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, Campus Koblenz

The symbols after the authors’ names indicate who wrote which section. When there are multiple symbols, it means that the first part of author(s) wrote it and the second part significantly corrected/updated and proofread it. E.g. ⋆/⊙,†/▲ would indicate ⋆ and ⊙ are the authors and † and ▲ corrected it.

Table of Contents

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 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.

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.

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 [1].

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

2.2 Agents Roles and Actions$^{\star,\circ}$

This section presents the basic behaviour of our agents given the actions that all agents share. Every agent team in the MAPC Scenario consists of 28 agents. These agents are divided into five roles: the Explorer agent, the Repairer agent, the Saboteur agent, the Sentinel Agent and the Inspector agent. As shown in Figure 1, each role is given different values in their attributes of maximum energy, health or visibility range. Moreover, the saboteur agent has a strength value, because it is the only agent which can attack enemy agents. There are six agents of each role except the Explorer agent role of which there are four agents. Except for the Sentinel role, all other roles allow their corresponding agents to execute some actions exclusive to this role. Every action has a minimal chance to fail, regardless of being an exclusive action or an special action which is only available

Explorer	Actions:	`skip`, `goto`, `probe`, `survey`, `buy`, `recharge`
	Energy:	12
	Health:	4
	Strength:	0
	Visibility range:	2
Repairer	Actions:	`skip`, `goto`, `parry`, `survey`, `buy`, `repair`, `recharge`
	Energy:	8
	Health:	6
	Strength:	0
	Visibility range:	1
Saboteur	Actions:	`skip`, `goto`, `parry`, `survey`, `buy`, `attack`, `recharge`
	Energy:	7
	Health:	3
	Strength:	4
	Visibility range:	1
Sentinel	Actions:	`skip`, `goto`, `parry`, `survey`, `buy`, `recharge`
	Energy:	10
	Health:	1
	Strength:	0
	Visibility range:	3
Inspector	Actions:	`skip`, `goto`, `inspect`, `survey`, `buy`, `recharge`
	Energy:	8
	Health:	6
	Strength:	0
	Visibility range:	1

Fig. 1: The different agent roles in the MAPC scenario [1].

to the specific agent. Some actions can be executed on distant agents. They are called ranged actions. The drawback of ranged actions is that they have a much higher failing chance if the target of the action is further afar. First, the actions all agents are able to execute will be presented. After that the exclusive actions of the agents will be described. Those actions are `skip`, `goto`, `survey`, `buy` and `recharge`.

skip The `skip` action should be used as a last resort if there is nothing else for an agent left to do. This action's only purpose is to tell the server that an agent did not time out but was not interested in executing a different action. If the `skip` action is executed when an agent could have e.g. recharged instead, it can be seen as a wasted step for this particular agent.

goto The `goto` action is used to traverse over edges from one vertex to another adjacent vertex. Said traversing is only possible when the costs of the edge to traverse are lower than or equal to the energy the agent currently has. Else, the execution of the method will fail. By successfully executing the `goto` action, the current energy of the agent is reduced by the traversing costs of the edge.

survey When the ability `survey` is executed, weights of edges in the visibility range of the agent are retrieved. The count of edge weights an agent gets as percept is determined randomly based on the visibility range of the agent.

buy With the action `buy` an agent is able to upgrade its values like maximum health and visibility range. Saboteur agents can furthermore increase their strength through this action.

recharge If an agent has a low energy level the ability `recharge` fills up the energy of the agent. By each `recharge` action the current energy is recharged by half of the maximum energy.

The role specific actions are explained in the following.

parry The `parry` action can be used by repairer agents, sentinel agents and saboteur agents. By using the action an incoming attack can be fully neglected and the health of the agent is preserved.

inspect Only inspector agents can use the `inspect` action. Inspecting is a ranged action. The action reveals the inner stats and details of the targeted agent. By inspecting it is possible to find out which role an enemy agent has. Inspecting is not needed to be able to tell if an agent is disabled as one could assume from the action's name. This is because the `visibleEntity` percept includes the agent's current state.

repair Repairing is an ranged action which is unique to repairer agents. The `repair` action repairs an agent of the same team. Repairing can not be executed on the agent itself.

probe Explorer agents are the only agents which can reveal the value of vertices with its `probe` action. As long as a vertex is not probed the vertex value is calculated as 1. Probing is an ranged action and thus can be executed on distant vertices.

`attack` The `attack` action can only be executed by saboteur agents. It is used to attack an enemy agents and reduce its health by a specific amount until it gets disabled. Attacking is a ranged action.

The use and embodiment of these unique abilities into the agent role specific behaviour is explained in Section 3.2.

2.3 Agent Programming Concepts

Write this part!

2.3.1 BDI$^{\blacktriangle,\diamond/\circ}$

Using cognitive modelling techniques for agent development enable autonomous behaviour and reasoning for automated problem solving. This also allows the analysis of real world behaviour through simulations with agents. Problems can hence be automatically solved through self-organised groups of agents and new knowledge can be drawn from such simulations. The beliefs, desires and intentions (short: *BDI*) model, is a widespread software model for developing intelligent agents situated in complex and dynamic environments. There, the basic characteristics of an agents' mental state are expressed through beliefs, desires and intentions which are explained in more detail later in this section. The BDI logic system is quite easy to implement in software agents and to some extent promises human-like reasoning behaviour. Hence, it has been widely used in the field of artificial intelligence in computer science. This section begins with the roots of the BDI model by mentioning the scientific papers that led to it. It then continues with an explanation of practical reasoning after which the components of the BDI model are presented in greater detail.

