing 44
5.5 Zone Forming 45
5.5.1 Zone Calculation† 45
5.5.2 Zone Finding Process. 51
5.5.3 Zone Building Roles and the Lifecycle of a Zone. 52
6 Implementation Details 54
6.1 BDI in AS(L) and Jason▲ 54
6.2 Information Flow 59
6.3 Lifecycle of one Step 59
7 Discussion and Conclusion 59
7.1 Competition results⊙,∘ 59
7.2 Lessons learned⊙/∘ 62

1 Motivation and Background†

The *Multi-Agent Programming Contest* is an annual online programming contest hosted by the Clausthal University of Technology since 2005. Participation is free to any interested groups, and in former years rewards were given to winning teams in the shape of book vouchers. This year's MAPC, however, was an "informal" contest, and no prizes were awarded. The aim of the MAPC is to promote academic interest in the field of multi-agent systems, that is, systems in which multiple artificial agents have to collaborate to achieve a goal.

The subsections are quite short. We probably should merge them later on. Also, we'll have to take a look of just about how much we want to talk about the MAPC Mars scenario at this point already. If we keep it as general as we currently are, this will probably be fine. Within the fundamentals part, we can then talk about agents, actions, roles, attributes, what is a zone and so forth.

The nature of the task in which the agents compete in the MAPC has changed over the years, but since 2011 it has been the same "Agents on Mars" scenario, which will be described below. The winner and further rankings of each year's contest are determined by having each group's agent systems face off against the other in a tournament, and awarding points to each team according to their performance in each match.

1.1 The "Agents on Mars" scenario†

The "Agents on Mars" scenario is the one that has been used in the yearly MAPC since 2011. In it, two opposing teams of agents are placed on vertices in a randomised graph. Each vertex in the graph has a value which is used for scoring, and agents can traverse the graph by moving along the edges connecting the vertices. The "Agents on Mars" name relates to the fictional background "story" of the scenario: Man has populated Mars, and must find and occupy wells of water on the surface of the planet and protect them from "pirates".

The simulation is turn-based, and each agent can perform one action per turn. There are 28 agents in each team, and each agent belongs to one of five different agent classes, where the agent's type determines the kind of actions the agent can perform and other values used to further differentiate agent classes. The goal of each match is to have a higher score than the opponent's team at the end of a predetermined number of steps (400 in the 2014 MAPC). A high score is achieved by finding localised parts of the graph which contain high-value vertices, surrounding these "zones" with one's own agents, and protecting them from enemy agents' attacks. The full background story, as well as a more detailed official description of the scenario, can be found in the scenario description provided by the MAPC organisers [2].

1.2 The *MAKo* (Multi-Agents Koblenz) team†

The German University of Koblenz-Landau participated in the 2014 MAPC with a small team of graduate students in the scope of a research lab. The students who participated in research lab until its conclusion were Artur Daudrich, Sergey Dedukh, Manuel Mittler, Michael Ruster, Michael Sewell and Yuan Sun. The research lab spanned a single semester and consisted of an initial seminar phase and the longer project phase, where the students designed and implemented the multi-agent architecture used to participate in the MAPC.

2 Scientific Background and Fundamentals

2.1 MAPC: Contest and Scenario

Write this section

What is the general scenario? Agents on mars trying to find water in a competitive manner against another team. Explain the concept of zones, gaining points, achievements and also how agents differ from each other. Introduce the notion of *zoning* as the process of finding and forming a zone (including upkeeping by defense?). I've put this section first so the later chapters can already rely on the reader knowing what the scenario is about. Furthermore, we can then directly rule out concepts which we are presenting by applying them theoretically onto the scenario/our needs.

2.2 Agent Programming Concepts

Write this section

2.2.1 BDI.▲

Using cognitive modelling techniques for simulating human behaviour, without requiring people interactions can save a lot of people forces, resources, time and money. So that a variety of researchers are contributing to intelligent agents field. Beliefs, desires and intentions (BDI) agent is a kind of intelligent agent. BDI model describes the basic characteristics of agents' mental state since the BDI logic system is easy to be realised in the computer, and has been wildly applied in these fields. In recent years, many scholars have used Java, Jason or some other languages to implement BDI agent model in computer.

In 1987, Bratman[12] discussed the relationship among beliefs, desires, intentions and actions as well as considering that they play important roles in option behaviours. This is the foundation of BDI model and BDI logic. In 1991, Rao and Georgeff[22] modelled the BDI agent behaviour and treated beliefs, desires and intentions as three modal operators and applied BDI agent to airline traffic management. Nowadays, the research on BDI agents are not only used in high value domains but also in daily lives. We can see the applications are not only in high technology industrial aspect as air-plane or space shuttle but also in commercial field or entertainment such as robot soccer games.

The BDI model is a popular and well-studied architecture of agent for intelligent agents situated in complex and dynamic environments. The model has its roots in philosophy with Bratman's theory of practical reasoning[42]. Practical reasoning involves two important processes: deciding what goals we want to achieve, and how we are going to achieve these goals. The former process is known as deliberation, the latter as means-ends reasoning[52]. When an agent is placed in an environment, it should decide what to do and how to do. There are a lot of options of affairs states, but not all of them are good choices. Some other affairs more or less have influences on the feasibility of achieving these goals. The deliberation process is to understand and filter what options are available, in addition, generate the set of alternatives which will be chosen as following. These chosen options become intentions which can be treated as the outputs of deliberation. For example, if you are standing in a supermarket and very thirsty, then you are faced with a decision to choose a drink. There are a lot of options like wine, beers, milk, water and juice, however, picking up a bottle of wine is not available to you if you are younger than 18 years old. After collecting all the available options, you must choose and commit to some of them which become intentions next. Subsequently, we need the mean-ends reasoning process to plan how to achieve these intentions. Furthermore, your intention is to buy a bottle of water, then you plan to go to the shelf with water on it, and stretch your arm to get a bottle of water on the top. Finally, you execute this plan to get a water.

As a theory of practical reasoning, BDI model has three attributes that are belief, desire and intention.

Beliefs represent the informational state of the agent and be updated appropriately after each sensing action.They may be implemented as a variable, a database, a set of logical expressions, or some other data structure[40]. Belief means how the agent look at the world and it is the basis of BDI model. Belief includes the information about environment, other agents and itself. An agent needs to be allowed to update its beliefs at any time. Updating information comes from the perception of the environment, and the execution of intentions. An agent can use sensors to perceive the environment to get signals to believe. In addition, after executing some intentions, these become the information believed by the agent. Belief is not the same concept as knowledge. Beliefs are only required to provide information on the likely state of the environment, but knowledge is the realisation of a fact. Beliefs are just the state believed by agents but no one can ensure what they believe are true. Simple to say, knowledge is true belief.

Desires represent the motivational state of the agent[40]. They represent objectives or situations that the agent would like to accomplish or bring about. They are state of affairs that the agent would wish to bring about or to keep. Desires may be achieved or never achieved, and it doesn't need to believe that desires must be achieved. Desires are different from goals although they look pretty similar. Desires can be inconsistent and the agent need not know the means of achieving these desires. Desires have the tendency to 'tug' the agent in different directions. They are inputs to the agent's deliberation process, which results in the agent choosing a subset of desires that are both consistent and achievable. Such consistent achievable desires are usually called goals[52]. For example, sleeping and working may be both my desires, but they can not be my goals at the same time because they have conflicts.

Intentions are desires or actions that the agent has committed to achieve[23]. Intentions are stronger than desires. Desires are just wishes that may be achieved or may be not, but intentions to an extent are decided to be achieved. Michael Wooldridge concluded four roles of intentions playing in practical reasoning. The roles are intentions drive means-ends reasoning; intentions constrain future deliberation; intentions persist and intentions influence beliefs upon which future practical reasoning is based[52]. Intentions driving means-ends reasoning means that intentions have decisive influences on actions the agent will execute. Agents are expected to determine ways of achieving intentions. Intentions constraining future deliberation means that options that are inconsistent with this intention will not be entertained. intentions persisting means that intentions will be never given up unless the reason is rational.As we know, intentions are committed desires which can not be easily abandoned. For if I immediately drop my intentions without devoting resources to achieving them, then I will never achieve anything[52]. But when a good reason exists, I still can drop intentions instead of persisting them for a long time without achievement. If it is very clear that the intentions will never been achieved, then there is no need to keep them. Similarly, if the reason for intention is no longer true, then intentions should be given up. Another reason of dropping intentions is the intentions have been achieved already. Intentions influencing beliefs upon which future practical reasoning is based means that believing intentions will be achieved. If I adopt an intention, then I can plan for the future on the assumption that I will achieve the intention. For if I intend to achieve some state of affairs while simultaneously believing that I will not achieve it, then I am being irrational[52]. Agents should believe that they believe there is at least some way that the intentions could be brought about and believe that under "normal circumstances" agents will succeed with the intentions, or say it in another way, agents do not believe they will not bring about their intentions. Generally speaking, intentions are not random ideas, but the wants to a reasonable extent. It plays an important role in BDI model that leading to actions, constraining future deliberation and influencing future beliefs. Specifically, the agents should drop off some intentions at times to avoid resource wasting. It is necessary to keep a good balance between these different concerns.

For some reason, the spacing here between paragraphs seems to increase at times. I haven't quite figured out why and when this happens.

Beliefs, desires and intentions are three attributes of BDI model and constitute the foundation of BDI agents. While, some other components building connection between beliefs, desires and intentions are also indispensable to implement BDI agents and make the BDI architecture completed. Several basic components can be found in the following figure, which is a brief BDI architecture. A variety of BDI agents have been designed to fit the unstable environment and to complete all sorts of tasks. Different BDI agents have different architectures, but their core ideas are same.

The sensors of the BDI agent perceive the environment and convert the perceptions to signals as the inputs to the belief revision function. It will collect the perceptions from outside as well as the beliefs which are stored in the beliefs set. After mapping the information, computing and revising using belief revision function, the new beliefs will set into the beliefs set. The belief revision function prefers minimal change rather than modifying a lot. There are not big differences between the revised beliefs set and the previous one to preserve as much information as possible by the change[51]. The belief revision function is used to keep the beliefs set being updated to fit the unstable environment, on the other hand, it can avoid the inconsistent situations occurring among those beliefs like the agent believes what it doesn't believe.

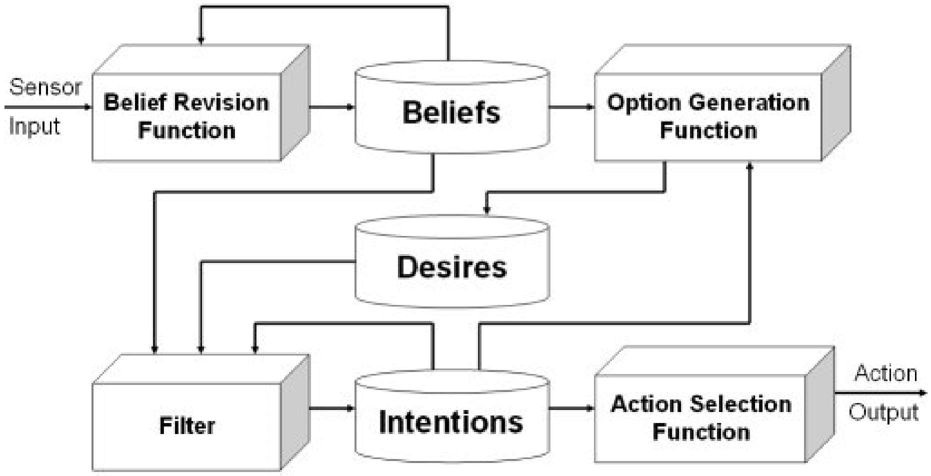


Fig. 1: Brief BDI architecture [3]

The beliefs set contains information about the current environment which the agent has. The data in beliefs set may be sentences, rules or some other manifestations. In the AGM (named after the names of their proponents, Alchourrón, Gärdenfors, and Makinson)approach, an agent's beliefs are modelled by a deductively closed set of formulas called a belief set[17]. The current set of beliefs is represented by a deductively closed set of logical formulae K called belief base, the new piece of information is a logical formula P, and revision is performed by a binary operator * that takes as its operands the current beliefs and the new information and produces as a result a belief base representing the result of the revision[51]. Many researchers are doing researches in belief revision using AGM approach. The option generator reads the beliefs information and returns a list of options which are current desires into the desires set. It determines the desires depending on the agent's current beliefs and current intentions. The desires set contains many desires are possible courses of actions available to the agent, and These desires can be no matter achieved or not. The filter determines the agent's intentions depending on current beliefs, desires, and intentions. It needs to consider about more situations than the functions in previous steps. Desires will become more rational after filtering.

@yuansun1990: I couldn't find it quickly but please link to the original work introducing AGM as well. And naturally, also quickly look over it so you know that your statements are correct.

The intentions set stores the agent's current focus, which are going to be executed or committed to be executed at some time. Once an intention is adopted, it should not be immediately dropped out because of the commitment. But in some situations, the intentions should be given up depending on three commitment strategies having been proposed in Rao and Georgeff's work: blind, single minded and open minded. A blindly committed agent is an agent who maintains his intentions until he believes that he has achieved them. A single minded committed agent is an agent who maintains his intentions as long as he believes that they are still options. An open minded committed agent is an agent who maintains his intentions as long as they are still goals[34]. The action selection function determines the actions to perform depending on current intentions. We would achieve nothing if we just have intentions instead of knowing how to do it. Normally, there is a plan library of mapping between the intentions and actions. The intentions go through the planner and the actions which mapping the corresponding intentions are founded. At last, a plan of how to achieve the intentions come out and the agent will execute these actions.

For understanding the relationships between the seven main components of BDI agent architecture, one table is presented as followed:

Component	**Meaning**	**Formalisation**
Beliefs set	Information about the current environment which the agent has	B
Belief revision function	determines a new set of beliefs depending on perceptual inputs and the agent's current beliefs	$B \times P \to B$
Options	determines desires depending on the agent's current beliefs	$B \times I \to D$
Desires set	possible courses of actions available to the agent	D
Filter	determines the agent's intentions depending on current beliefs, desires, and intentions	$B \times I \times D \to I$
Intentions set	the agent's current focus	I
Action selection function	determines an action to perform depending on current intentions	$I \to A$

Table 1: Components of brief BDI agent architecture

This table shows the order of using the seven components of BDI architecture as well as giving the formulas of each function. There, Bel is a set of all possible beliefs, Des is a set of all possible desires and Int is a set of all possible intentions. Therefore, an agent state can be presented as (B, D, I) with $B \subseteq Bel, D \subseteq Des, I \subseteq Int$. P is a set of current perception which are obtained by the sensors of the agent. we can understand the the process of BDI agent working better through seeing this table. Firstly, B has stored some beliefs which are read by BRF while it getting the perception from the sensors. After operating BRF, some of beliefs are removed, some are added, some are modified and so on. So the new beliefs set B built on basis of P and original B. Subsequently, Options use the new B and current Intentions set I to determine the desires set D and store it. Moreover, the filter select intentions by referencing B, D, I, then the new intentions set comes out. Finally, the action selection function makes a plan to execute actions to achieve I.

The process of $B \times P \rightarrow B$, $B \times I \rightarrow D$ and $B \times I \times D \rightarrow I$ belongs to deliberation. They are deliberated in-depth gradually and the range of intentions are narrowed, especially the filter which should consider of all there datasets. At last, the intentions are limited in particular ranges. The plans will be made more effective and more targeted. $I \rightarrow A$ can be treated as the process of means-ends reasoning, whose output is planning. B, DI are connected to function parts instead of connecting to each other directly. They are just databases and need rules or mechanism to help them execute actions.

With the increasing needs of intelligent agents, more and more applications base on BDI model are applied in our life. PRS and dMARS are both BDI-based systems for the reaction control system of the NASA Space Shuttle Discovery. Additionally, an air-traffic management system, OASIS, is well-known as a BDI-dased agent. The system architecture for OASIS is made up of one aircraft agent for each arriving aircraft and a number of global agents including a sequencer wind modeller coordinator and trajectory checker[40]. Furthermore, robot soccer which are designed using BDI model becomes very popular in universities. We can feel that, BDI agents bring many profits to human beings, they make the life more convenient. However, it still has development space in this field.

Although the BDI model is developed during about 30 years, some obstacles are not overcome and some challenges are still there. Most BDI implementations do not have an explicit representation of goals. The agents should reason the goals from the current beliefs and intentions. Besides, the BDI model contains three attributes, beliefs, desires and intentions. In some situations, not all the three attributes are needed. Sometimes, an agent collect the beliefs and jump to intentions directly without desires. However, for some distributed multi-agents, just three attributes are not sufficient to execute the actions. Furthermore, the agents in the multi-agents system don't have a explicit mechanisms for interaction and integration among them. When an increasing number of agents join the system, the interaction with each agent will be more and more difficult. As an intelligent agent, the BDI agent don't have a good ability to learning from from past behavior or other agents' behavior. So that the rate of development will not be high if lack mechanisms to learn from others. However, BDI model has its own advantages. Beliefs, desires and intentions are similar to the mental activities of human beings. Therefore, it is not easy to construct the logics or mechanisms for it. With the wildly used of computers and mobile devices, the situation of multi-agent interaction will be better. As many computer languages and logic languages are grasped by more people, the BDI agent will bring human beings more surprises.

An introduction of beliefs, desires and intentions model is presented in this section. The BDI agent belongs to intelligent agent which are autonomous, computational entities. The BDI agent executes actions on the basis of BDI model that containing three main attributes which have close relationship with each other. The brief BDI agent architecture is a clear description of process of

BDI agents work. And They follows the practical reasoning theory. Different BDI implements show different architectures, but the core idea of these agents are still beliefs, desires and intentions. With an increasing number of BDI applications go into the humans life, more challenges come up too. The BDI model has its own advantages and disadvantages, but I still believe that it can bring more surprises to own life in the future.

2.2.2 Formal Methods.$^\diamond$

One of the challenges of multi-agent systems is making sure that the agent will not behave in an unacceptable or undesirable way. Agents may act in complex production environments, where failure of a single agent may cause serious losses. Formal methods have been used in computer science as a basis to solve correctness challenges. They represent agents as a high-level abstractions in complex systems. Such a representation can lead to simpler techniques for design and development.

There are two roles of formal methods in distributed artificial intelligence that are often referred to. Firstly, with respect to precise specifications they help in debugging specifications and in validation of system implementations. Abstracting from specific implementation leads to better understanding of the design of the system being developed. Secondly, in the long run formal methods help in developing a clearer understanding of problems and their solutions. [35]

To formalise the concepts of multi-agent systems different types of logics are used, such as propositional, modal, temporal and dynamic logics. In the following several paragraphs these logics, their properties and introduced operators will be briefly discussed. Describing the details of interpretations and models of each individual logic is not the purpose of this report and is left out for further reading.

Propositional logic is the simplest logic and serves as the basis for logics discussed further in this section. It is used to represent factual information and in our case is most suitable to model the agents' environment. Formulas in this logic language consist of atomic propositions (represinting known facts about the world) and truth-functional connectives: $\wedge, \vee, \neg, \rightarrow$ which denote "and", "or", "not" and "implies", respectively [19].

Modal logic extends propositional logic by introducing two different modes of truth: possibility and necessity. In the study of agents, it is used to give meaning to concepts such as belief and knowledge. Syntactically, modal operators in modal logic languages are defined as $\diamond$ for possibility and $\square$ for necessity. The semantics of modal logics are traditionally given in terms of sets of so-called *possible worlds*. A world here can be interpreted as a possible state of affairs or sequence of states of affairs (history). Different worlds can be related via a binary accessibility relations, which tells us which worlds are within the realm of possibility from the point of view of a given world. In the sense of the accessibility relation, a condition is assumed *possible* if it is true somewhere in the realm of possibility and it is assumed *necessary* if it is true everywhere in the realm of possibility [27].

Dynamic logic is also referred to as modal logic of action. It adds different atomic actions to the logic language. In our case, atomic actions may be represented as actions that agents can perform directly. This makes dynamic logic very flexible and useful for distributed artificial intelligence systems. Necessity and possibility operators of dynamic logic are based upon the kinds of actions available [18].