In 1987, Bratman [14] discussed the relationship between beliefs, desires, intentions and actions as well their roles in agent behaviours. This can be seen as the introduction of the BDI model which was revised in 1991 by Rao and Georgeff [44]. They formalised the model to a first order logic and treated beliefs, desires and intentions as three modal operators while giving intentions equal importance compared to beliefs and desires. Rao and Georgeff then continued to apply this theoretical foundation to concrete BDI agents, applying them in an airline traffic management application [43]. Nowadays, BDI agents are deployed in high technology industrial areas such as space shuttle development. For example, PRS [29] and dMARS [22] are both BDI-based systems for the reaction control system of the NASA Space Shuttle Discovery.

The BDI model originated from a philosophical background as it was introduced in Bratman's theory of practical reasoning [46]. Practical reasoning involves two important processes: deciding what goals should be achieved, and how they should be achieved. The former process is known as *deliberation*, the latter as *means-ends reasoning* [55]. Means-ends reasoning is a method to make plans for achieving goals based on current states. A *plan* is a strict order of actions needed to be executed to achieve a goal. The idea of means-ends reasoning is to reduce the difference between the current state and the state where a goal is achieved. For this purpose, the method is applied recursively. When an agent is placed in an environment, it should autonomously decide what to do and how to do it. In general, an agent could execute many different actions but maybe only a few of them would have a desired effect on the environment. Various external properties can have influence on the feasibility of achieving the agent's goals. The deliberation process filters what options are actually possible in the current state. It then determines which of these options will become intentions. For example, if you are thirsty and standing in a supermarket, then you might be faced with the decision to choose a drink. There can be a lot of options like wine, beer, water, lemonade or juice. However, picking up a bottle of wine or beer is not allowed in Germany for people under the age of 16. 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. If your intention is to buy a bottle of water, you may then plan to go to the shelf with water on it, reach for such a bottle, take it to the checkout counter and pay for it. Finally, you have to execute this plan to buy the bottle of water.

The BDI model, as an applicable theory of practical reasoning, consists of three components which are beliefs, desires and intentions. These are subsequently explained each.

Beliefs represent the informational part of the agent [43] and are 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 [43]. Beliefs model the agent's look at the world. They can express information about the environment, other agents or the agent itself. An agent may update its beliefs at any time. Agents receive new information from the perception of the environment and the execution of intentions. An agent can use sensors to perceive the environment and store their output as beliefs. Beliefs are not the same concept as knowledge. Knowledge is the realisation of a fact whereas a belief models "knowledge" which is believed by the agent. Beliefs do not necessarily have to be true from a global perspective. But from the agent's perspective, they are. Hence, beliefs can be seen as local knowledge. Likewise, beliefs can be global knowledge when they are true in a global sense.

Desires represent the motivational part of the agent [43]. In other words, desires represent objectives or situations that the agent would like to accomplish or bring about. Desires can be but do not need to be achieved. Multiple desires can be inconsistent with each other and the agent does not need to know the means of achieving these desires. These desires are filtered within the agent's deliberation process. Therefore, a subset of both consistent and achievable desires called *goals* are chosen by the agent [55]. For example, sleeping and working may be both one's desires, but they can not both be one's goals at the same time because they conflict with each other.

Intentions are desires or actions that an agent committed itself to achieve [26]. Their meaning is stronger than that of desires. Desires are merely wishes that may be achieved or may be not, but intentions are decided to be achieved to a reasonable extent. Michael Wooldridge [55] concluded four functions of intentions in practical reasoning:

- Intentions driving means-ends reasoning means that intentions have decisive influences on the actions which the agent will execute. Agents are expected to determine ways of achieving intentions.
- Intentions constrain future deliberation. This means that options which would be in conflict with an already chosen intention will not be considered.
- Intentions persist means that intentions will not be given up unless there is a rational reason to do so. This is important because if an agent immediately dropped its intentions without devoting resources to achieving them, it will never achieve anything [55]. Hence, intentions are committed desires which can not be easily abandoned. A reason to drop an intention nevertheless could be e.g. when an external event made the intention impossible or a lot more difficult to achieve.
- Intentions influence beliefs upon which future practical reasoning is based. This expresses that an intention is believed to be eventually achieved under “normal circumstances”. Practically, after deciding on an intention, the agent will add the intention as a belief. This allows the agent to plan for the future on the assumption that the intention will be achieved.

Intentions play an important role in the BDI model as they lead to actions and influence future beliefs and intention selection.

Beliefs, desires and intentions are the foundation of BDI agents. Yet, further components are needed to build the connection between these three components and thereby implement BDI agents. Naturally, the architecture of BDI agents differs in detail depending on their tasks. However, they all share a core architecture which is depicted in Figure 2. Its components are explained in the following paragraphs.

The sensors of a BDI agent perceive the environment and convert the perceptions P to signals as inputs for the *belief revision function*. This function collects the external perceptions as well as the beliefs which are already stored in the beliefs. It then processes the information and updates the beliefs accordingly. The belief revision function prefers minimal change over modifying a lot and hence tries to preserve as much information as possible [5]. While processing the new and old information, the belief revision function will also solves inconsistencies, e.g. when an agent perceives information which contradicts a belief it has.

The *belief set*’s data may be modelled as sentences, rules or some other manifestations. Formally, we denote the belief set with the letter B. In the *AGM approach* (named after their proponents, Alchourrón, Gärdenfors, and Makinson) [4], an agent’s beliefs are modeled by a deductively closed set of formulas called a *belief set* [20]. This approach is broadly used in belief revision research. It makes several postulates for one revision operator which mapping each belief set and one sentence of beliefs to generate new beliefs. Meanwhile, these postulates allow the belief revision process to retain as many previous beliefs as possible to reduce the amount of change.

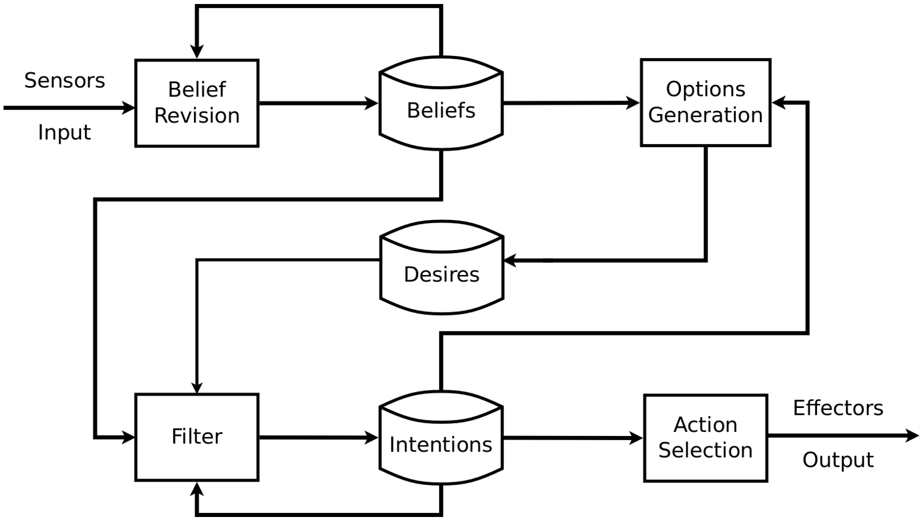


Fig. 2: Brief BDI architecture [7]. Squares symbolise functions whereas the elliptic shapes symbolise data sets. Arrows indicate data transmission.

The *option generator* commits a list of desires into the *desire set* D. It determines the desires depending on the agent's current beliefs and current intentions. The desire set contains desires, which express the agent's wishes it wants to achieve. Next, the filter determines the agent's intentions depending on the current beliefs, desires, and intentions. Only achievable and rational desires will then become intentions.

The *intention set* I stores the agent's current focus, which is accompanied by a concrete plan. Once an intention is adopted, hence stored in the intention set, the agent will pick and commit to one intention in order to achieve it. The *action selection function* determines the actions to perform depending on the current intentions. This function draws its knowledge from a plan library which is not depicted in Figure 2. Finally, the matching plan is inspected for the next action A to be executed by the agent.

Table 1 summarises the BDI agent architecture as described by Michael Wooldridge [55]. The process of $B \times P \to B$, $B \times I \to D$ and $B \times I \times D \to I$ belongs to deliberation. They are deliberated gradually, so the range of intentions is narrowed down. This happens especially due to the filter function which operates on all of the datasets. $I \to A$ can be treated as the process of means-ends reasoning as the difference between the committed intention and the current situation decreases. B, D and I are connected through functions instead of being connected directly. This is because they are treated simply as functionless databases on which algorithms may work.

Although the BDI model has been introduced about 30 years ago, it is still a field of on-going development. The BDI model restricts itself to the three main components: beliefs, desires and intentions. In some situations, not all the three attributes are needed. However, for some distributed multi-agents, only three attributes may not be sufficient. One problem here is for example that the BDI model does not capture communication between agents or interaction between them.

Beliefs, desires and intentions were introduced in this section. The BDI agent belongs to the kind of intelligent agents which are autonomous, computational entities. They follow the practical reasoning theory. Different BDI implementations show different architectures, but the core of these agents is most of the time built by beliefs, desires and intentions.

With an increasing number of BDI applications, more challenges come up too. One of such challenges is the correctness of agents' behaviour. It is described in the following Section 2.3.2.

2.3.2 Formal Methods$^\diamond$

As was mentioned in Section 2.3.1, usage of BDI agents brings up several challenges. One of the important challenges of multi-agent systems is to make 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

Component	**Meaning**	**Formalisation**
Belief set	Information about the environment which the agent is located in	B
Belief revision function	Determines a new set of beliefs depending on perceptual inputs and the agent's current beliefs	$B \times P \rightarrow B$
Options generation function	Determines desires depending on the agent's current beliefs and current intentions	$B \times I \rightarrow D$
Desire set	Are states which an agent wishes to bring about	D
Filter function	Determines the agent's intentions depending on current beliefs, desires, and intentions	$B \times D \times I \rightarrow I$
Intention set	The agent's current focus	I
Action selection function	Determines an action to perform depending on current intentions	$I \rightarrow A$

Table 1: Meaning and formalisation of the components of the BDI agent architecture.

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. [39]

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 [23].

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 $\Box$ 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 [31].

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 [21].

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.

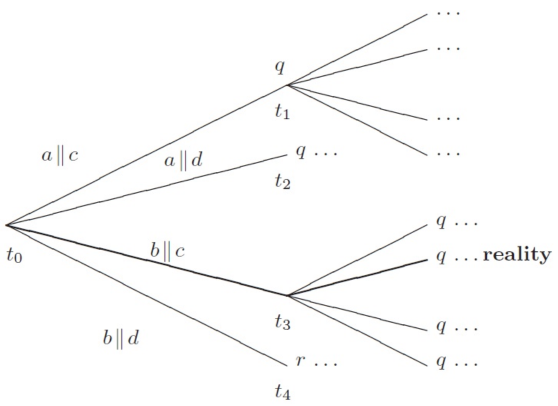
Discrete (or *dense*): discrete steps (like natural numbers) or always having intermediate steps (like real numbers).