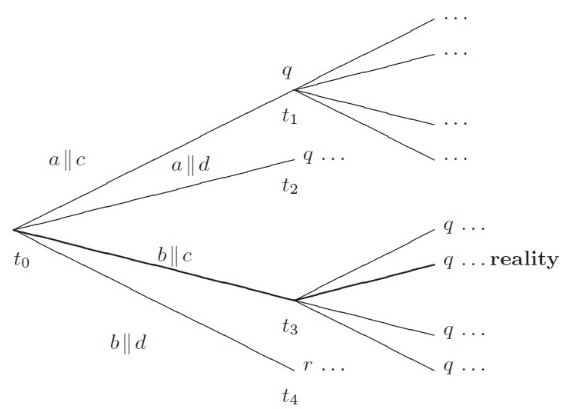
Temporal logic is the logic of time. There are several variations of this logic, such as:

Linear (or *branching*): single course of history or multiple courses of history.
Discrete (or *dense*): discrete steps (like natural numbers) or always having intermediate steps (like real numbers).
Moment-based (or *period-based*): atoms of time are points or intervals.

We will concentrate on discrete moment-based models with linear past, but consider both linear and branching futures.

Linear temporal logic introduces several important operators. $p \cup q$ is true at a moment t on a path, if and only if q holds at a future moment on the given path and p holds on all moments between t and the selected occurrence of q. Fp means that p holds sometimes in the future on the given path. Gp means that p always holds in the future on the given path. Xp means that p holds in the next moment. Pq means that q held in a past moment [35].

Fig. 2: An example branching structure of time [35].



Branching temporal and action logic is built on top of both dynamic and linear temporal logics and captures the essential properties of actions and time that are of value in specifying agents. It also adds several specific branching-time operators. A denotes "in all paths at the present moment". The present moment here is the moment at which a given formula is evaluated. E denotes "in some path at the present moment". The reality operator R denotes "in the real path at the present moment". Figure 2 illustrates the example of branching time for two interacting agents.

For modeling intelligent agents, quite often the BDI concept is used, which was described earlier in this report. BDI stands for three cognitive specifications of agents: beliefs, desires and intentions. To model logic of these specifications we will need to introduce several modal operators: Bel for beliefs, Des for desires, Int for intentions and K_h for know-how. Considering these operators, for example, the mental state of an agent who desires to win the lottery and intends to buy a lottery ticket sometime, but does not believe that he will ever win can be represented by the following formula: $DesAFwin \wedge IntEFbuy \wedge \neg BelAFwin$. For simplification in future we will consider only those desires which are mutually consistent. Such desires are usually called goals.

It is important to note several important properties of intentions, which should be maintained by all agents [45]:

Satisfiability $xIntp \rightarrow EFp$. This means that if p is intended by x, then it occurs eventually on some path. An intention following this condition is assumed to be satisfiable.

Temporal consistency $(xIntp \wedge xIntq) \rightarrow xInt(Fp \wedge Fq)$. This requires that if an agent intends p and intends q, then it (implicitly) intends achieving them in some undetermined temporal order: p before q, q before p, or both simultaneously.

Persistence does not entail success $EG((xIntp) \wedge \neg p)$ is satisfiable. This is quite intuitive: just because an agent persists with an intention does not mean that it will succeed.

Persist while succeeding This constraint requires that agents desist from revising their intentions as long as they are able to proceed properly.

The concepts introduced above may be used in each of the two roles of formal methods introduced earlier. The two most commonly used reasoning techniques to decide an agent's actions are theorem proving and model checking. The first one is more complex in terms of calculations, when the second one is more practical, but it requires additional inputs, though it does not prove to be a problem in several cases.

Considering the practical implementation, the architecture of an abstract BDI-interpreter can be described as follows. The inputs to the system are called events, and are received via an event queue. Events can be external or internal in relation to the system. Based on its current state and input events, the system selects and executes options, corresponding to some plans. The interpreter continually performs the following: determine available options, deliberate to commit to some options, update the state and execute chosen atomic actions. After that, it updates the event queue and eliminates the options which have already achieved or are no longer possible.

Add or leave out a caption for all listings. Currently we are not consistent with that.

```
BDI-Interpreter
initialise_state();
do
  options := option-generator(event-queue, B, G, I);
  selected-options := deliberate(options, B, G, I);
  update-intentions(selected-options, I);
  execute(I);
```

```
    get-new-external-events();
    drop-successful-attitudes(B, G, I);
    drop-impossible-attitudes(B, G, I);
  until quit.
```

As was mentioned above, options are usually represented by plans. Plans consist of of the name or type, the body usually specified by a plan graph, invocation condition (triggering event), precondition specifying when it may be selected and add list with delete list, specifying which atomic propositions to be believed after successful plan execution. Intentions in this case may be represented as hierarchically related plans.

Getting back to the algorithm and assuming plans as options, the option generator may look like the following. Given a set of trigger events from the event queue, the option generator iterates through the plan library and returns those plans whose invocation condition matches the trigger event and whose preconditions are believed by the agent.

```
option-generator(trigger-events, B, G ,I)
options := {};
for trigger-event ∈ trigger-events do
  for plan ∈ plan-library do
    if matches(invocation(plan, trigger-event) then
      if provable(precondition(plan), B) then
        options := options ∪ plan;
return options.
```

Deliberation of options should conform with the execution time constraints, therefore under certain circumstances random choice might be appropriate. Sometimes lengthy deliberation becomes possible by introducing meta-level plans into the plan library, which form intentions towards some particular plans.

```
deliberate(options)
if length(options) ≤ 1 then return options;
else metalevel-options :=
          option-generator(b-add(option-set(options)));
  selected-options := deliberate(metalevel-options);
  if null(selected-options) then
      return random-choice(options);
  else return selected-options.
```

Coordination is one of the core functionalities needed by multi-agent systems. Especially when different agents act autonomously and have different roles and possible actions.

One of the approaches developed by Singh [46] represents each agent as a small skeleton, which includes only the events or transitions made by the agent that are significant for coordination. The core of the architecture is the idea that agents should have limited knowledge about the designs of other agents. This limited knowledge is called the significant events of the agent. There are four main types of events:

– flexible, which can be delayed or omitted,

- inevitable, which can only be delayed,
- immediate, which the agent is willing to perform immediately,
- triggerable, which the agent performs based on external events.

These events are organised into skeletons that characterise the coordination behavior of agents. The coordination service is independent of the exact skeletons or events used by agents in a multi-agent system.

To specify coordinations, a variant of the linear-time temporal language with some restrictions is used. Two temporal operators are introduced for this purpose: $\cdot$, which is the before operator, and $\odot$, which is the operator of concatenation of two time traces, the first of which is finite. Such special logic allows a variety of different relationships to be captured.

Overall, formal methods provide a logic abstraction for multi-agent systems. They help to find self-consistent models of an agent's behavior. However, relatively high complexity does not allow these methods to be implemented in real-time systems. Therefore, the role of formal methods nowadays is limited to debugging, validation and design purposes.

In our project we unfortunately did not apply any formal methods for debugging or validating, mostly because of the limited time for development.

2.2.3 Negotiation and Argumentation.$^{\diamond}$

In a multi-agent environment, where each agent has its own beliefs, desires and goals, achieving a common goal usually requires some sort of cooperation. In most cases, it can be achieved through communication and negotiation among groups of agents. Often, negotiation is supported by some arguments which help to identify which agent is most suitable for completing a certain task. Among the reasons why one agent could be more suited than another could be the agent's better position, better resources for completing the task, importance of the current goal and so on. Some arguments can be also used to change the intentions of other agents. This could be the arguments like reserving the vertex to explore or the enemy to attack and many others. Argumentation is essential when agents don't have the full knowledge about other agents or environment. In such cases, exchanging information helps to develop the consensus and make cooperative decisions.

To negotiate effectively, a BDI agent requires the ability to represent and maintain a model of its own properties, such as beliefs, desires, intentions and goals, reason with other agents' properties and be able to influence other agents' properties [26]. These requirements should be supported by the agent programming language we choose for our project.

As was mentioned above, negotiation is performed through communication. Negotiation messages can be of the following three types: a *request*, *response* or a *declaration*. A response can take the form of an acceptance or a rejection. Messages can also have several parameters for justification or transmitting negotiation arguments. The arguments are produced independently by each agent using the predefined rules, which will be discussed later in this sub-chapter. Every agent can send and receive messages. Evaluating a received message is the vital part of negotiation procedure. Only the evaluation process following an argument may change the core agents' beliefs, desires, intentions or goals.

There are always several ways of modelling agents for negotiation. Agents can be *bounded* if they do not believe in "false"; *omniscient* if their beliefs are closed under inferences; *knowledgable* if their beliefs are correct; *unforgetful* if they never forget anything; *memoryless* if they do not have memory and they cannot reason about past events; *non-observer* if their beliefs may change only as a result of message evaluation; *cooperative* if they share the common goal [26]. For our project in most of the cases we assumed an agent as knowledgable and memoryless - agents remember only about the current round of negotiation and abolish previous round results when the new round starts. During the zone building process the agents also act as cooperative, since they share the common goal of building a zone.

For every negotiation round an agent needs three types of rules: *argument generation*, *argument selection* and *request evaluation*. We discuss them below.

Argument generation is a process of calculating the arguments for negotiation. An argument may have preconditions for its usage. Only if all preconditions are met, an agent is allowed to use the argument. To check the precondition an agent verifies if it is hold in the agent's current mental state.

In their work Kraus et al. [26] point out six types of arguments, which can be used during negotiation:

1. An appeal to prevailing practice.
2. A counterexample.
3. An appeal to past promise.
4. An appeal to self-interest.
5. A promise of a future reward.
6. A threat.

An appeal to prevailing practice refers to the situation when an agent refuses to perform the requested action, because it contradicts with one of its own goals. In this case the agent, who issued the request may refer to one of the third agent's actions in the similar situation. The algorithm of calculation of the argument here will be: find a third agent who performed the same action in the past and make sure that this agent had the same goals as the persuadee agent.

A counterexample is similar to appealing to prevailing practice, however in this case the counterexample is taken from the opponent's own history of activities. Here it is assumed that the agent somehow has the access to the persuadee's past history.

An appeal to past promise can be applied only when the a gent is not a memoryless agent. This type of argument is a sort or a reminder to the previously given promise to execute an action is some particular situation. The algorithm of checking if this argument apply is: verify that the persuadee agent is not a memoryless agent, then check if the agent received a request from the opponent in the past with promise of future reward and that reward was the intended action right now.

An appeal to self-interest is a type of argument that convinces the opponent that the performed action will serve towards one of its desires. This argument cannot be applied to knowledgable or reasonable agent, since it can compute the implications by itself. To calculate this argument an agent needs to: verify that the opponent is not a knowledgable or reasonable agent; select one desire the opponent has; generate the list of actions that will lead from the current world state to the opponent's desire fulfillment; check whether the performed action appear in the list. If such opponent's desire is found then the argument is applicable.

A promise of a future reward is a promise given by the agent to the opponent as a condition for the opponent agent to help with executing an action. On order to remember the promise, the opponent naturally should not be a memoryless agent. The calculation algorithm here is: find one opponent's desire, first consider joint desires, trying to find one that can be satisfied with help of the agent; like in self-interest argument generate a list of actions, that lead to the desire fulfillment; out of the resulting list of action select one, which the agent can perform, but the opponent cannot, and which has minimal cost. This action will be offered as a future reward in return to executing requested action right now.

A threat to perform an action that contradicts with opponent's plans in case if the requested action will not be executed can also be a good argument. An algorithm for calculating it includes: find one opponent's desire that is not in agent's desire set, first consider desires with higher preference; find a contradicting action to the desire or like in "appeal to self-interest" find a list of actions needed to satisfy the desire and find an action that undoes effects of one of those actions. This action will then be selected as a threat argument in case if a requested action will not be executed.

An agent can generate several arguments at the same time, but only one of them can be used for every negotiation round. To be able to identify which argument should be used an argument selection rule is required. Kraus et al. [26] proposed to use the argument types in the same order as they were introduced earlier in this subchapter. In this case the weakest argument is selected first and if it will not succeed, then the stronger argument is taken.

Request evaluation rules define how the request is being processed by the agent when it receives one. Request evaluation should end with a response message back to the sender stating either that the argument is accepted and the agent will perform the prescribed action, or that the argument did not persuade the agent to fulfill the request. Also as it was mentioned above during the request evaluation agent's beliefs, desires, intensions or goals can be changed. As an example of such changes in our project can be saboteur switching to zone defending after negotiation with other saboteurs: the beliefs about the zone he has to defend are added and the primary goal is changed to zone defending. Another example is an agent adopting a role of zone coach after negotiation about the best zone: the goal to invite other agents to the newly created zone, regularly check for enemy agents near the zone and so on. Request evaluation always depends on the arguments that are used, the agents participating in the negotiation and the request itself.

For the implementation of negotiation procedures and making collective decisions in our project we mostly used the *bidding* method described in [9]. In this method all the participating in negotiation agents are sending their "bids" to the other agents. These bids contain the appropriate arguments for the current negotiation target. Every agent waits for bids from all other agents and after that performs a comparison of bids: every bid is compared with all other bids. The comparison of two bids includes argument selection and request evaluation at the same time: the arguments are selected one by one in each of two bids and compared until one of the arguments prevail. For the conflict situations when all arguments appear to be the same we assumed that the bid that came from the greatest agent in terms of agent name comparison wins. The agent with the winning bid then fulfills the request: adopts a certain role or performs a prescribed action.

2.2.4 Agent Societies.

add, adapt and improve Rahul's part if it fits and is helpful for our later work

2.3 Agent Programming Languages°

We investigated several agent programming languages against their suitability for the Mars-scenario. They were proposed by our supervisors. Our goal was to determine what language we wanted to use for multi-agent programming if any. The following sections present the basic structure of various languages together with examples. These examples are in no relation to the Mars-scenario but are simplified to a minimum to ease understanding. Using the Mars-scenario for examples instead would have meant to either make them complex or to trivialise them to a point where they become too superficial to suit the scenario. Section 2.3.1 first introduces the situation calculus. Although not an agent programming language, it serves as a foundation of the logic programming language GOLOG presented in Section 2.3.2. It also helps understanding the subsequent Section 2.3.3 which summarises the main concepts of FLUX. FLUX is another logic programming language which was partly motivated from the flaws of GOLOG. Section 2.3.4 introduces a Java-based agent programming language. After that, AgentSpeak(L) is presented in Section 2.3.5 which is another logic programming language. Jason is an interpreter for this language and is discussed in Section 2.3.6. The section focusses mainly on the extensions that Jason adds to AgentSpeak(L). The final Section 2.3.7 summarises the previous sections and explains our decision for choosing Jason.

2.3.1 Situation Calculus.°

This section gives a short summary of the situation calculus, which was first introduced by McCarthy and Hayes [33]. The situation calculus is mainly a first-order logic but also uses second order logic to encode a dynamic world [29]. It is a theoretic concept and was consequently not under consideration to be used as an agent programming language to compete in the MAPC. Yet, it is being presented to serve as basis for the later illustrated languages GOLOG and FLUX. The situation calculus consists of the three first-order terms: *fluents*, *actions* and *situations* [33, 11]. Fluents model properties of the world. Actions may change fluents and hence may modify the world. Every action execution creates a new situation. This is because a situation is a history of actions up to a certain point in time starting from the initial situation s_0 [44, 29]. There can only be one initial situation as it models the situation before any action has been executed [36].

Fluents can be evaluated to return a result. As they are situation dependent, the evaluation result may change over time. Fluents are distinguished in *relational fluents* and *functional fluents* [29]. Relational fluents can hold in situations. Their evaluation hence may return either true or false [11]. An example is given in Equation 1. It expresses whether or not the agent p has a coffee in situation s.

$$hasCoffee(p, s) \tag{1}$$

Functional fluents return values instead [29]. As an example, a fluent $location(p, s)$ may return some coordinates (x, y) This then expresses the agent p's location in situation s.

Actions also depend on situations. The reason for this is that certain actions may only be executed when specific fluents hold. As fluents are only modified by actions, their result can be determined by the history of action executions contained in the current situation. Describing when an action is executable is done by *action precondition axioms* [30]. This is expressed by the predicate $Poss(a, s)$ with a being an action. As a recurring example, let us think of the ability to pour an agent p coffee. This must only be possible when p does not already have coffee. Equation 2 illustrates how this can be formalised.

$$Poss(pourCoffee(p), s) \Leftrightarrow \neg hasCoffee(p, s) \tag{2}$$

As mentioned before, the execution of any action must alter the situation: $do(a, s) \rightarrow s'$. Its effects on fluents are described by *action effect axioms*. Equation 3 shows how pouring a coffee to p will result in p having coffee afterwards.

$$Poss(pourCoffee(p), s) \rightarrow hasCoffee\big(p, do(pourCoffee(p), s)\big) \tag{3}$$

In Equation 3, it is unclear whether other fluents are affected by the action execution. For example, reasoning about $location(p, s')$ would not be possible with $do(pourCoffee(p, s)) \rightarrow s'$. This is called the *frame problem* (cf. Hayes [24]). Defining for every fluent how every action does or does not affect it is only a theoretical solution. The reason for that is that the resulting complexity of $\mathcal{O}(A * F)$ would be too high even in most small worlds. A feasible solution to this problem was proposed by Reiter [41]. His approach was to define every effect of all actions only once. Thus, Reiter reduced the complexity to $\mathcal{O}(A * E)$. This solution is known as the *successor state axiom* and is shown in Equation 4.

$$Poss(a, s) \rightarrow \big[F(do(a, s)) \Leftrightarrow \gamma_F^+(a, s) \vee F(s) \wedge \neg\gamma_F^-(a, s)\big] \tag{4}$$

$F(do(a,s))$ means that the fluent F will be true after executing the action a. The first part of the disjunction is $\gamma_F^+(a,s)$ and expresses that the action made the fluent true. $F(s) \wedge \neg\gamma_F^-(a,s)$ as the second part expresses that the fluent had been true before and the action had no influence on it. For a reasonable example, there needs to be a second action which does not influence the fluent given in Equation 1. Therefore, the $sing(s)$ action will be introduced which has no effect on any fluents and can be executed anytime as shown in Equation 5.

$$Poss(sing, s) \Leftrightarrow \top \tag{5}$$

Given Equation 1, 2, 3 and 5 an example can be compiled as done in Equation 6:

$$\begin{aligned} \text{Poss}(a,s) \rightarrow \big[&\text{hasCoffee}(p, \text{do}(a,s)) \\ &\Leftrightarrow [a = \text{pourCoffee}(p)] \\ &\vee \ [\text{hasCoffee}(p,s) \wedge a \neq \text{pourCoffee}(p)]\big] \end{aligned} \tag{6}$$

Equation 6 then formalises that an agent p may only have coffee if it was poured coffee or if it already had coffee and the action was not to pour p a coffee.

2.3.2 GOLOG.°

This section gives a summary of the logic programming language GOLOG. Moreover, its problems in context of the Mars-scenario are shown. If not further specified, all information except for the examples is taken from Levesque et al. [29] who introduced the language. GOLOG builds on the situation calculus. To allow high-level programming, the language adds complex actions like loops, conditions, tests and non-deterministic elements. As an example, a GOLOG program should have a robot pouring other agents coffee until everybody does have coffee. After that, the robot should sing and terminate. Such a program would reuse the fluent of Equation 1, the action precondition axioms of Equation 2 and 5, the successor state axiom of Equation 6 and extend them with the two procedures given in Equation 7 and 8:

$$\begin{aligned} &\textbf{proc}\ \texttt{main}\ [\textbf{while}\ (\exists p)\neg \mathit{hasCoffee}(p) \\ &\qquad \textbf{do}\ \texttt{pourSOCoffee}(p)\ \textbf{endWhile}]; \\ &\quad \mathit{sing}\ \textbf{endProc}. \end{aligned} \tag{7}$$

$$\begin{aligned} &\textbf{proc}\ \texttt{pourSOCoffee}\ (\boldsymbol{\pi} p)\ [\neg \mathit{hasCoffee}(p)\textbf{?}; \\ &\qquad \mathit{pourCoffee}(p)]\ \textbf{endProc}. \end{aligned} \tag{8}$$

Equation 7 shows the procedure which can be seen as the main method. It loops as long as there exist agents without coffee and tells the robot to pour some agent coffee which is lacking coffee. In the end, the robot sings. Equation 8 allows the robot to non-deterministically choose an agent p to pour coffee by using the π-operator. The ?-operator is similar to the `if`-operator in other programming languages like Java. Due to the non-determinsmic operator, there can be two different resulting situations as shown in Equation 10 with the initial configuration given in Equation 9:

$$\neg hasCoffee(p, s_0) \Leftrightarrow p = \text{Jane} \vee p = \text{John}. \tag{9}$$

$$\begin{aligned} s &= do\Big(sing, do\big(pourCoffee(\text{Jane}), do(pourCoffee(\text{John}), s_0)\big)\Big), \\ s &= do\Big(sing, do\big(pourCoffee(\text{John}), do(pourCoffee(\text{Jane}), s_0)\big)\Big) \end{aligned} \tag{10}$$

Levesque et al. [29] highlight multiple problems with GOLOG. Problems which are relevant when considering to apply GOLOG on the Mars-scneario, are given in this part. One problem is that complete knowledge is assumed in the initial situation. This is not the case for the Mars-scenario and scenarios with unknown worlds that get explored by agents in general.

The second problem is that GOLOG does not offer a simple solution for sensing actions and reactions of agents on sensed actions. Sensing actions are actions by agents that may not modify fluents but the internal knowledge of agents by detecting some properties in the world [47]. This can be seen as a side-effect of GOLOG not being developed for unknown worlds. Again, this would be a feature which is needed for the Mars-scenario.

A third problem is that exogenous actions cannot be handled. Exogenous actions are actions outside of the agent's control. In the Mars-scenario, this e.g. could be the loss of an agent's health due to an enemy agent attacking it.

Thielscher [47] highlights a fourth problem. It arises from GOLOG being *regression-based*. This means that deciding whether an action is executable is only possible after looking at all previous actions and how they might have affected the world. As a result, reasoning takes exponentially longer over time and hence GOLOG does not scale. Due to these problems, GOLOG is unsuitable for a multiple agent-based scenario like the Mars-scenario without considerable modifications and extensions.

2.3.3 FLUX.°

This section gives a summary of the logic programming language FLUX which offers solutions to the earlier shown problems of GOLOG. Except for the examples and if not specified otherwise, the information of this section is taken from Thielscher [47] who first introduced FLUX. This is done by using the *fluent calculus* instead of the situation calculus. Both are similar but the fluent calculus adds *states*. A state z is a set of fluents $f_1, \ldots, f_n$. In FLUX, it is denoted as $z = f_1 \circ \ldots \circ f_n$. In every situation there exists exactly one state with which the current properties of the world are being described. Yet, the world can be in the same state in multiple situations. FLUX uses *knowledge states* for representing agent knowledge. These are denoted through $KState(s, z)$ meaning that an agent knows that z holds in s. Knowledge states can be incomplete as opposed to knowledge in GOLOG.

The frame problem in the fluent calculus is solved through *state update axioms* as described by Thielscher [48]. The axioms define the effects of an action as the difference between the state before and after the action. This is modelled with ϑ^- for negative and ϑ^+ for positive effects. Both are simply macros for finite states. Due to using states, reasoning is linear in the size of the state representation. That is, after every action execution, the world represented by its fluent is processed. This is called being *progression-based.* Therefore, FLUX can outperform GOLOG as determining whether a property currently holds is only a matter of looking it up in the state. With GOLOG however, the property must be traced back to the initial situation by looking at all action executions and their effects.

Disjunctive and negative state knowledge is modelled through constraints. FLUX uses a constraint solver to simplify these constraints until they are solvable. This is done by using *constraint handling rules* introduced by Frühwirth [21]. Their general form is shown in Equation 11. It consists of one or multiple heads H_m, zero or more guards G_k and one or multiple bodies B_n. The general mechanism is that if the guard can be derived, parts of the constraint matching the head will be replaced by the body and hence get simplified.

$$H_1, \dots, H_m \Leftrightarrow G_1, \dots, G_k \mid B_1, \dots, B_n \tag{11}$$

A FLUX program can be separated into three main parts with the constraint solver building the kernel which is the foundation of a FLUX program. The domain encodings are built on top of this. Included are the initial knowledge state(s), domain constraints, as well as the action precondition and state update axioms. The final part of a FLUX program is the programmer defined intended agent behaviour called strategy. As a trivial example program, the previous example implemented in GOLOG will be transferred into FLUX. This is done by using the logic programming language Prolog in which FLUX is typically implemented (cf. [49, 32]). The example features the domain encodings as well as the strategy.

```
1 perform(sing, []).
2 poss(sing, Z) :- all_holds(hasCoffee(_), Z).
3 state_update(Z, sing, Z, []).
```

Listing 1.1: Defintion of the `sing`-action.

Listing 1.1 shows the definition of the `sing`-action. Empty arrays denoted by `[]` could be replaced by sensed information. They would then effect the outcome of the methods. As this is a trivial example, no sensed information is assumed. Line 2 is the precondition that singing is only possible in a state where every agent has coffee. As singing should not alter any fluents, the state `Z` in Line 3 is not modified and returned again as `Z`.

```
4 perform(pourCoffee(P), []).
5 poss(pourCoffee(P), Z) :-
6       member(P,[jane,john]),
7       not_holds(hasCoffee(P), Z).
8 state_update(Z1, pourCoffee(P), Z2, []) :-
```

```
9         update(Z1, [hasCoffee(P)], [], Z2).
```

Listing 1.2: Definition of the `pourCoffee`-action

The `pourCoffee`-action is defined similarly in Listing 1.2. Line 6 ensures that Prolog will only look for agents that actually exist instead of iterating over memory addresses. The action must modify the state by adding `hasCoffee(P)` to the state as it is done in Line 9. The array after it corresponds to ϑ^{-}. It is empty in this case as no fluents are removed.

```
10  main_loop(Z) :-
11    poss(sing, Z)
12      -> execute(sing, Z, Z);
13    poss(pourCoffee(P), Z)
14      -> execute(pourCoffee(P), Z, Z1),
15         main_loop(Z1);
16    false.
```

Listing 1.3: Main method which either tells the robot to sing or to pour coffee.

Listing 1.3 models the main method and thus is similar to Equation 7. When singing is possible, the robot will do so and terminate. Else, it will pour someone a coffee and call the main loop again. Line 16 ensures that Prolog will return the false-value `No` if neither of the both actions get triggered at some point.

```
17  init(Z0) :-
18         not_holds(hasCoffee(jane), Z0),
19         not_holds(hasCoffee(john), Z0).
```

Listing 1.4: Initial configuration.

The initial configuration in Listing 1.4 is comparable to Equation 9 but due to Prolog interpreting from top to bottom, the result will be `Z = [hasCoffee(john), hasCoffee(jane)]`.

Schiffel and Thielscher [43] successfully applied FLUX to the gold mining domain. It is a scenario where multiple agents with different roles work together on mining gold in an unknown terrain [43]. The requirements for solving the problems arising from this scenario are comparable to those appearing in the Mars-scenario. Given the former short presentation and this knowledge, it can be said that FLUX could be applied to the Mars-scenario.

2.3.4 Jadex.$^{\odot\circ}$

Nowadays a couple of agent frameworks are available for developing multi-agent applications. An overview of existing tools and techniques is given by the European co-ordination action for agent-based computing, namly AgentLink [31]. This section presents Jadex, which is an agent framework focused on the development of goal-oriented agents following the belief-desire-intention model. It aims at bringing middleware and reasoning-centred agent platforms together. For that purpose, Jadex adds a rational reasoning engine to existing middlewares.

The most commonly used middleware for Jadex is the *Java Agent Development Framework* (short: JADE) [4]. Jadex integrates agent-theories through object-oriented programming in Java and XML descriptions. Therefore, no new language is introduced. Jadex reuses already existing technologies instead. JADE provides a communication infrastructure, platform services such as agent management and a set of development and debugging tools. It enables the development and execution of peer-to-peer applications which are based on the agent paradigm (autonomous, proactive and social). Agents are identified by a unique name and provide a set of services. They can register and modify their services and/or search for agents providing given services. Additionally they are capable of controlling their life cycle and they can dynamically discover other agents and communicate with them The communication happens by exchanging asynchronous messages via an *agent communication language* (short: ACL). Jadex complies with the standard given by the Foundation for Intelligent Physical Agents (FIPA). FIPA "is an international organszation that is dedicated to promoting the industry of intelligent agents by openly developing specifications supporting interoperability among agents and agent-based applications."[16]. A FIPA ACL message has a certain structure and parameters. Mandatory parameters are the type of the communicative act, the participants in the communication, the content of the message, the description of the content and the control of the conversation.

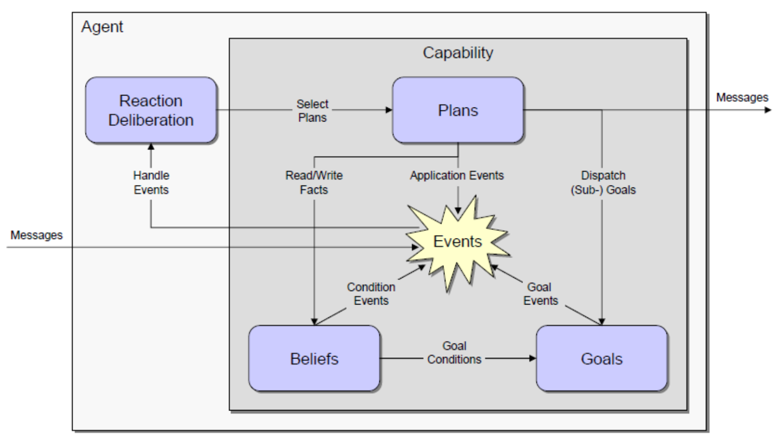


Fig. 3: Jadex abstract agent [37]

Figure 3 depicts an abstract view on a Jadex agent. Every agent may receive messages which trigger internal events that can change his internal knowledge, plans or goals. Interactions with the outside like the environment or other agents happens through the sending of messages. In more detail, beliefs are single facts stored as Java objects which represent the knowledge of an agent. They are stored as key-value pairs. The advantage of storing information as facts is that the programmer has a central place for the knowledge and can query the agent's beliefs. Monitoring of the beliefs is possible too.

The goals are momentary desires of an agent for which the agent engages into suitable actions until it considers the goal as being reached, unreachable, or not wanted any more. Referring to [13], Jadex distinguishes between four generic goal types. A perform goal is directly related to the execution of actions. Therefore, the goal is considered to be reached, when some actions have been executed, regardless of the outcome of these actions. An achieve goal is a goal in the traditional sense, which defines a desired world state without specifying how to reach it. Agents may try several different alternative plans, to achieve a goal of this type. A query goal is similar to an achieve goal, but the desired state is not a state of the (outside) world, but in internal state of the agent, regarding the availability of some information the agent wants to know about. For goals of type maintain an agents keep track of a desired state, and will continuously execute appropriate plans to re-establish this maintained state whenever needed. In contrast to goals, events are (per default) dispatched to all interested plans but do not support any BDI-mechanism. Therefore, the originator of an internal event is usually not interested in the effect the internal event may produce but only wants to inform some interested parties about some occurrence. Plans represent the behavioural elements of an agent and are composed of a head and a body part. The plan head specification is similar to other BDI systems and mainly specifies the circumstances under which a plan may be selected, e.g. by stating events or goals handled by the plan and preconditions for the execution of the plan. Additionally, in the plan head a context condition can be stated that must be true for the plan to continue executing. The plan body provides a predefined course of action, given in a procedural language. This course of action is to be executed by the agent, when the plan is selected for execution, and may contain actions provided by the system API, such as sending messages, manipulating beliefs, or creating sub-goals (cf. [15])

Jadex is not based on a new agent programming language. Instead, a hybrid approach is chosen, distinguishing explicitly between the language used for static agent type specification and the language for defining the dynamic agent behaviour. An agent in Jadex consists of two components: An *agent definition file* (short: ADF) for the specification of beliefs, goals, and plans as well as their initial values and on the other hand procedural plan code. The procedural part of plans (the plan bodies) are realised in an ordinary programming language (Java) and have access to the BDI facilities of an agent through an application interface (API). The plan body is a standard Java class that extends a predefined Jadex framework class and has at least to implement the abstract body() method which is invoked after plan instantiation.

```
public class ServeCoffeePlanB1 extends Plan {
   // Plan attributes.

   public ServeCoffeePlanB1() {
       // Initialisation code.
   }
```

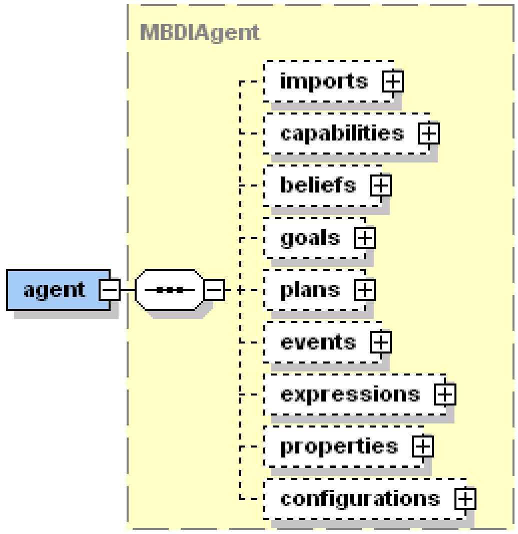


Fig. 4: Jadex top level ADF elements [14]

```
    public void body() {
        // Plan code.
    }
}
```

The plan body is associated to a plan head in the ADF. This means that in the plan head several properties of the plan can be specified, e.g. the circumstances under which it is activated and its importance in relation to other plans.

```
<agent xmlns="http://jadex.sourceforge.net/jadex-bdi"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://jadex.sourceforge.net/jadex-bdi
            http://jadex.sourceforge.net/jadex-bdi-2.0.xsd"
 name="CoffeeAgent">

 <plans>
   <plan name="serve">
     <body class="ServeCoffeePlanB1"/>
     <waitqueue>
       <messageevent ref="request_serving"/>
     </waitqueue>
   </plan>
 </plans>

 <events>
   <messageevent name="request_serving"
                 direction="receive" type="fipa">
```

```
      <parameter name="performative" class="String"
                 direction="fixed">
        <value>jadex.bridge.fipa.SFipa.REQUEST</value>
      </parameter>
    </messageevent>
  </events>

  <properties>
    <property name="debugging">false</property>
  </properties>

  <configurations>
    <configuration name="default">
      <plans>
        <initialplan ref="serve"/>
      </plans>
    </configuration>
  </configurations>
</agent>
```

There are two types of plans in Jadex. A *service plan* and a *passive plan*. The service plan, as the name indicates, is an instance of a plan which waits for service requests. Therefore a service plan can set up its private event wait queue and receive events for later processing, even when it is working at the moment. In contrast to that, a passive plan is only running when it has a task to achieve. For this kind of plan the triggering event and goals must be specified must be specified in the agent definition file to let the agent know what kinds of events this plan can handle. When an agent receives an event, the BDI reasoning engine builds up the so called applicable plan list which contains all plans that can handle the current event or goal. The candidates are selected and instantiated for execution.

The execution model for Jadex looks like the following:
When an agent receives a message it is placed at a message queue. In the next step the message has to be assigned to a capability, which can handle the message. A suitable capability is found by matching the message against the event templates defined in the event base of each capability. The best matching template is then used to create an appropriate event in the scope of the capability. After that the created event is subsequently added to the agent's global event list. The dispatcher is responsible for selecting applicable plans for the events from the event list. After plans have been selected, they are placed in the ready list, waiting for execution. The execution of plans is performed by a scheduler, which selects plans from the ready list.[37]

All in all Jadex is a powerful framework that supports easy agent construction with XML-based agent description and procedural plans in Java. Additionally, it offers tool support for development debugging. It comes for example with a BDI-Viewer that allows observing and modifying the internal state of an agent and a logger agent that collects log-outputs of any agent. Judging from this

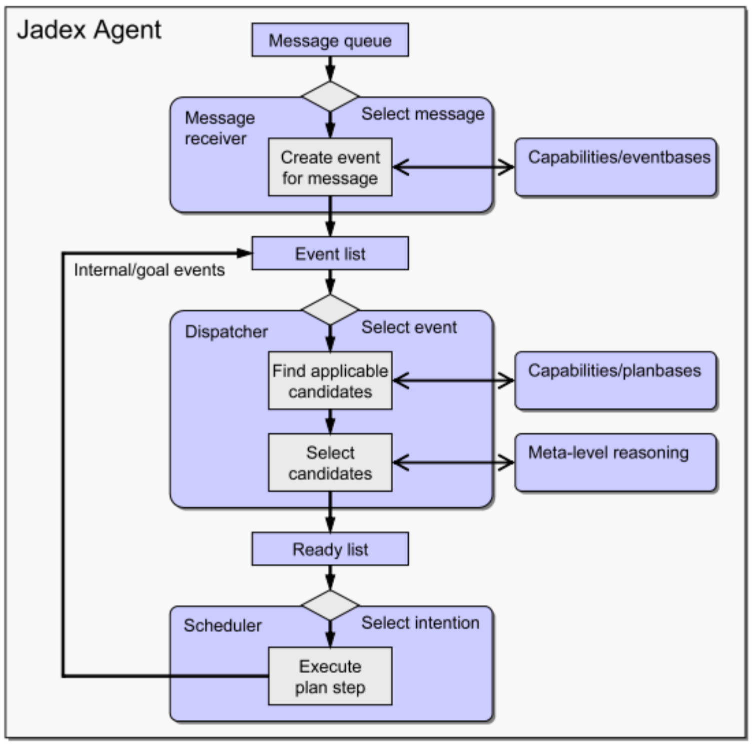


Fig. 5: Jadex execution model [37]

knowledge, Jadex seems suitable for the purpose of competing in the multi-agent programming contest.

2.3.5 AgentSpeak(L).°

This section gives an overview of the general concepts of the logic programming language AgentSpeak(L). The language was developed by Rao [39]. Except for the examples, this section takes its information from the given paper. The idea behind AgentSpeak(L) was to make the theoretic concept of BDI-agents usable in practical scenarios.

The main language constructs are *beliefs*, *goals* and *plans*. Beliefs represent information that an agent has about its environment. A belief `hasCoffee(p)` for example denotes that an agent knows that the person `p` has coffee. In AgentSpeak(L), variables are indicated by using a capital first letter whereas terms with a small first letter are constants.

```
~hasCoffee(jane).
~hasCoffee(john).
```

Listing 1.5: Initial beliefs.

Listing 1.5 shows the initial beliefs an agent has for our earlier introduced example. The tilde expresses that the agent knows that neither `john` nor `jane` has coffee. At any given time, the sum of all current beliefs of one agent are called its *belief base* [8].

Goals can be dividied into *achievement goals* and *test goals*. The first expresses the wish of an agent to reach a state where a belief holds where the second tests whether a belief holds in the current state. Beliefs hold when the agent knows they are true or when the variables can be bound to at least one known configuration. For example, given an achievement goal `!hasCoffee(p)` means that an agent wants to achieve that person `p` has coffee. Similarly, `?hasCoffee(p)` expresses that an agent tests whether `p` has a coffee. Hence, this expression will evaluate to true or false depending on the current agent's knowledge. Achievement goals are comparable to desires. Listing 1.6 shows the initial achievement goal which express that the agent wants to have served everyone coffee.

```
!servedCoffee.
```

Listing 1.6: Initial goal.

Events are introduced to allow agents to react on changes in their own knowledge or the world. They can be distinguished into the addition and removal of beliefs or goals. Additions are denoted by a plus- and removals by using a minus-sign in front of the goal or belief:

- `+hasCoffee(p)`: an agent is informed that `p` now has coffee.
- `-hasCoffee(p)`: an agent is informed that `p` no longer has coffee.
- `+!hasCoffee(p)`: an agent is informed that it wants `p` to have coffee.
- `-!hasCoffee(p)`: an agent is informed that it no longer wants `p` to have coffee.

– `+?hasCoffee(p)`: an agent is informed that it should test for the belief.
– `-?hasCoffee(p)`: an agent is informed that it no longer needs to test for the belief.

In order to handle new events, an agent will look for a matching plan.