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 [39].

Fig. 3: An example branching structure of time [39].



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 3 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 [49]:

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.

```
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.
```

Listing 1.1: An abstract BDI interpreter [49].

As was mentioned above, options are usually represented by plans. Plans consist 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.
```

Listing 1.2: Option generation for a BDI interpreter [49].

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.
```

Listing 1.3: Option deliberation for BDI interpreter [49].

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 [50] 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 $\bigodot$, 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.

2.3.3 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 do not 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 [30]. 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 [30]. 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 the 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 held in the agent's current mental state.

In their work Kraus et al. [30] point out six types of arguments, which can be used during negotiation:

- An appeal to prevailing practice.
- A counterexample.
- An appeal to past promise.
- An appeal to self-interest.
- A promise of a future reward.
- 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 other agents' actions in a 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 agent is not a memoryless agent. This type of argument is a sort of a reminder to the previously given promise to execute an action in some particular situation. The algorithm of checking if this argument is applicable 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 a future reward and that reward was the currently intended action.

An appeal to self-interest is a type of argument that convinces the opponent that the performed action will serve towards fulfilling one of its desires. This argument cannot be applied to a knowledgeable 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 knowledgeable 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 appears in the list. If such an 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. In 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 considering joint desires, trying to find one that can be satisfied with help of the agent; like in the self-interest argument generate a list of actions that lead to the desire fulfillment; out of the resulting list of actions 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 for executing the requested action right now.

A threat to perform an action that contradicts with an opponent's plans in case 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 the agent's desire set, first considering 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 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. [30] 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 used.

Request evaluation rules define how incoming requests will be processed by the agent. 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 was mentioned above, during the request evaluation agents' beliefs, desires, intentions or goals can be changed. An example of such changes in our project can be a Saboteur agent switching to zone defending after negotiation with other Saboteur agents: the beliefs about the zone it 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. Zone defending and roles of agents in the zone are explained more in detail in Section 3.5.3. 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 [11]. In this method all the agents participating in negotiation 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. We use alphanumeric sorting on agent names as a tie-breaking strategy in the case where all arguments appear to be equal. The agent with the winning bid then fulfills the request: adopts a certain role or performs a prescribed action.

2.4 Agent Programming Languages°

Currently, multiple 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, AgentLink [35]. We investigated several agent programming languages, for their suitability for the "Agents on Mars" scenario. Our goal was to determine which specialised 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 unrelated to the "Agents on Mars" scenario and are kept simple for ease of 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 2.4.1 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 2.4.2. It also helps in understanding the subsequent Section 2.4.3 which summarises the main concepts of FLUX. FLUX is another logic programming language which was partly motivated by the flaws of GOLOG. Section 2.4.4 introduces a Java-based agent programming language. After that, AgentSpeak(L) is presented in Section 2.4.5 which is another logic programming language. Jason is an interpreter for this language and is discussed in Section 2.4.6. The section focuses mainly on the extensions that Jason adds to AgentSpeak(L). The final Section 2.4.7 considers the previously presented agent programming languages and explains our decision for choosing Jason.

2.4.1 Situation Calculus°,†

This section gives a short summary of the situation calculus, which was first introduced by McCarthy and Hayes [37]. The situation calculus is mainly a first-order logic but also uses second order logic to encode a dynamic world [33]. It is a theoretical concept and is consequently not applicable to multi-agent scenarios without any concrete implementation. 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* [37, 13]. 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 [48, 33]. There can only be one initial situation as it models the situation before any action has been executed [40].

Fluents can be evaluated to return a result. As they are situation dependent, the evaluation result may change over time. Fluents are distinguished into *relational fluents* and *functional fluents* [33]. Relational fluents can hold in situations. Their evaluation hence may return either true or false [13]. An example is given in Equation 1. It expresses whether or not the agent p has a cup of coffee in situation s.

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

Functional fluents return values instead [33]. As an example, a fluent $\mathit{location}(p, s)$ 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* [34]. This is expressed by the predicate $\mathit{Poss}(a, s)$, 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.

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

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

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

In Equation 3, it is unclear whether other fluents are affected by the action execution. For example, reasoning about $\mathit{location}(p, s')$ would not be possible with $\mathit{do}(\mathit{pourCoffee}(p, s)) \rightarrow s'$. This is called the *frame problem* (cf. Hayes [27]). 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 [45]. 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.

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

$F(\mathit{do}(a, s))$ 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:

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

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 coffee.

Although the situation calculus contains further concepts, this quick introduction should suffice to get an understanding of it. Section 2.4.2 shows an implementation of these concepts into an agent programming language.

2.4.2 GOLOG$^{\circ,\dagger}$

This section gives a summary of the logic programming language GOLOG. If not further specified, all information except for the examples is taken from Levesque et al. [33] 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 has 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:

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

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

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 coffee for some agent which is lacking coffee. After completion of its coffee-pouring task, the robot sings. Equation 8 allows the robot to non-deterministically choose an agent p to pour coffee for 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:

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

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

GOLOG is *regression-based.* This means that deciding whether a property holds in a situation requires looking at all prior changes to it. For this, the property must be traced back to the initial situation by looking at all action executions and their effects. Depending on how often this needs to be done and how many actions have already been executed, reasoning can take a long time.

As shown, GOLOG transfers the earlier presented concepts of the situation calculus into a logic programming language. It enables the comfortable use of control flow statements like `while`-loops. These statements are macros which a GOLOG interpreter expands into solvable formulas. Levesque et al. [33] provide such an interpreter written in Prolog in their paper. The next section introduces another logic programming language FLUX which is based on a different calculus. In the later Section 2.4.7, critical differences between FLUX and GOLOG for applying them to a multi-agent scenario are discussed.

2.4.3 FLUX$^{\circ,\dagger}$

This section gives a summary of the logic programming language FLUX. Except for the examples and if not specified otherwise, the information of this section is taken from Thielscher [51] who first introduced FLUX. The language uses 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 and states in general can be incomplete. This is important for modelling agents discovering unknown environments. Their knowledge of the environment will then become more and more complete over time.

The frame problem in the fluent calculus is solved through *state update axioms* as described by Thielscher [52]. 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.* The performance over time therefore does not worsen much as determining whether a property currently holds is only a matter of looking it up in the state.

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 [25]. 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 to FLUX. This is done by using the logic programming language Prolog in which FLUX is typically implemented (cf. [53, 36]). 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 1.4: Defintion of the `sing`-action.

Listing 1.4 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 1.5: Definition of the `pourCoffee`-action

The `pourCoffee` action is defined similarly in Listing 1.5. 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 1.6: Main method which either tells the robot to sing or to pour coffee.

Listing 1.6 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 coffee and call the main loop again. Line 16 ensures that Prolog will return the false-value `No` if neither of both actions gets triggered at some point.

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

Listing 1.7: Initial configuration.

The initial configuration in Listing 1.7 is comparable to Equation 9 but due to Prolog interpreting from top to bottom, the result will be `Z = [hasCoffee(john), hasCoffee(jane)]`.

Besides GOLOG and FLUX there are other agent programming languages which are not logic programming languages. The next subsection presents Jadex, which allows imperative programming by using Java.

2.4.4 Jadex$^{\odot,\circ}$

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 middleware. The most commonly used middleware for Jadex is the *Java Agent Development Framework* (short: JADE) [6]. Jadex integrates agent theories through object-oriented programming in Java and XML descriptions. What is meant by autonomous and proactive is, that an agent acts without human intervention or guidance. The term social means that agents should work together to accomplish their goals. 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 organisation that is dedicated to promoting the industry of intelligent agents by openly developing specifications supporting interoperability among agents and agent-based applications.” ([19]) A FIPA ACL message has a certain structure and contains five mandatory parameters. The first is the type of the communicative act which tells the recipient how to react on the message. Next, the participants in the communication, namely sender and receiver, are specified. The third parameter contains the actual content of the message. As a fourth parameter, there is the description of the content which denotes the language of the content and its encoding. The last parameter is called control of conversation and contains, amongst other things, a conversation identifier. Its purpose is to associate a message with a specific conversation and to chronologically sort it in said conversation.

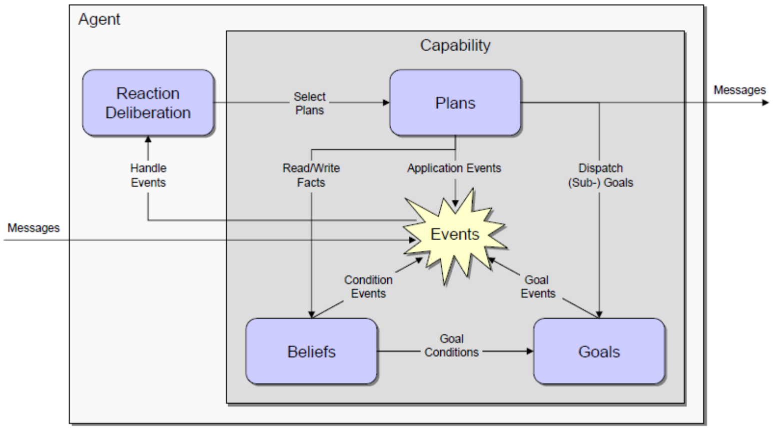


Fig. 4: Jadex abstract agent [41]

Figure 4 depicts an abstract view of 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.

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 [15], 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 an internal state of the agent, regarding the availability of some information the agent wants to know about. For goals of type *maintain*, an agent keeps 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 by 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. [17])

Jadex is not based on a new agent programming language but chooses a hybrid approach instead. It distinguishes 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

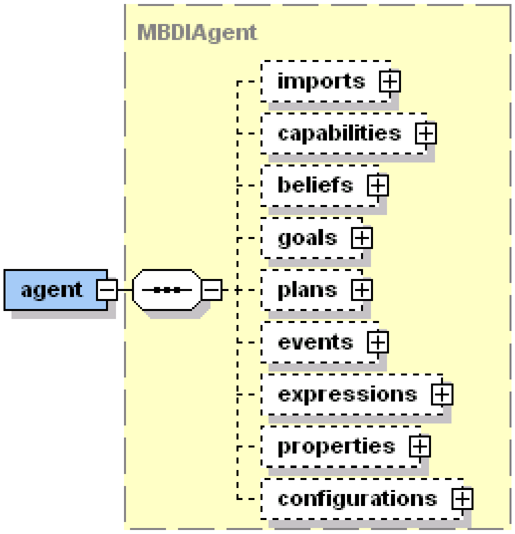


Fig. 5: Jadex top level ADF elements [16]

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. It must at least implement the abstract `body()` method which is invoked after plan instantiation.