Plans can be seen as programmer-defined agent instructions. They lead to the execution of actions or the splitting of goals into additional goals. Plans, which an agent wants to execute, are similar to what are called intentions for BDI-agents. The set of plans known to an agent are called the *plan library* [8]. A plan is triggered by events and is context-sensitive. This means that the execution of a plan can be restricted to states in where certain beliefs exist. Listing 1.7 illustrates this by showing when the `sing`-action is being executed. Line 4 is the triggering event of the plan. In this case, an agent will consider executing this plan, when it notices that someone is poured coffee. Hence, this plan is called a *relevant plan*. The underscore denotes an anonymous variable similar to its use in Prolog. Its meaning is that it will match any term. Line 5 is the plan's context. The plan is called an *applicable plan* if the context's beliefs are all known to the agent. In this particular case, the agent must know that there is no person without coffee indicated by the use of the tilde. At last, Line 6 contains the body of the plan. Here, the agent should achieve the goal `sing`. This will trigger a new event which calls the plan in Line 9. As its context is empty, the plan can be executed immediately and evaluates to true as there is no body. Line 7 expresses how the event of someone being poured coffee should be alternatively handled. As AgentSpeak(L) is interpreted from top to bottom, it will only be seen as an applicable plan, if the former relevant plan did not trigger. Therefore, if the agent knew that there was still someone left without coffee, it will want to achieve the `servedCoffee` goal again.

```
4  +hasCoffee(_):
5      ~hasCoffee(_)
6      <- !sing.
7  +hasCoffee(_)
8      <- !servedCoffee.
9  +!sing.
```

Listing 1.7: Events for handling someone being poured a coffee as well as the `sing` plan.

Listing 1.8 contains the plan for serving coffee. It uses a test goal to pick someone without a coffee as shown in Line 11. The person will be bound to the variable `X`. After that, an achievement goal is added to the agent's set of intentionsto introduce the set of intentions earlier pour `X` coffee. The plan does not feature any context as this minimal example ensures that the goal `!servedCoffee` will only exist when there actually is a person without coffee.

```
10  +!servedCoffee:
11      <- ?~hasCoffee(X);
12         !pourCoffee(X).
```

Listing 1.8: Definition of the `servedCoffee` plan.

Listing 1.9 shows a plan which states that if an agent receives an event to achieve the goal `!pourCoffee` for some person `X`, it will pour coffee to `X`. Additionally, the knowledge about `X` not having any coffee is removed in Line 16.

```
14 +!pourCoffee(X)
15     <- +hasCoffee(X);
16        -~hasCoffee(X).
```

Listing 1.9: Definition of the `pourCoffee` plan.

AgentSpeak(L) is suitable for multi-agent scenarios as the Mars scenario. Especially when comparing FLUX to the components of AgentSpeak(L) plans, it becomes clear that there are many similarities. A FLUX's action's precondition is similar to a plan's context and the state update axiom is implicitly included in a plan's body. Yet, the state in AgentSpeak(L) does not have to be manually and explicitly modified. Furthermore, a plan's body enables further possibilities like adding new goals. The main difference between FLUX and AgentSpeak(L) is that FLUX is based on fluent calculus. It is hence a more general approach focussing on modelling the change of fluents. AgentSpeak(L) on the other hand was strictly developed as an application for BDI-agents.

2.3.6 Jason.°

This section gives a quick overview of Jason, which is an interpreter for AgentSpeak(L). All information if not marked differently is taken from Bordini et al. [8]. Besides being an interpreter, Jason extends AgentSpeak(L) by several concepts. The most important ones will be discussed in this section.

With Jason, terms can represent more than a constant or a variable. They can be strings, integer or floating point numbers or lists of terms. Therefore, more complex programmatic operations and arithmetic expressions are possible with Jason. Furthermore, Jason introduces annotations. With these annotations, metadata can be added to triggering events and beliefs. This metadata can be accessed programmatically. Listing 1.10 shows the earlier used initial beliefs with added annotations. The `source` annotation is the only one with its meaning predefined by Jason. It expresses the source of the information. If an agent determined something itself, the `source` is `self`. Did the agent receive the information as a perception of the environment, then the `source` will be `percept`. The source can also be a constant identifying a different agent if that agent is the source of this information. With the example given in Listing 1.10, an achievement goal `?~hasCoffee(X)[reliability(Y)]` will bind `X` to `john` and `Y` to `0.3`. The `reliability` has no further meaning unless the value bound to `Y` is used later.

```
~hasCoffee(jane)[source(self)].
~hasCoffee(john)[source(percept), reliability(0.3)].
```

Listing 1.10: Annotation of beliefs in Jason.

Another concept added to AgentSpeak(L) by Jason is called *internal actions*. It was first introduced and implemented by Bordini et al. [10]. Most characteristic for these actions is that they do not affect the environment in which the agents are located in. This means they have no effect on the external world but only on the internal states of the agents as the name suggests. Hence, any effects of internal actions occur immediately after the action execution instead of only after the next environment processing cycle. As a result, internal actions can not only be used within a plan's body but also in its context. Internal actions start with a dot followed by a library identifier, another dot and finally the action name. Bordini et al. [10] implemented various internal actions which are not identified by any explicitly named library. These methods reside in the so called *standard library* and omit the library declaration when being called. An example for this is `.gte(X,Y)` which returns the truth value of `X≥Y`. A realisation of the same function outside the standard library could e.g. be called `.math.gte(X,Y)`. The standard library is included in Jason. Furthermore, Jason extends this library by various actions including multiple list operations like sorting or retrieving the minimum. Developers can write additional internal actions in Java or any other programming language which supports the programming framework Java Native Interface.

Arguably, the most important internal action is `.send`. This action enables inter-agent communication as initially proposed and implemented by Vierira et al. [50]. It is similar to KQML performatives which had been introduced in SECTIONXYZ.

if we talked about KQML earlier, reference it

```
1  .send(Receiver, Illocutionary_force, Message_content).
2  .send([agent1, agent2], tell, ~hasCoffee(john)).
```

Listing 1.11: Parameters of the internal action `.send` and an example.

In Line 1 of Listing 1.11 the structure of the `.send` action is shown. Line 2 shows example usage of this action. The `Receiver` is the identifying name or a list of identifying names for the agent(s) to which the message should be addressed to. The `Illocutionary_force` is a constant that specifies what all recipients should do with the message. It can be:

- `tell`: add the `Message_content` to the recipient's belief base.
- `untell`: remove the `Message_content` from the recipient's belief base.
- `achieve`: add the `Message_content` as an achievement goal to the recipient.
- `unachieve`: make the recipient remove the achievement goal `Message_content`.
- `tellHow`: `Message_content` is added to the recipient's plan library.
- `untellHow`: `Message_content` is removed from the recipient's plan library.
- `askIf`: asks if `Message_content` is in the recipient's belief base.
- `askOne`: asks for the first belief matching `Message_content`.
- `askAll`: asks for all beliefs matching `Message_content`.
- `askHow`: demand all plans a recipient has that match the triggering event given in the `Message_content`.

Jason automatically processes the messages as needed when a message arrives at an agent. A developer can override Jason's default behaviour if further or

should we explicitly talk about an agent's inbox? Compare with @adaudrich's mentioning of the inbox!

different processing is desired. Jason also automatically adds `source` annotations. This allows agents to determine the sender of any received message.

There is special support for defining environments with Jason. Instead of having to do this in AgentSpeak(L), it can be done in Java. For doing so, a developer has to extend the `Environment` class and specify the `getPercepts(String agentName)` and `executeAction(String agentName, Term action)` methods. The first method must return a list of literals restricted to what the agent identified by `agentName` can perceive. When the second method is called, the programmer must specify how the given `action` affects the environment. It returns a boolean to indicate whether the execution was successful. Such an action can fail if for example a repairer agent would try to execute the `attack` action which it cannot according to the Mars-scenario. To call the `executeAction` method from an agent, all it has to do is execute e.g. `attack`. Jason will then call `executeAction(String agentName, Term action)` with the parameters bound to the agent's name and the `attack` action. For the MAPC itself, no fully simulated environment is needed. Instead, it is enough to delegate the actions to the MAPC server and process the server replies by returning the transmitted percepts to the respective agents. Therefore, percepts do not have to be modelled or modified in the environment developed with Jason itself.

Jason also allows running multi-agent systems over networks in a distributed manner. Hence, the workload can be distributed over multiple machines. SACI [25] and JADE are the two fully implemented distributed architectures usable out of the box with Jason [9]. Fernández et al. [20] could not prove the intended performance benefits. The authors tested both SACI and JADE with Jason where one host would run the environment and the other one the agents. They increased both the amount of agents as well as the size of the environment. Fernández et al. [20] saw that with increasing complexity, the system became slower compared to when agents and the environment were run on a single machine. This was due to the added communication cost between the two hosts although connected by Gigabit Ethernet. As a result, a distributed infrastructure with Jason is only advisable, if the workload cannot be handled by one host alone. In our case, replying in time has such an importance that trying to keeping the workload processable by one host alone would be the preferred strategy.

2.3.7 Choice of a programming language.$^{\circ/\odot}$

Based on the previous sections, this section summarises why we chose Jason for developing our agents. Generally, we could have started from scratch without using a designated agent programming language. We decided against this idea because of our inexperience with agent programming and artificial intelligence in general. The fear was to overlook difficulties in the beginning which would later force us to spend more time on fixing mistakes we made in the beginning than on the actual agent development. To prevent this, we were interested in using an already developed and approved agent programming language.

Given the Mars scenario, Jason can be used to implement a suitable multi-agent system. In fact, two teams successfully participated in the 2013 Multi-

Agent Programming Contest by using Jason [1]. Yet, there was no competing team using Jadex or FLUX. This is of interest because the scenario of 2013 is comparable to the scenario of 2014 [2]. As the whole team was inexperienced with logical programming prior to this research lab, being able to develop the environment and some operations via internal actions in Java was beneficial. Furthermore, the contest organisers provided a Java library which would simplify the communication with their server. Instead of having to manually compile XML messages and parse the XML server replies, this library allowed simple method calls for server interaction. Thus, deciding against FLUX meant not having to implement the communication with the server ourselves. The library would also have been usable with Jadex. But just like Flux, Jadex does not assist the developer in modelling the environment like Jason does. Jason's support for environments allowed us to focus more on agent programming There, we preferred Jason and Jadex over FLUX, because these two languages are built around BDI, which we found to be a clearer structuring of agents. FLUX on the other hand serves as a quite generic approach to programming multi-agent systems. Besides the support for developing environments, Jason and Jadex are also different in the way how the initial beliefs, goals and plans are being programmed.

integrate the text below:

The difference lies in the storing of beliefs, goals and plans. In Jadex they are stored in the agent definition file (XML) while in Jason they are stored as facts within the Jason belief base. Another slight difference is that in Jadex plans have to be written in Java whereas in Jason the programmer can use a combination of Java and AgentSpeak with internal actions. We didn't chose Jadex for our research lab because of the overhead that comes with the XML-syntax.

3 Team Organisation and Collaboration Tools$^{\odot/\circ}$

The topic of this section is the team structure as well as the software we used for working together. In the first part of this section, the organisation of the team is shown. It focuses on the distribution of tasks and explains how we worked together. The second part presents what software tools we have used for working together. It also remotely discusses the usefulness of the Jason plugin for our tasks.

At the beginning of the project the team had to define a structure for collaboration. We decided to have a flat hierarchy with all members as part of dynamically built, small development teams. Michael Ruster was selected as a project leader with his role mainly focussed on organisation. He carried out the external communication with the supervisors and contest organisers. The project leader was also responsible for configuring the server and running test simulations together with the organisers of the MAPC as well as running the final simulations. His leadership followed a democratic management style, so decisions were made by the team as a whole through majority decisions. Once in every week, the team met in person to discuss the current progress and the upcoming course of actions. All meetings were recorded by a minute taker. The minutes logged the attendees, open issues from last meeting, the decisions made in the current meeting as well as a list of assigned tasks to work on until the next meeting. We did not have a designated minute taker but would rotate alphabetically by surname. The person to take the minutes was also the one to present our progress at our weekly meetings with our supervisors. During the weekly team meetings, many of our algorithms were initially developed and discussed. At the end of each meeting, the worked out tasks for the next meeting were assigned to dynamic groups. These groups mainly consisted of two or three people with more people working on tasks we found to be more important. Team members were assigned for a specific task due to personal interest or expertise. In the beginning, we also tried some hacking sessions, but quickly found out that working from home worked best for us. This was advantageous because no fixed timeslots needed to be found. Instead, everybody would work independently when they found the time while staying in contact with the others through chat or voice over IP. The possibility to share the computer screen contents over IP offered by the voice over IP solutions we used, was of great help. Therefore, multiple persons could work on the same code at once with one programming and the others reviewing it real-time. If there was need for reconciliation, for example when tasks of different groups were closely interrelated or dependant on one another, short-termed voice over IP calls were held. To the end of development, the groups diverged mainly into Artur Daudrich and Michael Sewell working on the Java-side of our code and the rest focussing on implementations in AgentSpeak(L). The prior group hence concentrated on implementing background calculations like internally modelling and constructing the graph and environment design. Accordingly, Manuel Mittler, Michael Ruster, Sergey Dedukh and Yuan Sun focussed more on agent programming and developing strategies.

For collaboration on the code, GitHub[1] was chosen as a revisioning system. No team member had any prior experience with GitHub. Some few members had worked with SVN as a revisioning system before. Nevertheless, we decided to use GitHub as it additionally offers a Wiki and an issue tracker. A Wiki is an online collaboration tool which enables users to create and edit hypertext pages within their Web browser (cf. [28]). We used the included Wiki for gathering the minutes of our weekly meetings. GitHub's issue tracker was used for complex problems, ideas or bug reports. It allowed discussions clearly separated by bug or feature. This distinct separation was not given for all bug reports as not all problems were transformed into issues. Instead, many small problems were discussed on our team chat. For this, we used the instant messenger Google Hangouts[2] as all team members already had registered a Google account. The main advantage of Google Hangouts over the issue tracker was that the response time was a lot lower due to its rather informal style. Some were just mentioned and discussed in the Hangouts group chat and then quickly solved after. It was also frequently used for short-dated organisational discussions, which would not have fit well into a ticket. As for voice over IP, we both used Google Hangouts and Skype[3]

[1] `https://github.com/` – last accessed 24.10.2014
[2] `https://www.google.com/+/learnmore/hangouts/` – last accessed 24.10.2014
[3] `https://www.skype.com/` – last accessed 24.10.2014

because some team members preferred one application over the other. Eclipse was chosen as the IDE because all of our team members were familiar with it and a plugin[4] for Jason exists. Besides syntax highlighting for AgentSpeak(L), the plugin also includes a promising mind-inspector for debugging agents and step-based debugging. Unfortunately, we had to find out that the plugin was not of great use for the Mars scenario. This was due to the short time frame per simulation step which for one resulted in each agent receiving a lot of information. Consequently, the mind-inspector often crashed or refreshed the information too fast. Similarly, step-based debugging was not possible because only the current code execution was halted but not the server simulation. As a result, debugging was mainly reduced to analysing log files generated from manually added print statements.

In sum, it can be said that we tried to keep our organisation to a basic form. We made sure that we were able to work well-structured but still quite self-organised and democratic in decision finding to encourage creativity in problem solving. Analogously, we spent little time on deciding what tools to use. Instead, we preferred software most of us had already used before or which was the leading free project for the given task. These approaches allowed us to concentrate on the actual multi-agent system development. It was necessary due to our inexperience and the scant time we had until the competition.

4 Architectural (?) Structure

Carefully reread and update our previous/next sections to find out if we need this at all. Maybe we can just cut it out so we don't have to stretch few information into complete subsections which could be covered in later sections alongside their original topics.

4.1 Agents

Talk a bit about generalisation e.g. a saboteur is a specialisation of an agent. I.e. both share exploring but the saboteur also knows how and when to attack. Explain what tasks our agents have and where our priorities are.

4.2 Simulation Phases

Explain the general split up into an exploration and a zoning phase. Also mention the parallel aggressive strategy of the saboteurs. Talk about repairers, explorers and inspectors and their special tasks but without going into details abouttheir complete strategy (this is part of the next chapter).

5 Algorithms and Strategies

Write this part

5.1 General Strategy Overview

This could also be an introductory text which motivates the following subsections.

Write this part or leave it out depending on how much we cover in the prior section introduction.

[4] http://jason.sourceforge.net/mini-tutorial/eclipse-plugin/ – last accessed 24.10.2014

5.2 Agent Specific Strategies†

Because each of the five different agent types (or *roles*) in the MAPC scenario — Explorers, Repairers, Saboteurs, Sentinels and Inspectors — have different capabilities in terms of the actions they can perform, they must each act according to role-specific strategies and tactics in order for the team to perform well. This section will give a short overview of how different agent types behave differently from each other.

Explorer agents are the only ones who can perform the `probe` action. Vertices must be probed in order to learn their value, which is critical for zoning. Accordingly, an Explorer will spend most of his time seeking out, moving towards and finally probing vertices whose value is not yet known. Since there are 6 Explorers in a team, care must be taken to make sure that multiple Explorers don't move towards the same unprobed vertex, as this is generally a suboptimal usage of their time.

In our implementation, we consider all vertices in the map to be "worthy" of being probed and thus to know their value. However, due to the way our team performs zoning (see Section 5.5.2), we prioritise vertices in a specific way which we call "cluster probing". Because our zone calculation algorithm (see Section 5.5.1) puts agents around a centre vertex in a circular manner and with a maximum distance of two edges away, and because we want the Explorers' probing to help us with quickly finding and establishing high-value zones, Explorers should avoid probing vertices in e.g. a straight line. Rather, an Explorer's probing pattern should mimic the circular shape of the zone calculation algorithm. Because of this, for probing we prioritise unprobed vertices first by distance, then by the number of edges they share with already probed vertices. The result is that Explorer movement is similar to a spiral pattern (provided the Explorer isn't disturbed by e.g. nearby enemy agents). Explorer agents don't stop this probing pattern until they can no longer find any unprobed vertices.

Repairer agents are the only agents who can perform the `repair` action for restoring health to disabled agents. Because the team loses out on possible points for every disabled agent in the team, to achieve a high score it is essential to quickly repair damaged agents. In our implementation, Repairers' actions are prioritised so that they will attempt to repair any disabled friendly agent in their visibility range, and it is the "job" of the disabled agents to find and move towards the closest friendly repairer. If a Repairer agent is aware of a friendly disabled agent outside of his visibility range, and the Repairer is currently not used for zoning, however, then the Repairer will also move towards the disabled agent.

Saboteur agents are the only agents that can disable enemy agents using the `attack` action. In our implementation, a saboteur's role is very aggressively defined, and is prioritised thusly: if you see a non-disabled enemy, attack it; otherwise find and move towards an enemy you can attack.

Throughout most of our development phase, Saboteur agents were the only agent type we would use the `buy` action for to extend their visibility range once for every Saboteur because we believed that this would give us an offensive edge against other teams. We decided to try out other buying strategies as well, however, by having our agents play matches against copies of themselves, except that the copies used different strategies for buying upgrades. Through this brief, empirical and rather informal testing period, we discovered that a surprisingly simple and novel strategy to buying upgrades actually led to persistently higher scores than our initial approach of buying one visibility range upgrade per Saboteur: by choosing a single Saboteur agent, which we call the *uber-Saboteur*, and allowing him to buy an unlimited (well, limited only by the amount of money available) number of upgrades as needed, we were able to outperform teams using our more conventional approach of upgrade buying. To be more specific, our uber-Saboteur would buy an upgrade whenever no active enemy was within his visibility range and he had the money for it. The kind of upgrade (maximum energy, visibility range, maximum health or strength) depends on the relative improvement that buying that upgrade will bring to the upgrade-specific "module". For example, if the uber-Saboteur has a maximum health of 3 and a visbility range of 1, then increasing the maximum health to 4 would be an improvement of 33 %, while increasing the visibility range from 1 to 2 would be an improvement of 100 % — so the uber-Saboteur will choose to buy an upgrade to the latter.

We also call in Saboteurs to defend a zone if it gets attacked by an enemy agent.

Sentinel agents don't have a unique action that they can perform. Their strength is that they start with a visibility range of 3 by default, which is the highest of all the agent types. This is useful during exploration and to be warned of incoming enemy agents, but we don't use any Sentinel-specific logic in our implementation worth mentioning.

Inspector agents are uniquely able to perform the `inspect` action. We consider it important to inspect every enemy agent once during each match to learn and store that agent's role (it is very important to know which enemy agents are Saboteurs, so that we can avoid them), but once that goal has been achieved, Inspector agents lose most of their importance. Achievement points for performing `inspect` actions are not awarded for inspecting an enemy agent more than once, and `inspect` is not needed to be able to tell if an agent is disabled (the `visibleEntity` percept includes the agent's current state). The only use case for inspecting an enemy agent more than once during a single match is to learn if they have bought any upgrades since the first time they were inspected. But because buying an upgrade for an agent is such a rare occurrence during the actual contest (cf. this year's and the last years' matches), we don't have to re-inspect very often — we could probably just inspect each enemy agent once to learn their role and then leave it at that. In our implementation, however, we toggle an enemy agent to be ready to be inspected again 50 turns after it was last inspected.

5.3 Exploration$^{\star,\circ}$

One precondition of the Agents on Mars Scenario is that all agents start with an empty belief base. No agent knows about its local and global environment. Every agent gets beliefs about its local environment quickly by receiving percepts from the server. But the agent stays unaware of the global environment. For our strategies it is crucial to have as much information of the overall environment as possible. This is necessary to e.g. find the shortest path from one vertex to another or to calculate a zone returning high scores. To achieve this, it is mandatory to somehow store information about the map like its vertices, edges between vertices, paths and agent positions.

The following subsections describe our different approaches on how to store and process this information. In more detail, the first section, Section 5.3.1, presents our initial approach with its up- and down-sides. After that a basic overview of our initial map building approach is given in Section 5.3.2. Finally, this chapter concludes with a description of the second approach we used and finally stuck to in Section 5.3.3.

5.3.1 Cartographer Agent.*

Very early in our development process we decided that we do not want to store the same information at different places. This means we did not want that each agent stores all map data in its belief base. The intention behind this decision was to reduce the effort in synchronising and maintaining data between the single agents. Our initial approach was to install one omniscient pseudo agent we called the "cartographer" agent. The cartographer agent represented a map and had the purpose to calculate shortest paths between given vertices and to store map information. These map information include data like traversing costs, edges between vertices and vertex values. Every agent told the cartographer agent about its environment related beliefs and the cartographer agent stored these beliefs. Environment related beliefs are listed in the listing below:

```
visibleEntity(<Vehicle>, <Vertex>, <Team>, <Disabled>).
position(<Vertex>).
visibleVertex(<Vertex>, <Team>).
probedVertex(<Vertex>, <Value>).
visibleEdge(<VertexA>, <VertexB>).
surveyedEdge(<VertexA>, <VertexB>, <EdgeCosts>).
```

Listing 1.12: Map exploration related beliefs

If an agent needed to know a shortest path, it queried the cartographer agent and got the shortest path as an answer. Also the agent could query the cartographer agent whether a vertex was already probed or surveyed. See Figure 6 for the communication process of our first approach. Figure 7 shows the distribution of the single components related to map exploration.

Shortly after implementing this approach, we encountered two major problems, which both resulted in serious performance issues. One problem came from our

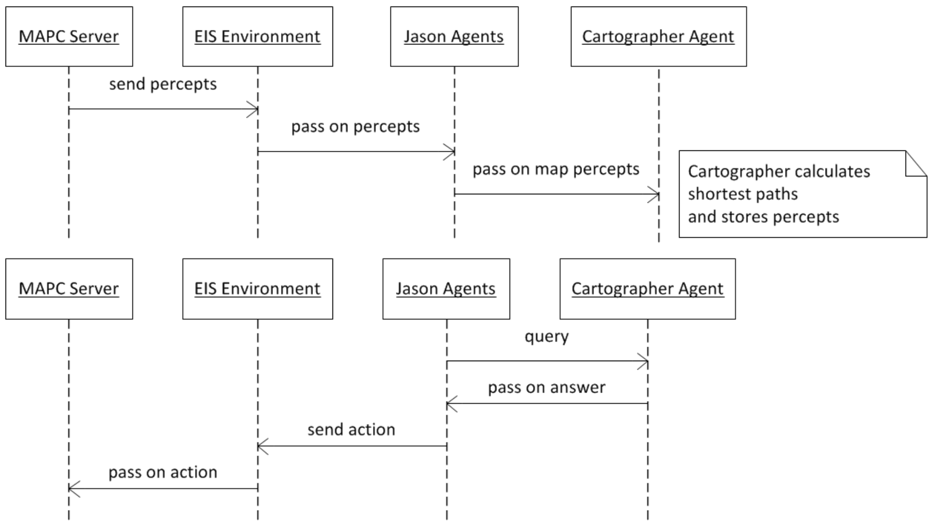


Fig. 6: Initial communication approach for map generation.

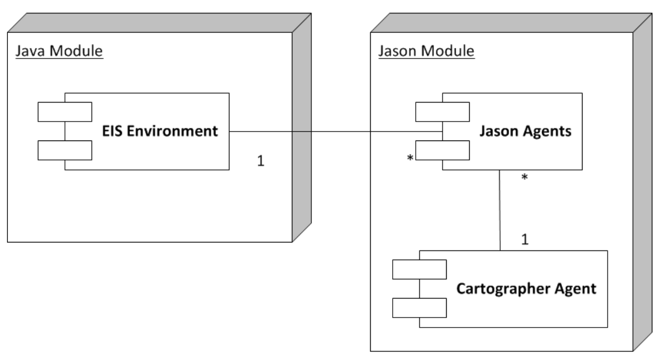


Fig. 7: Distribution of components between Jason and Java in our first approach.

implementation of the pathfinding. Although we used the Dijkstra-Algorithm, which is an efficient algorithm for this task, we encountered performance issues, because the pathfinding algorithm was executed every time an agent asked for a shortest path. This led to a lot of redundant calculation and processing in the cartographer agent. The second problem was related to communication between agents. To understand the latter problem, one need to know that Jason uses a message box system for communication between agents. This means that every message a sender sends to a receiver is queued in the receivers message inbox. In every Jason lifecycle only one message is processed. Although a Jason lifecycle is a lot shorter than a server lifecycle, after some execution time the inbox of the cartographer agent still was so full, that the processing of messages lagged far behind the receiving of these messages. Both issues resulted in blocked agents, which had been waiting for the response of their queries for rounds.

To illustrate the problem assume the following example: An exploring agent comes to an unvisited vertex. The first thing it does, is to ask the cartographer agent, whether this vertex was surveyed in the meantime. After it gets the answer it surveys the vertex or asks the cartographer for the next unsurveyed vertex and travels there. Then the whole procedure starts again. As one can see, there are two possible bottlenecks. The first one is the query for the state of an vertex and the second is the query for the next unsurveyed vertex, which includes calculating the shortest path to this vertex. As described before the cartographer has not only to handle queries, but also handles map information input of every agent in every server cycle. And due to the Jason communication approach this results in the cartographer not being able to handle queries immediate. In our tests we saw answer times for queries around ten till twenty server cycles, which means that our agents were idle most of the time, waiting for answers from the cartographer agent.

I currently don't know where the word "unsurveyed" appears the first. But it is not an actual word but something we invented. Hence, we must shortly mention, what we want to express with it.

5.3.2 Distance-Vector Routing-Protocol.*

To solve the problem with repeating calculations of shortest paths we decided to calculate all shortest paths from the beginning and store these paths and other information in a network of "node agents". Figure 8 shows the second approach we used. All percepts were passed through to the node agents. Jason agents queried the respective node agent directly instead of querying the cartographer. We extended our distribution model (see Figure 9) to include the node agents.

A node agent represents one vertex of the scenario map and holds information about this vertex. To make it easier to address node agents, we named node agents like the vertex they represented. Node agents store all their neighbours in a neighbour-list belief (`neighbours([<ListOfNeighbours>])`) and all available paths to other vertices as path beliefs (`path(<Destination>, <NextHop>, <Hops>, <CostToNextHop>)`). With regard to exploration and zoning we decided that an node agent also has to store the probed value of the vertex, and whether it was already probed or surveyed. We changed the cartographer agent's tasks, so that it only had to create the node agents at runtime. All information input and queries now were directed to the respective node agents.

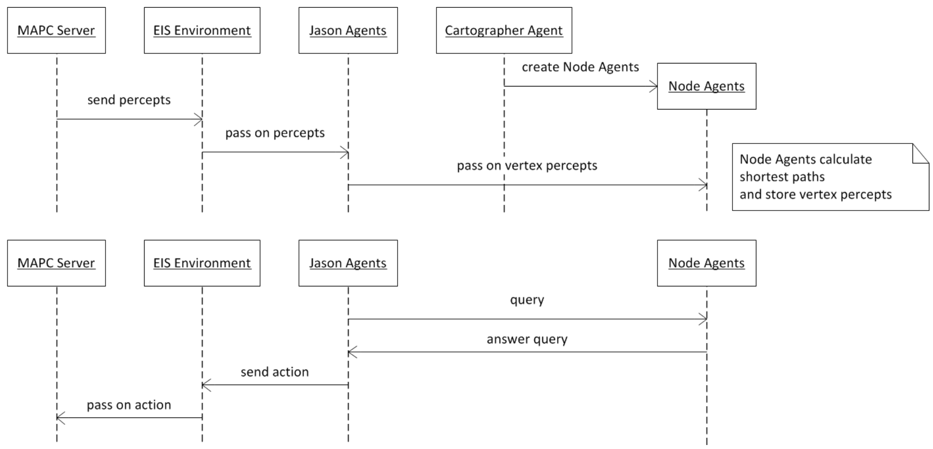


Fig. 8: Second communication approach for map generation.

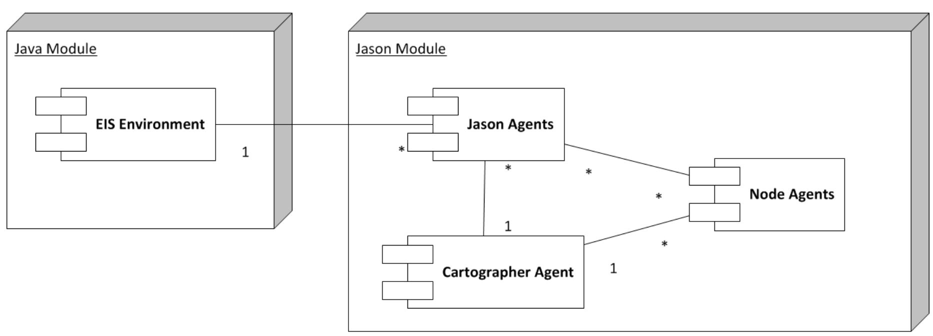


Fig. 9: Distribution of map components in our second approach.

The following list shows an example belief base of a node agent `v1`:

- `neighbours([v2, v3]).`
- `probed(true).`
- `probedValue(7).`
- `surveyed(true).`
- `path(v1, v1, 0, 0).`
- `path(v2, v2, 1, 3).`
- `path(v10, v2, 4, 3).`
- `path(v8, v3, 3, 2).`

A query for a shortest path would look like this:

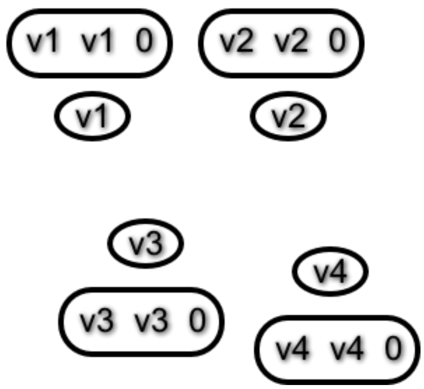
```
.send(v1, askOne, path(v8, NextHop, _, CostToNextHop)).
```

Listing 1.13: Query for shortest path from `v1` to `v8`

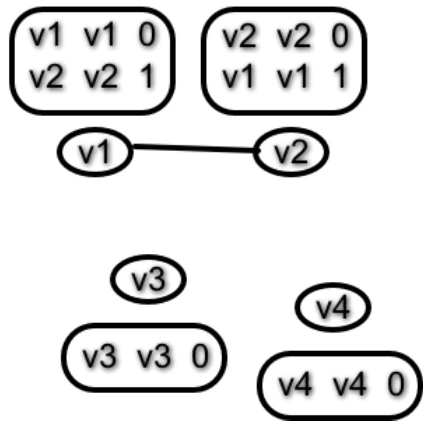
After looking up the belief in its belief base the node agent would unify the parameter `NextHop` with `v3` and `CostToNextHop` with `2` and response to the querying agent.

For propagating data between these node agents we used a modified *Distance-Vector Routing Protocol* (short: DV). DV is a routing protocol based on the Bellman-Ford algorithm and normally has its use in packet-switched networks. DV can be executed on a network of nodes. The basic idea is that each node informs all of its neighbour nodes about its belief base. The informed nodes then update their belief base and also inform all of its neighbours and so on. Because the Bellman-Ford algorithm has no loop detection, we had to implement some kind of break condition for the algorithm. We used the value of the calculated shortest path as a termination condition. If a new calculated path to another node agent is shorter than the already known path then the node agent has to inform its neighbours. Otherwise it just does nothing. At some point all information and paths are propagated through the whole network and all nodes have a consistent belief base. Figure 10 illustrates the algorithm for a small set of four neighbour nodes.

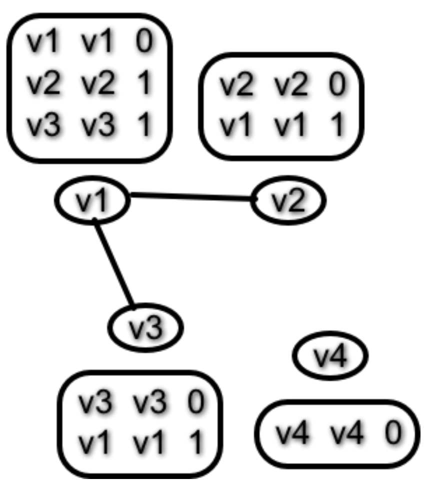
Fig. 10: Executing the Distance-Vector Routing Protocol algorithm as described in Section 5.3.2 on a small network of four nodes. Each node has a table attached, containing all accessible nodes. The first parameter is the destination node, the second one is the node to pass through and the third parameter shows the overall distance to the destination.



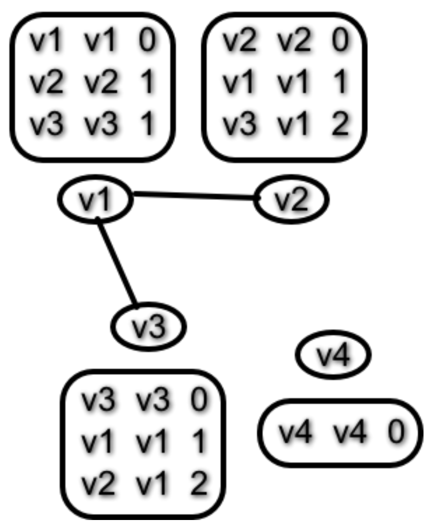
(a) Initial set of node agents with their belief base. Each node agent knows only about itself and the shortest path to itself. The travelling costs to itself are zero.



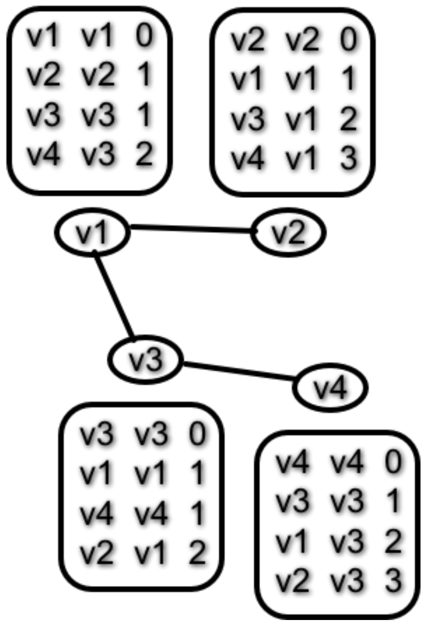
(b) The node agents `v1` and `v2` are informed that they are neighbours. So they know that there must be a path to the other node agent. Because they are direct neighbours they are one hop away. `v3` and `v4` are not connected to the network right now, so they will not be informed about the path between `v1` and `v2`.



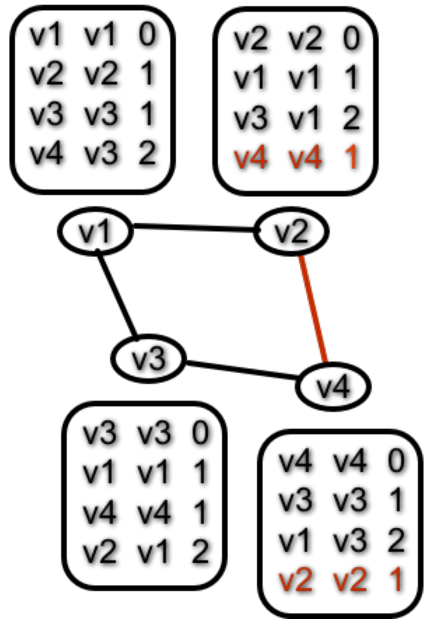
(c) Now `v3` is added to the network by informing `v3` and `v1` about their neighbourhood relation. At first only `v1` and `v3` update their belief base.



(d) In the next step `v2` is informed about the path from `v1` to `v3`. `V2` now knows that the shortest path to `v3` is by passing `v1` and that `v3` is 2 hops away.



(e) After that `v4` is added to the network and each node agent updates its belief base.



(f) In the last step `v2` and `v4` are informed that they are neighbours. Because of that the shortest path between both node agents changes and is updated. The paths for the other node agents stay as is, because they can not get better.

By distributing the information from one cartographer agent to a huge network of node agents we also distributed the load from one agent, the cartographer agent, between the respective node agents. But we still had queries which were not answered immediately, because now we had a lot of communication going on between node agents. Around 400 Jason agents were calculating shortest paths in parallel within the node agent networks and received at the same time information again and again by the exploring agents. This information however was redundant in most cases, but still had to be processed. This led to a high load on our system. As a first step, we restricted communication between node agents and real agents. The necessary map information was received directly from the Java EIS Environment module (short: JE). The JE used a filter to ensure that only map information was sent to node agents that they were not aware of. Since the message boxes no longer were flooded, communication between agents got a lot better. But two reasons made us to discard this approach also and to change to a solution based entirely on Java. First we were not able to reduce the load on our system by these changes. Further we observed that some beliefs were not received by the node agents. Even worse, over the time beliefs disappeared from node agents belief bases. Due to the constant high workload on our system, we saw that agents sent actions to the server too late, which led again to a lot of idle phases of our agents.

5.3.3 JavaMap.*

We decided to implement the whole map module in Java. This solved all of the previously described problems. The JavaMap module gets its map information directly from the JE and is queried through internal actions by the Jason agents (see: Section 2.3.6). Internally the JavaMap creates and manages a list of vertex-objects which are like the node agents from our previous approach. Every vertex has stored a list of all paths it knows and all vertex information. This includes the one-hop neighbourhood and the probed value of the vertex. Figure 11 shows how the map percepts are now passed to the JavaMap and how Jason agents query the JavaMap.

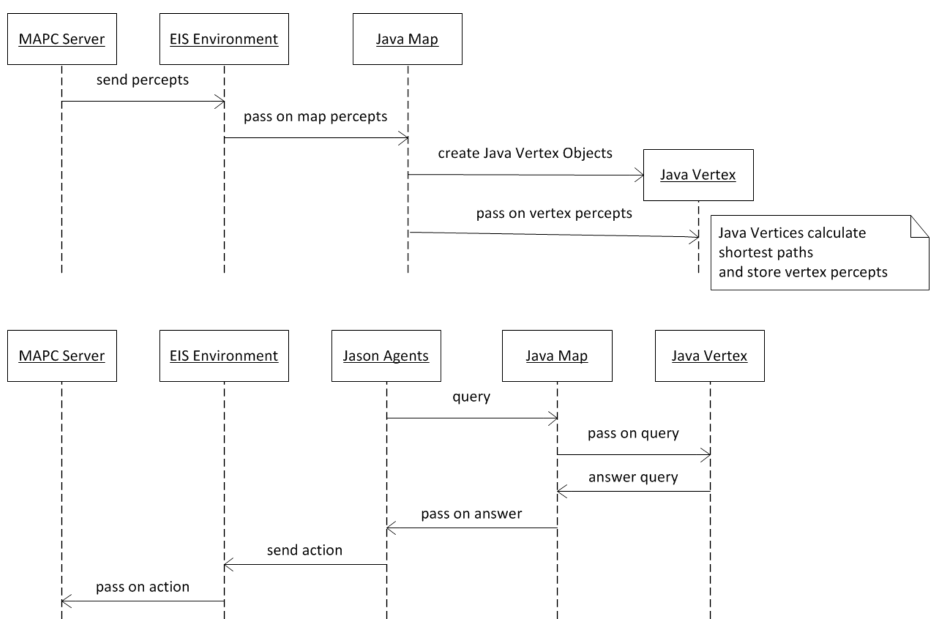


Fig. 11: Final communication approach for map generation.