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

   public ServeCoffeePlanB1() {
       // Initialisation code.
   }

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

Listing 1.8: Jadex Java class

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.

```xml
<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>
```

Listing 1.9: Jadex XML file

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 in the agent definition file to let the agent know which 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 on 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 [41].

In summary, 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 debugging during development. For example, it comes 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.

2.4.5 AgentSpeak(L)$^{\circ,\dagger}$

This section provides an overview of the general concepts of the logic programming language AgentSpeak(L). The language was developed by Rao [42]. Except for the examples, this section takes its information from the cited paper. The idea behind AgentSpeak(L) was to make the theoretic concept of BDI agents usable in practical scenarios.

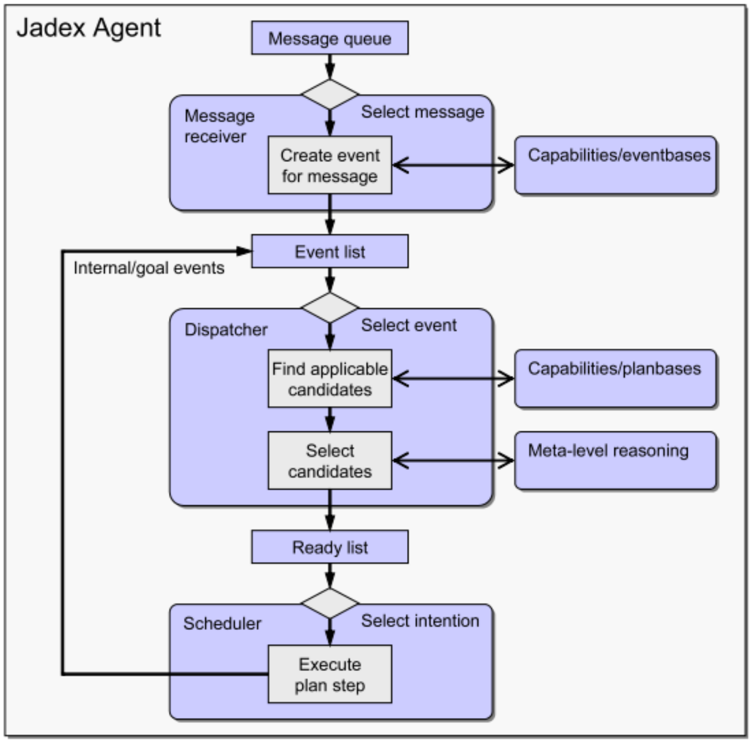


Fig. 6: Jadex execution model [41]

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 lower-case initial letter are constants.

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

Listing 1.10: Initial beliefs.

Listing 1.10 shows the initial beliefs of an agent in the example we just introduced. 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 an agent are called its *belief base* [10].

Goals can be divided 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, a given 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 coffee. Hence, this expression will evaluate to true or false depending on the agent's knowledge. Achievement goals are comparable to desires. Listing 1.11 shows the initial achievement goal which express that the agent wants to have served everyone coffee.

```
!servedCoffee.
```

Listing 1.11: Initial goal.

Events are introduced to allow agents to react on changes in their own knowledge or the world. If they receive new information from other agents or as a perception of the environment, an *external event* is triggered. *Internal events* are triggered when agents add knowledge or goals by themselves. Events in general 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 intentions for BDI agents. They are stored as a set of intentions. The set of plans generally known to an agent is called the *plan library* [10]. 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 1.12 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. Lastly, 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 1.12: Events for handling someone being poured a coffee as well as the `sing` plan.

Listing 1.13 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 intentions to 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 1.13: Definition of the `servedCoffee` plan.

Listing 1.14 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 for `X`. Additionally, the knowledge about `X` not having any coffee is removed in Line 16.

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

```
            -~hasCoffee(X).
```

Listing 1.14: Definition of the `pourCoffee` plan.

As shown, AgentSpeak(L) is another logic programming language for agent programming. It was specifically designed for developing BDI-agents. In the next section, an interpreter for this language will be given, which further extends it.

2.4.6 Jason$^{\circ,\dagger}$

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. [10]. 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 1.15 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`. When the agent received 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 1.15, 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 1.15: 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. [12]. 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. [12] 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 of 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. [54]. It is structurally based on KQML and FIPA [24]. A short overview of a FIPA message has been given in Section 2.4.4. We pass on presenting the structure of a KQML message here as both are similar and KQML is not further developed [38].

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

Listing 1.16: Parameters of the internal action `.send` and an example.

In Line 1 of Listing 1.16 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. Every message processing takes exactly one Jason lifecycle. To prevent information loss, every agent has a mail box. If multiple messages arrive at the same time, they are added to the box queue and processed first in first out. A developer can override Jason's default behaviour if further or 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 rules specified for the "Agents on 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.

Jason also allows running multi-agent systems over networks in a distributed manner. Hence, the workload can be distributed over multiple machines. SACI [28] and JADE are the two fully implemented distributed architectures usable out of the box with Jason [11]. Fernández et al. [24] 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. [24] 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.

This section's quick summary of Jason shows that it is a comprehensive agent programming language. Jason is not simply an AgentSpeak(L) interpreter but provides extensions to support development. The next section considers the suitability of the previously presented agent programming language for the "Agents on Mars" scenario.