In Figure 12 it is demonstrated how we changed the distribution of our components for map generation between Java and Jason. Unlike the first and second approach we now have a separation of concerns. The whole map generation and calculation is done entirely by Java and agent related actions and communication entirely in Jason.

This section is a bit short and does not have a real ending.

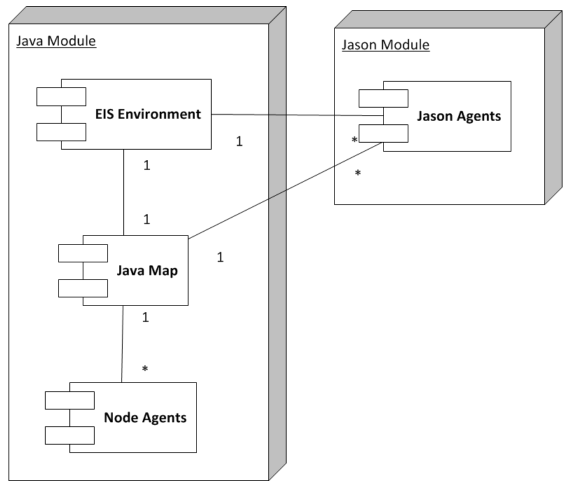


Fig. 12: Distribution of map components in our final solution.

5.4 Repairing◇

As was already mentioned in scenario description, any agent can get disabled after being attacked by the enemy saboteur. To get disabled in the scenario means to loose all the health points. Naturally, we have implemented several supporting algorithms of avoiding enemy saboteurs when possible and parry when there is a saboteur nearby. However following these algorithms cannot guarantee that the agent will never get disabled, mostly because there not always exists an escape route and not all agents can perform parry action. When an agent gets disabled it looses most of its functionalities: only skip, recharge and goto actions remain active. Additionally repairers even being disabled can perform a repair action, but it costs more energy in this case. Disabled agents also cannot occupy a vertex and therefore cannot participate in zone building, which is a vital part of getting score points.

In Section 5.2 it was said that the primary task of repairer agents is to repair others and they should perform a repair action whenever they see a disabled friendly agent. But the question is how to get the disabled agent to the repairer. In our implementation, every time an agent gets disabled, it sets a high priority goal *getRepaired*. Following the plan of this goal, an agent requests the available repairer and its position from the *MapAgent*. If the returned repairer position is the same the agent's position, then the agents only recharges and waits to get repaired, otherwise the agent simply moves toward the returned repairer position. If there is no repairer available in the reachability range, it means that the graph is not sufficiently explored and the agents then tries to expand the known subgraph by moving to unvisited vertices.

Assignment of agents to their repairers is done inside our Java MapAgent. To be more flexible we decided to perform such assignment on every step. This allows to adapt to constantly changing situation, when agents are moving, some new agent get disabled and some agents get repaired. The assignment itself is done based on the hop distances between agents. First, all the distances between all disabled agents and all repairers is calculated. Then the closest distances are picked and the agents which have this distance between them are assigned to each other. If all repairers are assigned and there are still some unassigned disabled agents, they get assigned to the closest to them repairers. This assigning approach in most of the cases led to fast and effective repairing.

In addition to disabling agents moving to their repairers, repairers can also move towards the assigned to them disabled agents. This behaviour is only possible during the exploration phase of the simulation, because in zoning mode repairers moving somewhere in most of the cases will also mean the zone breakup, which we would like to avoid. We implemented this by making repairers explore the map in the direction of the assigned to them disabled agents, i.e. if the vertex is not surveyed, they survey, otherwise just go to the assigned disabled agent.

The seeming special case in repairing is when one of the repairers gets disabled. However since even disabled repairer can perform repair action, we decided to use the standard procedure of assignment even to the disabled agents. The only difference is that the goal to repair have higher priority then the goal to get repaired. This helps to use all the repairers more effective and prevents the situation when all the repairers are disabled and waiting to get repaired.

5.5 Zone Forming°

Zone forming is the most important part in the MAPC Mars scenario [2].It describes the process of agents finding and occupying vertices in a way that they enclose a subgraph. We called this process zoning. For our approach, zoning takes place after the map exploration phase. This should ensure that enough information about the graph has been gathered to calculate high valuable zones close to the agents' current positions. The algorithm for calculating zones and determining which agents have to occupy which vertices is presented in Section 5.5.1. Said algorithm is used in the process of finding and negotiating a zone to build, which is described in Section 5.5.2. After a zone that can be build has been found, agents get assigned dedicated roles. These roles determine the agents' duties and tasks throughout the lifecycle of a zone which they are part of. The lifecycle of a zone includes its creation, defence and destruction. Both roles and the lifecycle are featured in the last Section 5.5.3.

5.5.1 Zone Calculation†

The graph colouring algorithm used by the MAPC server to determine occupied zones is described in detail in the MAPC 2014 scenario description [2] and will not be explained again here.

Due to the way the server-side colouring algorithm works, placing n agents on the map so that they establish the highest possible zone value per step is anything but straight-forward. Even for $n = 1$, a single agent placed on an articulation point in the graph can establish a high-value zone if there are no enemy agents in either subgraph that it splits the map into. Figure 13 shows an example. To position themselves in an optimally-scoring way, agents *could*

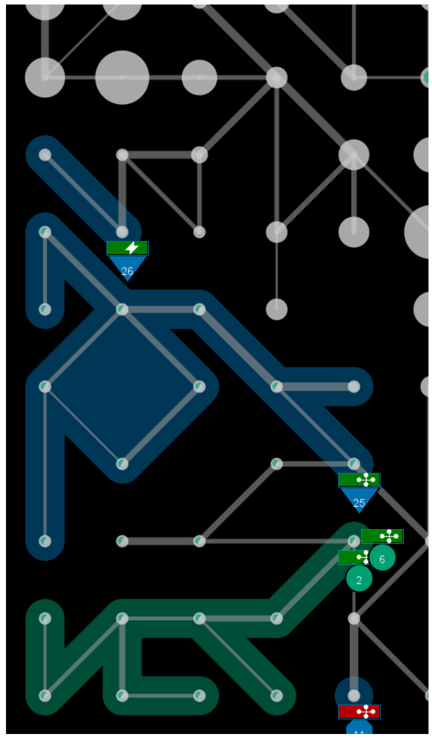


Fig. 13: By occupying an articulation point, a single agent can ostensibly establish a a high-scoring zone—provided that there are no enemy agents inside the subgraph that is split off from the main graph.

run the same algorithm locally to calculate the agent placement that will lead to the highest total sum of zone scores in each step by trying every possible permutation. However, the number of ways to place n agents on k vertices is $C\,(n+r-1, r-1) = \frac{(n+r-1)!}{n!(r-1)!}$, a number that increases rapidly with n and k. In particular, there are $C\,(28+600-1, 600-1) = 3.75 \times 10^{48}$ ways to place 28 agents on 600 vertices, which were the numbers used in the 2014 competition—far too many to calculate in real-time. Finding an algorithm that calculates high-scoring zones in a limited computation time is one of the major challenges of the MAPC competition.

Our team developed a heuristic algorithm for calculating zones that will be explained below. The goal is to find for every vertex in the graph a placement of agents around that vertex such that:

- All vertices that share an edge with the centre vertex will be included in the zone.
- Agents should only be placed on the centre vertex's two-hop neighbours, which are those vertices that are be connected to the centre vertex through a minimum and maximum of two edges.
- The constructed zone's value per agent, that is, the sum of the values of each vertex in the zone divided by the numbers of agents required to establish that zone, should be high. Ideally, it would be maximal, but the heuristic we use doesn't guarantee this.

Figure 14 shows some examples of zones that are found using our heuristic algorithm. Internally, every vertex in the graph is represented by a Java `Vertex` object, and the calculated zone is stored as a field of that object. The steps of the algorithm are best detailed graphically, as in Figure 15.

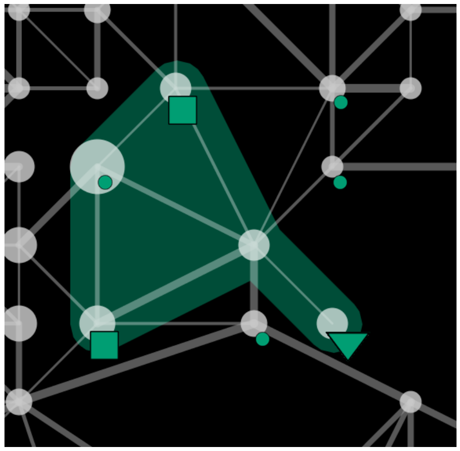
Definition 1. *Let V be the set of vertices and E the set of edges that the system knows about. For any $v \in V$, which we will use to denote the vertex that a zone is centered on, let $V_v^1 \subseteq V$ be the set of one-hop neighbours of v, that is, the set of vertices that share an edge with v: $V_v^1 = \{w|(v,w) \in E\}$. Similarly, V_v^2 denotes the set of two-hop neighbours of v, i.e. the set of vertices that includes exactly those vertices that share an edge with any vertex in V_v^1, excluding those in V_v^1 and v itself: $V_v^2 = \{u|(v,w) \in E, (w,u) \in E, u \notin V_v^1 \cup \{v\}\}$. Let V_v^{2+} denote the entire two-hop neighbourhood of v: $V_v^{2+} = \{v\} \cup V_v^1 \cup V_v^2$. Additionally, let A_v be an initially empty set that we will use to remember vertices we want to place agents on. A zone and its zone value are defined as specified by the graph colouring algorithm in [2]. Then, the goal of the zone calculation algorithm is to find, for every $v \in V$ in the graph, a set of agent positions $A_v \subseteq V_v^{2+}$ that establish a zone around v so that the zone's value* per agent *is high according to the heuristic used by the algorithm.*

Note that although V_v^1 and V_v^2 start off as defined above, by abuse of notation we will remove vertices from those sets as the algorithm progresses. This does not mean that the structure of the graph has changed. The algorithm for zone calculation is (re)-triggered every time a vertex in the vertex' two-hop neighbourhood (so within the ambit of the zone we're trying to calculate) is discovered during map exploration or changes its known value when it is probed by an Explorer agent, as these are the events that can lead to possible changes in A_v and thus the zone.

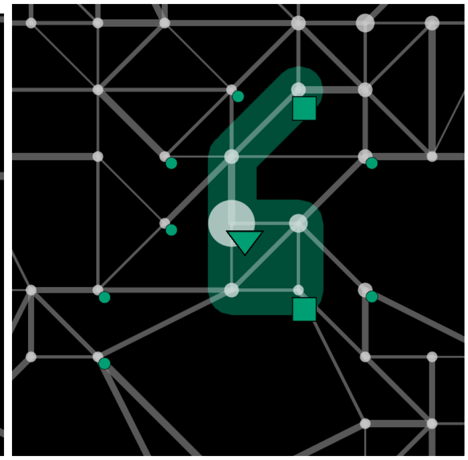
The zone centered around vertex v is is calculated through several steps:

1. Initially, $A_v = \emptyset$, and V_v^1, V_v^2 and V_v^{2+} as defined above. Iterate through every $w \in V_v^2$ and, for every w that is connected to 2 or more vertices in V_v^1, add it to A_v and remove it from V_v^2:

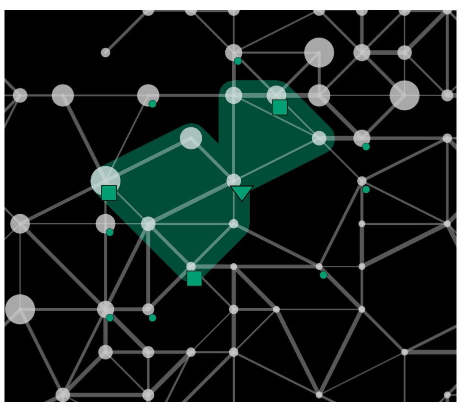
$$\forall w \in V_v^2 : \{(w,u_1),(w,u_2)\} \subseteq E, u_1 \neq u_2, \{u_1,u_2\} \subseteq V_v^1$$
$$\rightarrow A_v := A_v \cup \{w\}, V_v^2 := V_v^2 \setminus \{w\} \quad (12)$$



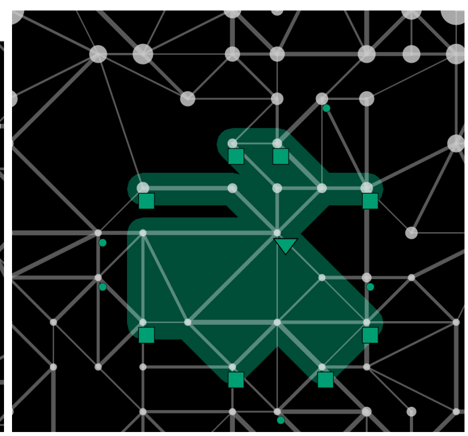
(a) This zone was calculated for a centre vertex that only has a degree of 1, i.e. that is a leaf vertex. Generally, it is preferable to place an agent on the cut vertex that leads to a leaf vertex rather than the leaf vertex itself, as this would establish at least a zone of equal size, and possibly larger.



(b) Here, the centre vertex has a degree of 3, and the calculated zone remains compact with only two additional agents used. A lot of optional agent positions remain.



(c) A zone where the centre vertex has a degree of 5, and the zone uses a total of 4 agents.



(d) A zone where the centre vertex has a degree of 7, and the zone uses a total of 9 agents.

Fig. 14: Four examples of zones calculated by the heuristic algorithm described in Section 5.5.1. The green squares and triangles represent the placement of agents, where the triangle is the agent on the center vertex. Vertices marked with a small green circle are optional agent positions that can be used to expand the zone if there are agents left over at the end of the zone building, as described in Section 5.5.2. The green-colored area represents the zone that is established by the given agent placement.

2. For every $w \in V_v^2$, if w is connected either directly or through a single one-hop neighbour of v to any $u \in A_v$, remove it from V_v^2:

$$\begin{gathered}\forall w \in V_v^2 : \exists u \in A_v : \rightarrow V_v^2 := V_v^2 \setminus \{w\} \\ \forall w \in V_v^2 : \exists u \in A_v : \exists x \in V_v^1 : \{(w,x),(x,u)\} \subseteq E \rightarrow V_v^2 := V_v^2 \setminus \{w\}\end{gathered} \tag{13}$$

   The reasoning behind this is that those vertices in V_v^1 that are neighbours of those in A_v will already definitely be included in the zone, and the vertices we remove this way will not contribute towards our goal of including all one-hop neighbours V_v^1 in the zone for v.
3. In the next step, "bridges" are discovered in the list of remaining two-hop neighbours V_v^2. A *bridge* is considered to be a connected triple of vertices where one of the vertices is directly connected to the other two. If such a bridge exists in V_v^2, all three involved vertices can be included in the zone around v by placing an agent on either end of the of the bridge and leaving out the in-between vertex:

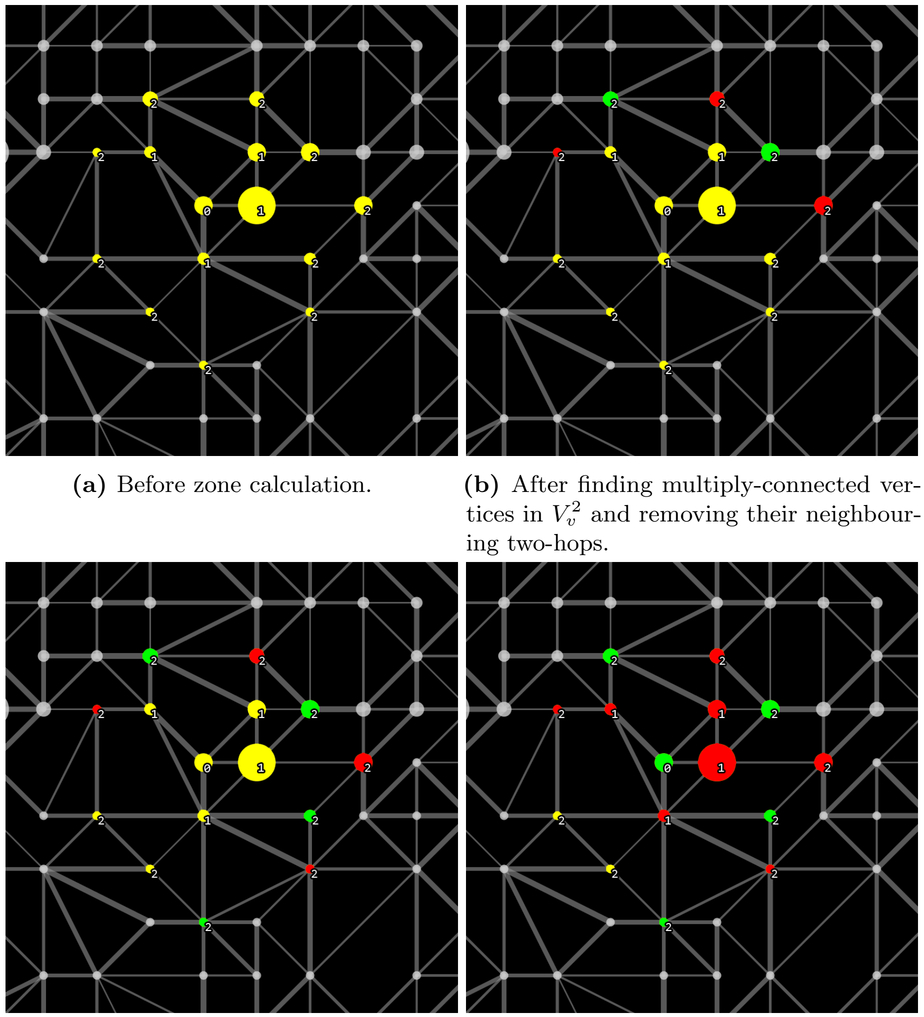
$$\begin{aligned}&\forall w_1, w_2, w_3 \in V_v^2 : (w_1, w_2), (w_2, w_3) \in E \\ &\qquad \rightarrow A_v := A_v \cup \{w_1, w_3\}, V_v^2 := V_v^2 \setminus \{w_1, w_2, w_3\}\end{aligned} \tag{14}$$

   Since three vertices can be captured in the zone for the "cost" of two agents, we consider this a good exchange to make.
4. Next, the algorithm checks if all one-hop neighbours V_v^1 are connected to the agent positions A_v. This is frequently the case, but not always. If a remaining, unconnected one-hop $u \in V_v^1$ is found, we check if it is connected to one or more of the remaining two-hop vertices in V_v^2. If that is the case, we choose the neighbouring two-hop $w \in V_v^2$ with the highest vertex value and add it to the list of agent positions A_v. If no such two-hop vertex is found, we add the unconnected one-hop vertex to the list of agent positions—this is the only case where a one-hop vertex can be added to the list of agent positions:

$$\begin{aligned}&\forall w \in V_v^1 : \neg\exists u \in A_v : (w,u) \in E \\ &\quad \rightarrow \begin{cases} A_v := A_v \cup \{x\}, V_v^2 := V_v^2 \setminus \{x\} & \text{if } \exists x \in V_v^2 : (x,w) \in E \\ A_v := A_v \cup \{w\} & \text{else}\end{cases}\end{aligned} \tag{15}$$

5. Finally, we include the center vertex v in the list of agent positions: $A_v := A_v \cup \{v\}$. Any vertices that remain in V_v^2 are saved as additional agent positions that could be used to extend the zone by otherwise idle agents, but unlike the vertices in A_v vertices are not required to establish the initial smallest zone that we calculated.

While we consider our algorithm to find zones of acceptably high zone values per agent, it can easily be shown to be suboptimal. For one, it only considers vertices within the two-hop neighbourhood of the centre vertex, and it is not difficult to



(a) Before zone calculation.