2.4.7 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 early mistakes than on the actual agent development. To prevent this, we were interested in using an already developed and approved agent programming language.

Given the "Agents on 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 [3]. Yet, there was no competing team using GOLOG, FLUX, Jadex or pure AgentSpeak(L). This is of interest because the scenario of 2013 is comparable to the scenario of 2014 [1].

Schiffel and Thielscher [47] 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 [47]. The requirements for solving the problems arising from this scenario are comparable to those appearing in the “Agents on Mars” scenario. Given the former short presentation and this knowledge, it can be said that FLUX could be applied to the “Agents on Mars” scenario.

AgentSpeak(L) is suitable for multi-agent scenarios such as the “Agents on Mars” scenario as well. Especially when comparing FLUX to the components of AgentSpeak(L) plans, it becomes clear that there are many similarities. As FLUX’ actions’ 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 focusing on modelling the change of fluents. AgentSpeak(L) on the other hand was strictly developed as an application for BDI-agents. We found BDI to be a clearer structuring of agents and would in this sense prefer the use of Jason, Jadex or pure AgentSpeak(L) over GOLOG or FLUX. As Jason is merely an interpreter and extension of AgentSpeak(L), we ruled out a solution purely based on AgentSpeak(L).

Levesque et al. [33] highlight multiple problems with GOLOG. All other agent programming languages under consideration are capable of modelling incomplete knowledge. One problem is that complete knowledge is assumed in the initial situation. This is not the case for the “Agents on 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 [51]. 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 “Agents on Mars” scenario and is supported by the other languages under consideration.

A third problem is that exogenous actions cannot be handled. Exogenous actions are actions outside of the agent’s control. In the “Agents on Mars” scenario, this e.g. could be the loss of an agent’s health due to an enemy agent attacking it. The other programming languages do not face this problem.

Thielscher [51] highlights a fourth problem. It arises from GOLOG being regression-based: This means that deciding whether an action is executable or whether a property holds 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 it does not scale. FLUX on the other hand is progression-based meaning that such a decision only requires a lookup in the current state. There was no indication that the reasoning would become more complex over time in Jadex and Jason.

Due to these problems, GOLOG is unsuitable for a multiple agent-based scenario like the “Agents on Mars” scenario without considerable modifications and extensions. This lead us to discard this option as we were not willing to invest time into extending a programming language just to be able to use it.

FLUX and Jadex do not assist the developer in modelling the environment like Jason does. In general, this could be an important argument for using Jason. But for the "Agents on Mars" scenario 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 itself. More important was that 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. Due to using Java, Jadex and Jason were capable of using this library. FLUX on the other hand was not. Thus, we decided against FLUX to not having to implement the communication with the server ourselves in a logic programming language. This was especially of relevance, as the whole team was inexperienced with logical programming prior to this research lab. Hence, being able to at least develop some operations in Java as done with Jadex's plans or with Jason's internal actions was also beneficial.

Besides the support for developing environments, Jason and Jadex differ in the way how the initial beliefs, goals and plans are being programmed. Jason allows all of these to be fully written in AgentSpeak(L). If needed, plans can additionally call methods written in Java. In Jadex, beliefs and goals have to be written in XML. Moreover, the plan's signature is given in XML and its execution code is programmed in Java. Although not an exclusion criterion, we found the overhead of XML due to the verbosity of its syntax to be a downside to Jadex.

Summarised, we quickly decided against the use of GOLOG and FLUX. As illustrated, GOLOG had multiple problems which made it unsuitable for the "Agents on Mars" scenario. Although these problems were not present in FLUX, we favoured being able to program in Java and using the official contest Java library. This allowed us to focus on the development of agent strategies over having to invest time for implementing the communication with the MAPC server. In the end, we decided for using Jason over Jadex due to Jason being already proven to be successfully applicable to the Mars scenario.

3 Algorithms and Strategies

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

Write this part

3.1 Simulation Phases$^{\star,\circ}$

This section illustrates our match strategy roughly from a high-level point of view. Our general approach for the simulation was to split it up into two phases, namely an *exploration phase* and a *zoning phase*. In the exploration phase, we tried to explore the map as quickly and complete as possible. Throughout the zoning phase, agents would look for zones, form and defend them. The exploration phase is explained in this section in more detail, while the zoning phase is covered in separate section due to its complexity.

Basically all agents were used for exploration, but we used different priorities for every role. The highest priority was always to use the role defining ability if applicable. For example if the Saboteur agent could attack any enemy agent, it attacked. If a Repairer agent could repair some damaged agent from our team, it repaired. When the highest priority action was not applicable each agent decided which action to do autonomously based on the information it got from the map component or its percepts. The component itself centrally stored all information about the map and is presented in Section 3.3.3. We distinguished three group types of exploring agents.

The first group consisted only of Explorer agents. Their highest priority was to get vertex values by probing. Secondarily they used the `survey` action to explore the map only if they came across vertices which were not surveyed. They were not actively searching for or going to not surveyed vertices. Instead, they moved somewhat circular towards vertices which were not surveyed yet as further described in Section 3.2.

The second group was comprised of Saboteur agents. We followed a very aggressive strategy and aimed to attack and disturb the enemy as much as possible. Consequently, we did not want to distract our Saboteur agents by exploring the map. Saboteur agents would only explore the map if they did not know of an active enemy somewhere on the already explored map. As part of our aggressive strategy, there was one Saboteur agent which would even then not start exploring. Instead, it would try to increase its visibility radius to find more distant enemies. This is explained in more detail in Section 3.2. The Saboteur agents stayed with this strategy throughout the whole simulation and not only the exploration phase.