(b) After finding multiply-connected vertices in V_v^2 and removing their neighbouring two-hops.

(c) Finding bridges.

(d) After zone calculation.

Fig. 15: The zone calculation algorithm shown in four steps. Vertices are coloured differently according to their current state as the algorithm progresses. Green vertices are vertices that an agent must be placed on, so those in A_v. Red vertices are those where placing an agent would be redundant because it does not help with the goal of including all one-hop neighbours V_v^1 in the zone. Yellow vertices denote notes where agents could be placed to extend the zone, but are not considered optimal agent positions in the eyes of the algorithm. Numbers shown next to vertices represent their edge distance from the centre vertex v.

think of possible graph structures where a different agent placement would lead to a better zone. For example, if one of the remaining yellow vertices in Figure 15d were an articulation point whose inclusion in the list of agent positions would add more than that single vertex to the zone, this would probably be a good vertex to place an agent on, yet this would not be discovered by our heuristic algorithm.

5.5.2 Zone Finding Process.°

This section explains how agents decide on what zones should be built. Each zone finding process can end successfully or fail for any individual agent. If it failed, the agent is not going to be a part of a zone and a new zone finding process is started. The first part of this section covers what changes are made so that with every failed zone finding process a successful one becomes likelier. If a zone finding process ended succesfully, the most valuable zone known to the agents will be built. This is ensured through agent communication which is presented in the last part of this section. Agents start looking for zones when they have finished the exploration phase. As explained in Section 5.3, explorer agents do not only survey but probe as well. Hence, other agents may finish the exploration phase earlier. Furthermore, zones can be broken up at any time forcing the agents to start looking for a new zone again. As a result, the zone finding process is in fact asynchronous. Problems arising from this are mainly dealt with throughout the actual building of zones which is illustrated in Section 5.5.3.

In the beginning, all agents have to centrally register themselves when they are ready for zone building to indicate this availability. Next, each agent uses the algorithm presented in Section 5.5.1 to determine the best zone in its neighbourhood. The algorithm uses a range parameter k which indicates the k-hop-neighbourhood up to which the algorithm will look for the best not yet built zone. This range starts at 1 and is incremented every time the agent finishes a zone finding process without being part of a zone afterwards. As a result, it is more probable for an agent to find a zone with a high per agent score which has not been build yet. Thus, it is also likelier for the agent to be part of a zone, because throughout every zone finding process only the most valuable zone is going to be built. The range has a maximum to ensure that an agent will not look for zones too far away from it. When a zone is broken up, the range will be reset, which is covered by Section 5.5.3.

After every agent interested in building a zone has determined the best zone in its neighbourhood, all such agents must send their best zone to all other agents. This is because all agents ready to build a zone should know about and hence only try to build the best globally, not yet built zone. At any time, every agent may only know about one zone. This zone will be the best zone an agent is aware of at the moment. Zones are being compared by their per agent score. A higher score indicates a better zone. Before building any zone, the agents will have to wait until the information about their best zone has reached all other agents. This is ensured by the agent having to wait for all other agents to reply to him. Therefore, when an agent receives information about a zone, it has two options.

One is to reply with a simple acknowledgement message expressing that it had received the message. The other is to reply with its own zone in case that its zone is better. Agents may not reply with information about a better zone if it is not their own. This is to prevent duplicate messages. Otherwise, multiple agents could reply with the same zone of which they had been informed about by the same agent. Whenever an agent receives information about a better zone, it replaces its former knowledge about the best zone with the new one. Agents which are not interested in building a zone but receive information about a zone simply ignore the message but reply with an acknowledgement. This way, the sender will still be able to determine when every agent has processed the sent information. In case the zone calculation algorithm did not return any zone to an agent, this agent has to ask all other agents for a zone. It will accept the first reply containing zone information as its new best zone because it is better than no zone. The agent will then continue similar to the earlier presented behaviour and wait until it received replies from all other agents. After an agent has received all replies, it may start building a zone as illustrated in the next subsection.

5.5.3 Zone Building Roles and the Lifecycle of a Zone.°

This subsection describes the two roles exclusive to zone building. It covers the roles' associated tasks and duties throughout the lifecycle of a zone as well as the lifecycle itself. These roles are those of a *coach* and a *minion*. Each zone is built by one coach and a varying amount of minions. Minions are agents which are dedicated to build a zone by obeying their coach's orders. Every agent may only be part of one zone at a time. The roles are assigned when the zone finding process has ended and a concrete zone is about to be built. Agents keep either of these roles until the zone is broken up or they have to leave it. The roles regulate the agents' behaviour throughout the time they spend in a zone.

Before looking at border cases, an ideal case of a zone lifecycle is presented. There, the zone finding process described in Section 5.5.2 ends with all agents knowing about the same best zone. This zone was found by one agent which will then become the zone's coach. Next, the coach informs the agents which will be part of the zone where to go to. On receipt of this message, the agents become minions and move to their designated vertex. The coach will also have to move to its vertex, which happens to be the centre vertex of the zone. Furthermore, the coach will unregister itself and all its minions to indicate their unavailability to build any other zone. In a zone, minions serve no other purpose than to occupy their designated vertex. If a minion becomes disabled, it has to move towards a repairer agents. Due to this, it has to leave its vertex. Therefore, the zone can no longer exist in its original form. In such a case, a minion has to inform its coach about its departure. The coach must then tell all its other minions that the zone can no longer be maintained. Consequently, all affected agents drop their role and restart looking for zones as illustrated in Section 5.5.2.

In reality, the zone finding process is asynchronous. Therefore, it is likely that some agents start looking for a zone when others have nearly finished. As a result, there can be multiple groups of agents with different knowledge about which zone would currently bring the highest score per agent. Each group could then be expecting a different agent to become a coach. This interferes with the assumptions that each agent may only be in one zone and have only one role at a time. To solve this problem, coaches do not only inform their minions about where to move to. Instead, they also transmit the per agent score of the zone they want to build together with this agent. Any agent can then compare the received zone score with the zone it wanted to build before. If it is higher, it must inform the coach of its former zone or its minions if it had been the coach itself. In case that the proposed zone's score is lower than the zone the agent intended to form, it must inform the coach who just proposed the new zone. Said coach will then have to inform all its minions that its zone is not going to be built.

Besides coaches and minions, there are also other agents who might be looking for a zone but will not be part of the one which will be built. Such an agent will have to start a new zone finding process. Prior to that though, it will look for any highly valuable vertex in its surrounding which is not yet occupied by anyone and move there. The range to look for such a vertex is the same as the range for finding a zone in the agent's neighbourhood presented in Section 5.5.1. It is increased after every zone finding process which does not result in a zone where the agent is part of. The idea is that with a wider range, the probability to find a highly valuable zone increases. Additionally, the agent will likelier move farther away from its position in case it is not part of the zone to be built. This should further ensure that zones are only proposed multiple times as best zones if they have a very high per agent score.

We assume due to our zone calculation algorithm that a vertex within a zone will be occupied by at most one agent. Then, any enemy agent close to a zone endangers it. This is because a zone may not spread across an enemy inside of it [2]. Moreover, enemy saboteur agents can disable agents inside a zone, which similarly destroys the zone in its original form [2]. Hence, coaches check once per step whether an enemy agent is close to the zone. If this is the case, the coach broadcasts a message to all saboteur agents to come and defend the zone. Saboteur agents which are not already defending a zone bid for this. The saboteur agent closest to the zone's centre will win the bidding. It then moves towards the enemy to disable it. If the coach detects in a next step that the enemy moved away from the zone, it will cancel the zone defence. The coach does so by using another broadcast as it does not know which saboteur agent was selected to defend the zone.

Explorer agents will still be probing when the first agents start looking for zones. Therefore, the most valuable zones may change with more and more vertices being probed. To prevent that agents build a zone once and stay there for ever if no agents attack them, zones will be split up periodically. The periodic trigger is linked to the overall steps of the simulation and not the lifetime of each zone respectively. Consequently, agents from different zones will have to restart looking for a zone at the same time. In addition to allowing new zones to be build which take the information of the newly probed vertices into account, this

This section is missing a proper ending

also allows for agents to start the zone finding process in a less asynchronous fashion.

6 Implementation Details

write this part

6.1 BDI in AS(L) and Jason▲

The BDI agent architecture has been a central theme in the multi- agent systems literature since the early 1990's. AgentSpeak is an agent-oriented programming language inspired by the work on the BDI architecture and BDI logics as well as on practical implementations of BDI systems[7].

AgentSpeak(L) is a programming language based on a restricted first-order language with events and actions [38]. What was written in AgentSpeak(L) decides the behaviour of an agent, such like the agent's interactions with the environment. In other words, we can design the beliefs, desires, and intentions of the agent by writing these notions in AgentSpeak(L) instead of representing them as model formulas explicitly.

Beliefs are the current states of the agent which are not immutable but updated when the environment changes or some events that can affect agents' beliefs are triggered; states which the agent wants to bring about based on its external or internal stimuli can be viewed as desires; and the adoption of programs to satisfy such stimuli can be viewed as intentions[38].

Jason is a Java-based platform that can interpret for an extended version of AgentSpeak(L) and it makes AgentSpeak(L) language practically suitable for multi-agent systems, therefore we implement Jason for this multi-agent system programming. Some details on the functioning of an AgentSpeak(L) interpreter is presented below:

An AgentSpeak(L) agent is defined by a set of beliefs giving the initial state of the agent's belief base, which is a set of ground (first-order) atomic formula, and a set of plans which form its plan library[7]. We can see that clearly from this figure, the beliefs stored in belief base do not only come from the perceptions of the environment but also can be added by the agent itself from the execution of a plan.

The beliefs from perceptions are annotated by `[source(percept)]`, and in our multi-agent programming, 34 this kind of beliefs are stored. While, some internal beliefs are also used in our codes, which are with annotations like `[source(self)]`. Beliefs describe basic current states of each agent such as the name of an agent, the energy, the role and so on. These beliefs are not immutable and can be used in writing plans and goals.

In the interpretation cycle, we see events also playing an important role. After the selection of the events by S_E, which is the event selection function, the event should be unified by the interpreter with the trigger events in the head of plans. An event has two types: internal event and external event. An event is internal

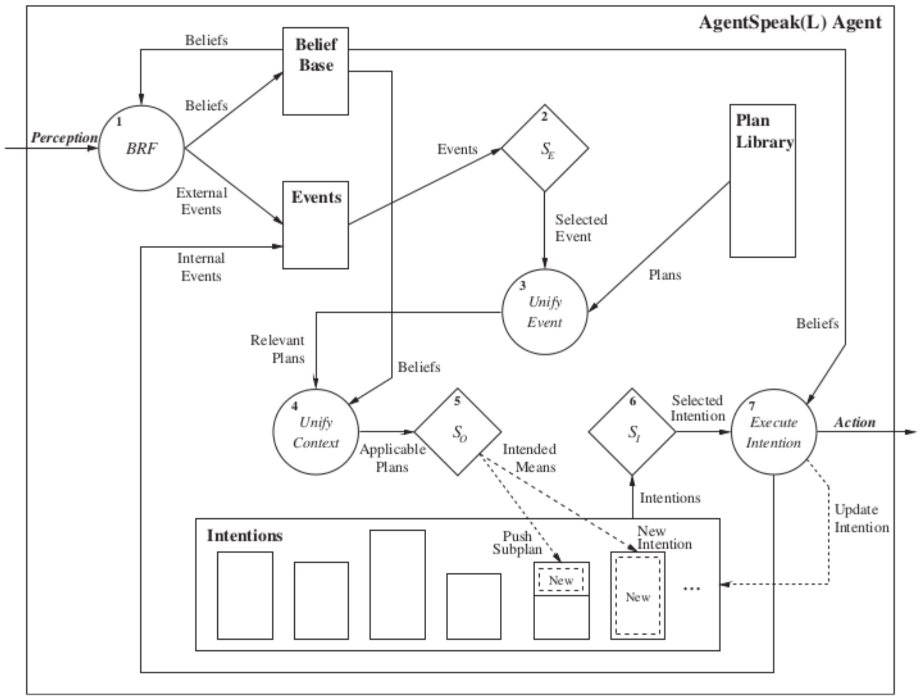


Fig. 16: An interpretation cycle of an AgentSpeak(L) program [7].

when a sub-goal needs to be achieved and an external event is generated from belief updates as a result of perceiving the environment. We have 88 events in total in our program and two of them are presented as follow:

```
+!doParry
  energy(Energy)
  & Energy < 2
  <- recharge.
```

Listing 1.14: Internal Trigger Event.

```
+lastActionResult(failed_status)
  <- -+health(0)[source(self)]
```

Listing 1.15: External Trigger Event.

In Listing 1.14, we see the internal trigger event `+!doParry` which is for achieving the goal `doParry`, on the other hand, in Listing 1.15, the trigger event `+lastActionResult(failed_status)` is external which means when the belief last action result is failed_status" has been added, it will execute what is shown in the body of this plan.

Desires in BDI model are always treated as goals which used in plans. A goal of AgentSpeak(L) has two types: achievement goal and test goal. When the associated atomic formula is true, and the agent wants to achieve a state of the world, it is stated as an achievement goal with being formed by an atomic formula prefixed with the `!`. On the other hand, a test goal which is prefixed with the `?` operator states that the agent wants to test whether the associated atomic formula is (or can be unified with) one of its beliefs [7]. We mostly implement achievement goals in our program, because achieving some states are always wanted. For example, `!doParry.` is an achievement goal that will execute the `parry` action when some events are triggered. Although achievement goals occupy about 99 percent of the goals, we still implement three test goals such as `?maxRange(MaxRange)` which checks the current range is maximum or not when finding the best zone for agents. All of these three test goals are implemented in zone building, and all the remaining goals are achievement goals.

With beliefs, trigger events and goals, we can make plans for each agent. An AgentSpeak(L) plan consists of a head which is formed from a triggering event, a conjunction of belief literals representing a context, and a body which is a sequence of basic actions or goals that the agent has to achieve when the plan is triggered[7]. In our program, there are 184 plans in total. 101 of them have the head of internal trigger events as well as 83 plans are with the head of external trigger events. The plans with internal trigger events are 18 more than those with external events, that means when plans are made, we add or delete goals which generated from the agent's own execution of a plan much more than adding or deleting beliefs based on the perception.

Different plans can ensure to complete different tasks which are required to be done by several kinds of agents. In this multi-agent system, several actions are already defined before starting this program. What required to do is to make good plans to let different agents implement these actions well so that good points can be acquired. For example, 6 out of 28 agents are allowed to execute the action "attacking". When we try to make plans for the agents who can attack, a variety of situations should be considered of; like when the enemy stands on the neighbour vertex of our agents, or when our agents don't find any enemies nearby, laying different plans is necessary. Of course, the allocation of plans' quantity can not be the same for executing different actions, because strategies for executing various actions differ under different circumstances. Building zones occupies 39 percent of the plans and the second largest part is contributed by the basic beliefs storing. However, all the plans for storing basic beliefs are with external trigger events meanwhile they are very fundamental. For instance, `+health(Numeral)[source(percept)] <- +health(Numeral)[source(self)].` is just to store the health updating every step. Therefore, basic beliefs storing can be ignored in this comparison. Zone building uses more plans than others. It is easy to know that building a zone is more complex than other action executing. Zone building will be affected by the maps given in the contest, the position where the agents of other teams occupy and many other situations as well. Additionally, building a good zone will get a lot of scores so more plans focusing on zone building is reasonable. Compared with zone building, the plan for the action `buy` is easy. When the specified saboteur agent does not see any enemy agents nearby, it will buy more health and increase its visibility range. The action `attack` uses a little more plans than other remaining actions since the strategy is designed offensively. Making agents from other teams disabled is our purpose, moreover the attack action should be executed rapidly and effectively. Consequently, in contrast, more plans are adopted by the attack action.

Now we have relevant plans that unified with the selected events and plans. However, these relevant plans can not be executed at this time, because the beliefs from the belief base also should be unified in the plans. After this, the option selection function S_O selects one applicable plan and puts it into the intentions.

Intentions are particular courses of actions to which an agent has committed in order to handle certain events. Each intention is a stack of partially instantiated plans[5]. The execution of plans may be started off by their trigger events. As mentioned above, trigger events can be external when coming from the perception of the changing environment, or be internal when generated from the sub-goals. We know that, applicable plans were chosen and putted into the intention stack. If the chosen plan is for an internal event, it will be pushed on top of that intention. Otherwise, if the chosen plan has a head with an external trigger event, it will create a new intention and be stored in the intention stack. The allocations of this two types of plans are different for various agents. According to a rough statistic, we can find the differences in the following figures.

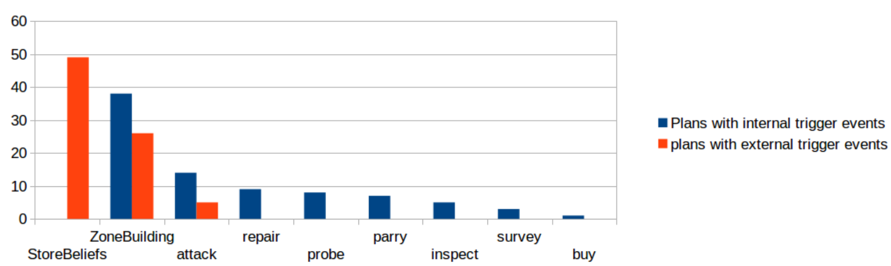


Fig. 17: Plan distribution for actions.

In Figure 17 we can see that all plans used by storing basic beliefs are with external trigger events as mentioned above. Besides that, only zone building and execution of action ”attack” adopt plans with external trigger events. The other remaining actions use few plans with external trigger events. In general, internal trigger events are more often used in plans. After calculating, about 41 percent of plans for zone building are for external trigger events but 23 percent of plans for execution of ”attack” are for external trigger events. So more zone building plans

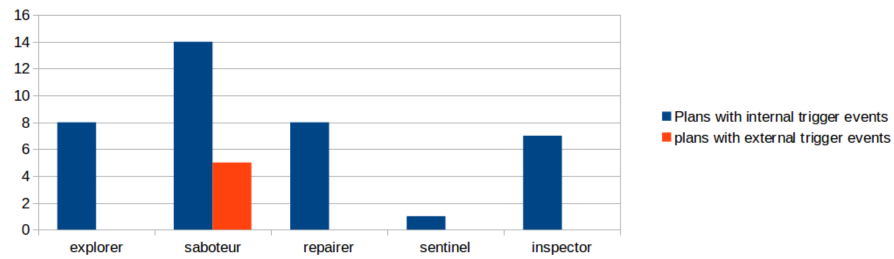


Fig. 18: Plan distribution for agents.

are triggered by the perception of the agent's environment than that for "attack". In this program, plans with internal trigger events which generated from the agent's own execution of a plan are adopted widely. Therefore, the execution of the previous plan makes more sense to the current plan of agents than the perception of the environment's impact.

Agents in this multi-agent system are divided into five classes by their roles—explorer, saboteur, repairer, sentinel, and inspector. They have various tasks to do to achieve points in the contest, moreover, the number of plans adopted by them are definitely different which can be seen in Figure 18. It is notable that what this figure presenting has removed the plans which available to be used by all kinds of agents such as going to another vertex, and presenting particular plans that only used by their relevant agents. Similar as the plan distribution for different actions, most plans are for internal triggered events. Just "saboteur" which the only role of agents can do action "attack" use the plans with external trigger events and this kind of plans do not occupy a big part. Furthermore, saboteurs using most plans is in accordance with our offensive strategy. Explorers, repairers and inspectors use about the same amount of plans, however, few plans are for sentinels. This is reasonable in view of there is not any actions only available for sentinels.

Beliefs, desires and intentions are introduced above, but many agentspeak languages contain all these three. One of the reasons to choose Jason as programming language is that Jason can either provide a library of essential internal actions, or be straightforward extensible by user-defined internal actions, which are programmed in Java[5]. Implementing internal actions provides the means for programmers to do important things for BDI-inspired programming, such as checking and dropping the agent's own desires/intentions[6]. Besides the original internal actions like `.print` or `.send`, 32 internal actions defined by our own are programmed in Java. Most of these internal actions are devoted to control the map, such as calculating the distance between two agents or searching the nearest position to the current Vertex and so on. These internal actions run internally by the agent resulting in saving the commuting time.

Although Jason is a kind of new language for our team members, it is considered as the best agent speak language chosen for this multi-agent system after we trying to learning it. In our program, BDI model is clearly described, meanwhile beliefs, desires and intentions are arranged reasonably. In addition, the features of Jason such as user-defined internal actions are great used to improve communication process in our program. All our effort contributes in acquiring the second place in this contest. However, not familiar with Jason also makes some problems, some functions are not implemented completely, or some strategy is not the most reasonable. There is still room for improvement and many things need to be perfect.

6.2 Information Flow

Who gets what information how and when? How do we communicate with the server?

I think, this has been covered sufficiently in Section 5.3.2 and could be left out here.

6.3 Lifecycle of one Step

Maybe illustrate what happens within one step and how we prevent multiple actions to be executed in one step.

Write this part or throw it out if it is too trivial.

7 Discussion and Conclusion

Fill this part

7.1 Competition results[⊙,∘]

The competition took place on two dates (15th and 17th of September) and each team had to play three simulations against all other teams. Every simulation consisted of a total of 400 steps. The team with the higher overall score at the end received three points for their victory. Said overall score is the sum of all 400 step scores. The score per step is composed of points for zones plus achievement points. Since the strategy of team MAKo was to extensively buy upgrades for the so-called artillery agent, most of the earned achievement points were consumed and therefore did not count towards the step score. Figure 19 shows the progress of achievement points over time. As can be seen, the achievement points of team MAKo go up and down due to the buying actions whereas the points of the other team increase constantly. On first sight, one could assume that this strategy was a drawback because achievement points earned at some point count into every future step score. But compared to the number of points awarded for zones, the achievement points are only a minor fraction of the step score. As it can be seen in Figure 20 the spending of achievement points did not interfere dramatically with the overall score. It was worth spending the achievement points for the purpose of attacking and disturbing the other team. This was because the amount of potential zone points they would have earned without being attacked, would

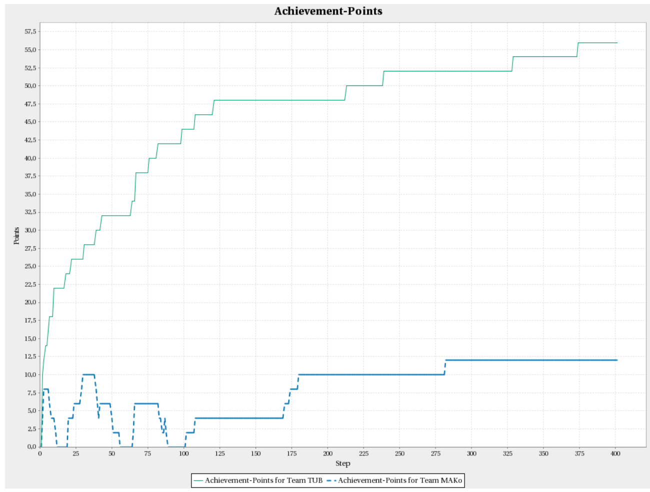


Fig. 19: Achievement points from the second match TUB against MAKo.

probably have been much higher than the amount of achievement points team MAKo spent for upgrades. At the end of the tournament team MAKo scored second with a total of 18 points. The winner of 2014 was, for the third time in a row, the team from the Federal University of Santa Catarina (UFSC). The final results are shown in Table 2. Statistics of all the individual games can be found

@manuelmittler: I just wrote "second match" as filler in Figure 19 and Figure 20. Which match was this actually from?

Pos.	Team name	Country	Score	Difference	Points
1	SMADAS-UFSC	Brazil	1180662 : 654624	526038	33
2	MAKo	Germany	617086 : 776868	-15782	18
3	TUB	Germany	904874 : 872399	32475	15
4	TheWonderbolts	Denmark	711001 : 1014669	-303668	15
5	GOAL-DTU	Denmark	653178 : 748241	-95063	9

Table 2: The results of the 2014 MAPC. Each team played three matches against every other team, and winning a match awarded 3 points.

add the references and link to the appendix

in the appendix.

Our team MAKo lost every second game against all opponents. This was due to the repairer agents not being able to repair. We were unable to figure out why this problem arose. But we found out that manually restarting our agents solved the problem. Unfortunately as a result, the knowledge about the graph acquired by the agents prior to restarting was reset. Consequently, the agents surveyed and probed redundantly. This behaviour was especially surprising as our agents

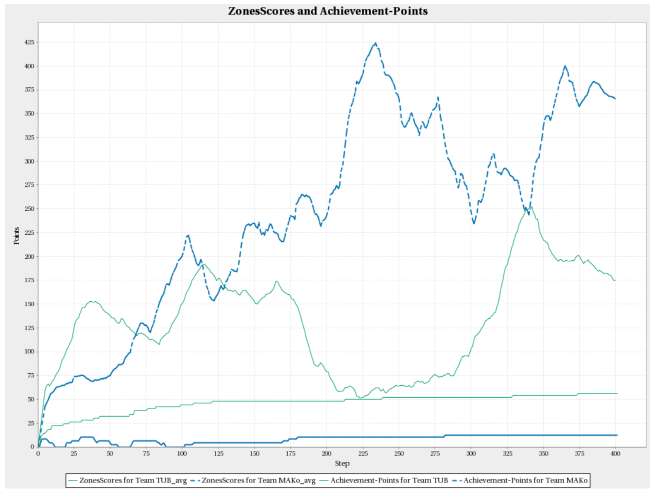


Fig. 20: Combined achievement and zone scores from the second match TUB against MAKo.

fully restarted automatically after each round and there were no such problems in all third rounds. The restarts were all manually supervised and showed no sign of failure at any time.

Disregarding this problem, our matches can be summarised as the follows. Our agents successfully explored the map, repaired disabled agents and attacked the opponent. Once our designated artillery agent had stopped upgrading, it did not have to move much anymore. Instead, it effectively attacked distant enemies and recharged mainly to attack afterwards again. This and the other saboteur agents which were always in search for enemy agents to attack helped disturb the zone building of the other teams. First, disabled agents were not able to build zones. Second, disabled agents needed to be repaired which could make the repairer agents temporarily unusable for zoning depending on the strategy the enemies implemented. If the enemy repairer agents moved towards disabled agents, they could break up a zone which they had been in earlier.

While monitoring the competition, we saw that zoning was subpar. Due to the fact that zones were broken up periodically, zones with a high value were sometimes discarded even when there was no need to. Furthermore, the asynchronous communication during zone finding did not work as well as hoped for. This was partly related to our agents being attacked and disabled by enemy agents. Agents which ought to form a zone unpredictably had to cancel their zoning availability and get repaired. Also, we did not implement an algorithm to detect the "end" of the graph, say the edges of the map. In the course of the matches, we saw that multiple maps favoured such an approach. Figure 21 shows the top left part of the graph from the third match of MAKo against GOAL-DTU. A single well-placed agent, here depicted as a green rectangle, could in this case create a zone over 25 vertices. Being able to detect these graph "ends" would

I'm not happy to talk about grpah "ends". Anybody has a suggestion for a better term? Maybe some explicit definition of leaves?

have enabled us to form greater zones with fewer agents than what our algorithm calculated. Nevertheless, the general idea regarding small zone forming proved

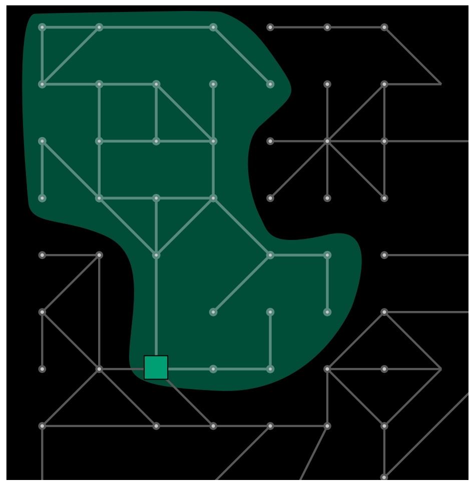


Fig. 21: One agent alone forming a zone over 25 vertices by exploiting the graph "ends".

to be a good choice. One big zone would have been easy to disturb. But having multiple small high value zones was quite effective in not providing the enemy with an easy target.

All in all, we are content with the results. Considering the short time we had until the competition without prior knowledge in this field, we managed to rank second. This is especially acceptable as the winning team won for the third time in a row and was only beat once due to technical problems.

7.2 Lessons learned$^{\odot/\circ}$

This final section gives an overview on the insights and knowledge we gained from this research lab. None of the MAKo team member had experience with Jason as a programming language before the research lab. Similarly, we had little experience with logic programming. Hence, getting into programming in AgentSpeak(L) was difficult at first. Most of the time, we felt that logic programming cost us more time than if we would have implemented the same in an imperative way.

Second, we found Jason to be quite slow when it comes to communication between agents. In our earlier approaches, agents often needed some information from others and could not continue with their reasoning until this information was given. Therefore, communication was a great bottleneck especially when exchanging information about the graph due to the amount of information. On account of this, letting agents communicate everything that they perceive while exploring the graph, to every other agent, was not an option. Consequently, we decided not to make agents share all their knowledge with each other. Instead, the most complete information about the graph should be available in one place. Our first approach here was to introduce a cartographer agent as illustrated in Section 5.3.1. It was an additional agent in the background which was sent all the information about the map which all 28 agents perceived. The drawback of this approach revealed when it came to querying the cartographer agent for information. Agents needed to do this frequently for instance when they wanted to know if a vertex was already surveyed or how a given vertex could be reached. As mentioned before, processing the received messages is quite slow. Together with calculating paths multiple times, the cartographer agent was not able to process messages in time and agents were idle waiting for replies. As described in Section 5.3.2, dividing the cartographer's work load onto our so-called node agents did not solve our performance issues. In the end, we settled for reimplementing these ideas imperatively as the JavaMap module.

Another issue arose initially during the contest. If a term in Jason contains a dash, it is interpreted as an arithmetic expression. We observed this during our first match against a team that had a dash in its name. So instead of handling `GOAL-DTU1` as a literal identifying an enemy agent, our agents tried to subtract `DTU1` from `GOAL` which lead to exceptions. The result was that every reasoning which considered the name of an enemy agent failed. Accordingly, we lost all three matches against GOAL-DTU. Luckily, the GOAL-DTU team agreed on a rematch on the subsequent matchday leaving us enough time to solve this problem. Furthermore, the organisers rescheduled the matches. Else, we would have played against SMADAS-UFSC on the same day and would have lost as well.

When we started programming, we found the mixture between logic programming for agents following the concept of BDI and imperative programming with Java appealing. Over the course of our development though, we came to knew that all our major problems were somehow connected to Jason. In the end, we maybe would have profited from starting from scratch rather than working with Jason. But all in all, we definitely learned a lot in terms of logic programming, multi-agent systems and their development and of course Jason in particular. Moreover, we are happy with our result in the competition and the gathered experience.

This section is all negative. But we also experienced/learned positive things. Add them.

References

[1] Tobias Ahlbrecht et al. "Multi-Agent Programming Contest 2013". In: *Engineering Multi-Agent Systems*. Springer Berlin Heidelberg, 2013, pp. 292–318.

[2] Tobias Ahlbrecht et al. *Multi-Agent Programming Contest Scenario Description*. Tech. rep. TU Clausthal, 2014.

[3] *BDI Architecture*. [Online; accessed 31-October-2014].

[4] Fabio Bellifemine et al. “JADE—a java agent development framework”. In: *Multi-Agent Programming*. Springer, 2005, pp. 125–147.

[5] Rafael H. Bordini and Jomi F. Hubner. *A Java-based interpreter for an extended version of AgentSpeak*. Tech. rep. 2007.

[6] Rafael H. Bordini and Jomi F. Hubner. *An Overview of Jason*. Tech. rep. 2006.

[7] Rafael H. Bordini and Jomi F. Hübner. “BDI Agent Programming in AgentSpeak Using Jason”. In: *Computational Logic in Multi-Agent Systems*. Springer, 2005, pp. 143–164.

[8] Rafael H. Bordini, Jomi F. Hübner, and Renata Vieira. “Jason and the Golden Fleece of agent-oriented programming”. In: *Multi-agent programming*. Springer, 2005, pp. 3–37.

[9] Rafael H. Bordini, Jomi Fred Hübner, and Michael Wooldridge. *Programming multi-agent systems in AgentSpeak using Jason*. Vol. 8. John Wiley & Sons, 2007.

[10] Rafael H. Bordini et al. “AgentSpeak (XL): Efficient intention selection in BDI agents via decision-theoretic task scheduling”. In: *Proceedings of the first international joint conference on Autonomous agents and multiagent systems: part 3*. ACM, 2002, pp. 1294–1302.

[11] Craig Boutilier et al. “Decision-theoretic, high-level agent programming in the situation calculus”. In: *AAAI/IAAI*. 2000, pp. 355–362.

[12] Michael E. Bratman, David J. Israel, and Pollac Martha E. “Plans and resource-bounded practical reasoning”. In: *Computational Intelligence* 4 (1988), pp. 349–355.

[13] Lars Braubach, Alexander Pokahr, and Kai Jander. *BDI User Guide: Chapter 3 Agent Specification*. [Online; accessed 22-October-2014]. 2010.

[14] Lars Braubach, Alexander Pokahr, and Kai Jander. *BDI User Guide: Chapter 3 Agent Specification*. [Online; accessed 22-October-2014]. 2010.

[15] Lars Braubach, Alexander Pokahr, and Winfried Lamersdorf. “Jadex: A short overview”. In: *Main Conference Net. ObjectDays*. Vol. 2004. 2004, pp. 195–207.

[16] FIPA TC Communication. *FIPA ACL Message Structure Specification*. [Online; accessed 24-October-2014]. 2002.

[17] James P. Delgrande. “Revising by an Inconsistent Set of Formulas”. In: *IJCAI’11*. 2011, pp. 833–838.

[18] Jerzy Tiurzyn Dexter Kozen. “Logics of program”. In: *In Jan van Leeuwen, editor, Handbook of Theoretical Computer Science* B (1990), pp. 789–840.

[19] Herbert B. Enderton. *A Mathematical Introduction to Logic*. Academic Press, San Diego, 1972.

[20] Victor Fernández et al. “Evaluating Jason for Distributed Crowd Simulations.” In: *ICAART (2)*. 2010, pp. 206–211.

[21] Thom Frühwirth. “Theory and practice of constraint handling rules”. In: *The Journal of Logic Programming* 37.1-3 (1998), pp. 95–138.

[22] Michael Georgeff et al. “The Belief-Desire-Intention Model of Agency”. In: *Intelligent agent V* (1999), pp. 1–10.

[23] Alejandro Guerra-Hernandez, Amal El Fallah-Seghrouchni, and Henry Soldano. *Learning in BDI multi-agent systems*. Springer Berlin Heidelberg, 2004, pp. 218–233.

[24] Patrick J. Hayes. *The Frame Problem and Related Problems on Artificial Intelligence*. Stanford University, 1971.

[25] Jomi Fred Hübner and Jaime Simão Sichman. "SACI: Uma Ferramenta para Implementação e Monitaração da Comunicação entre Agentes." In: *IBERAMIA-SBIA 2000 Open Discussion Track*. 2000, pp. 47–56.

[26] Sarit Kraus, Katia Sycara, and Amir Evenchik. "Reaching agreement through argumentation: a logical model and implementation". In: *Artificial Intelligence* 104 (1998), pp. 1–69.

[27] Saul A. Kripke. "Semantical analysis of modal logic I: Normal modal propositional calculi". In: *Zeitschrift für Mathematische Logik und Grundlagen der Mathematik* 9 (1963), pp. 67–96.

[28] Bo Leuf and Ward Cunningham. *The Wiki way: quick collaboration on the Web*. 1st ed. Boston, MA (USA): Addison-Wesley, 2001. ISBN: 9780201714999.

[29] Hector J. Levesque et al. "GOLOG: A logic programming language for dynamic domains". In: *The Journal of Logic Programming* 31.1 (1997), pp. 59–83.

[30] Fangzen Lin and Ray Reiter. "State constraints revisited". In: *Journal of logic and computation* 4.5 (1994), pp. 655–677.

[31] Eleni Mangina. *Review of Software Products for Multi-Agent Systems*. Report. 2002.

[32] Yves Martin and Michael Thielscher. "Addressing the qualification problem in FLUX". In: *KI 2001: Advances in Artificial Intelligence*. Springer, 2001, pp. 290–304.

[33] John McCarthy and Patrick Hayes. *Some philosophical problems from the standpoint of artificial intelligence*. Stanford University USA, 1969.

[34] Roberto Montagna et al. "BDI^{ATL}: An Alternating-time BDI Logic for Multiagent Systems". In: *EUMAS'05*. 2005, pp. 214–223.

[35] Munindar P. Singh, Anand S. Rao, and Michael P. Georgeff. "Formal methods in DAI: Logic-based representation and reasoning." In: *Multiagent Systems A Modern Approach to Distributed Artificial Intelligence*. The MIT Press, Cambridge, Massachusetts, 1999, pp. 331–376.

[36] Fiora Pirri and Ray Reiter. "Some contributions to the metatheory of the situation calculus". In: *Journal of the ACM (JACM)* 46.3 (1999), pp. 325–361.

[37] Alexander Pokahr, Lars Braubach, and Winfried Lamersdorf. "Jadex: A BDI reasoning engine". In: *Multi-agent programming*. Springer, 2005, pp. 149–174.

[38] Anand S. Rao. "AgentSpeak(L): BDI Agents speak out in a logical computable language". In: *MAAMAW '96 Proceedings of the 7th European workshop on Modelling autonomous agents in a multi-agent world : agents breaking away: agents breaking away*. 1996, pp. 42–55.

[39] Anand S. Rao. “AgentSpeak(L): BDI agents speak out in a logical computable language”. In: *Agents Breaking Away*. Springer, 1996, pp. 42–55.

[40] Anand S. Rao and Michael P. George. “BDI Agents: From Theory to Practice”. In: *AAAI*. Vol. 6. 1995, pp. 312–319.

[41] Raymond Reiter. “The frame problem in the situation calculus: A simple solution (sometimes) and a completeness result for goal regression”. In: *Artificial intelligence and mathematical theory of computation: papers in honor of John McCarthy* 27 (1991), pp. 359–380.

[42] Sebastian Sardina, Lavindra de Silva, and Lin Padgha. “Hierarchical Planning in BDI Agent Programming Languages: A Formal Approach”. In: *AAMAS '06 Proceedings of the fifth international joint conference on Autonomous agents and multiagent systems*. 2006, pp. 1001–1008.

[43] Stephan Schiffel and Michael Thielscher. “Multi-agent FLUX for the gold mining domain (system description)”. In: *Computational Logic in Multi-Agent Systems*. Springer, 2007, pp. 294–303.

[44] Stephan Schiffel and Michael Thielscher. “Reconciling situation calculus and fluent calculus”. In: *AAAI*. Vol. 6. 2006, pp. 287–292.

[45] Munindar P. Singh. “A critical examination of the Cohen-Levesque theory of intentions”. In: *In Proceedings of the 10th European Conference on Artificial Intelligence* (1992), pp. 364–368.

[46] Munindar P. Singh. “A customizable coordination service for autonomous agents”. In: *In Proceedings of the 4th International Workshop on Agent Theories, Architectures and Languages (ATAL)* (1997).

[47] Michael Thielscher. “FLUX: A logic programming method for reasoning agents”. In: *Theory and Practice of Logic Programming* 5.4-5 (2005), pp. 533–565.

[48] Michael Thielscher. “From situation calculus to fluent calculus: State update axioms as a solution to the inferential frame problem”. In: *Artificial intelligence* 111.1 (1999), pp. 277–299.

[49] Michael Thielscher. *Reasoning Robots: The Art and Science of Programming Robotic Agents*. en. Springer Science & Business Media, Jan. 2006. ISBN: 9781402030697.

[50] Renata Vieira et al. “On the formal semantics of speech-act based communication in an agent-oriented programming language.” In: *J. Artif. Intell. Res.(JAIR)* 29 (2007), pp. 221–267.

[51] Wikipedia. *Belief revision*. [Online; accessed 31-October-2014].

[52] Michael Wooldridge. *Multiagent systems: a modern approach to distributed artificial intelligence*. Ed. by Gerhard Weiss. Library of Congress Cataloging-in-Publication Data, 1999, p. 619.