The third and last group consisted of the remaining agents, which were the Repairer agents, the Sentinel agents and the Inspector agents. These agents were used mainly for exploring the map. They were coordinated by querying the map component for the next unexplored map area. We distinguished between explored areas, which are map areas where we know the weights of all vertex edges, and unexplored areas where we do not know the weights of all vertex edges. If agents of the third group came to a vertex that had unknown edge weights to at least one neighbour vertex, they used the survey action to get the information as percepts. These weights were then passed to the map component and they repeated the exploring, by querying for the next not surveyed vertex and going there. To prevent multiple agents from going to the same vertex and exploring the same area, the map component used an internal locking mechanism. For zoning we were not interested in a full coverage of the map. This was because a full map exploration would have consumed a lot more steps while bringing only little improvement to the knowledge about it as a whole. Of course it could be that at some point during the simulation we gain a full coverage of the map, but it was not a criteria to switch to the second phase in the simulation.

At some point in the simulation, around step 100, there were more agents available for exploring than vertices that needed to be surveyed. Responsible for this could be an almost fully explored map or a bottleneck vertex which is the only connection to other parts of the map. Agents without an assigned vertex would be idle and waiting for the next step. As we assumed that our agents would always be evenly placed on the map at the beginning of the simulation, we did not pursue a solution for the bottleneck vertex. Instead, we decided to remove every agent from the exploring agent team which could not be assigned an unsurveyed vertex and put it in the zoning team. In the zoning phase, all agents which were finished with their exploration phase were used to build zones. This included the Explorer agents but excluded the Saboteur agents, because they were following their own aggressive strategy. A detailed description of the zoning phase is given in Section 3.5.

3.2 Agent Specific Strategies[†,○]

Each of the five different agent types (or *roles*) in the MAPC scenario — Explorers, Repairers, Saboteurs, Sentinels and Inspectors — has different capabilities in terms of the actions they can perform. Thus, 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 execute the `probe` action. Vertices must be probed in order to learn their value, which is critical for building zones with high scores. Accordingly, an Explorer agent will spend most of its time seeking out, moving towards and finally probing vertices whose values are not yet known. Having multiple explorer agents move towards the same not yet probed vertex is in most cases suboptimal. Since there are 6 Explorer agents in a team, special care is taken to prevent this as described in Section 3.3.

@msewell: "For all assumptions or design choices you make, add a justification and hint to possible problems" - Matthias

update this reference to wherever we explain why multiple agents won't run onto the same node

In our implementation, we are generally interested in the value of all vertices and hence would want to have all vertices probed. Yet, we prioritise vertices in a specific way which we call *cluster probing*. Therefore, an Explorer agent's probing will rather be circular than e.g. a straight line. The motivation behind this is that our algorithm for zone calculation as presented in Section 3.5.1 puts agents around a centre vertex in a circular manner and with a maximum distance of two edges away. By having cluster probing mimic the circular shape of the zones calculated by our algorithm, we enable our agents to quicker find and establish high-value zones. To achieve this, our probing algorithm prioritises not yet probed vertices first by distance, then by the number of edges they share with already probed vertices. The result is that the Explorer agents' movement is similar to a spiral pattern, provided the Explorer agents are not disturbed by e.g. nearby enemy agents. Explorer agents do not stop this probing pattern until they can no longer find any not yet probed vertices.

Repairer agents are the only agents who can perform the `repair` action for restoring health of disabled agents. Disabled agents can only move and recharge and cannot be used in building zones. Therefore, a team loses out on possible points for every disabled agent in the team. So to achieve a high score it is essential to quickly repair damaged agents. In our implementation, Repairer agents' actions are prioritised so that they will attempt to repair any disabled friendly agent in their visibility range. It is the "job" of the disabled agents to find and move towards the closest friendly Repairer agent. If a Repairer agent is aware of a friendly disabled agent outside of its visibility range, and the Repairer agent is currently not used for zoning, however, then the Repairer agent will also move towards the disabled agent. More detail on the process of repairing is given in Section 3.4.

@sdedukh, **@msewell**: Section 3.4 says, this is only the case throughout exploration phase. Code-wise, I did not find any clue that repairers would move towards disabled agents at any time. What's correct here? Update both sections accordingly.

Saboteur agents are the only agents that can disable enemy agents using the `attack` action. In our implementation, a Saboteur agent's role is very aggressively defined. The prioritisation is therefore to attack all non-disabled enemies within the Saboteur agent's visibility range. If the Saboteur agent does not see such an enemy, it will try to find one and then move towards it to attack it.

Throughout our development, Saboteur agents were the only agent types which would use the `buy` action. Our initial strategy would allow each Saboteur agent to buy one upgrade to extend its visibility range. Doing so increases the probability for successful ranged attacks [1] and would support the Saboteur agents' offensive strategy. We decided to try out other buying strategies by having our agents play matches against copies of themselves with the copies using different strategies for buying upgrades. Although it was a brief, empirical and rather informal testing approach, we discovered a strategy which would lead to persistently higher scores. Instead of each Saboteur agent buying a visibility range upgrade once, we would allow a special Saboteur agent to buy multiple upgrades. We called this agent the *artillery agent* due to its ability to successfully attack agents multiple hops away. The artillery agent decided to buy upgrades whenever it would not have a non-disabled enemy within its visibility range and buying was possible given the amount of available money. Thus, we were able to outperform copies of our agents which were using our more conventional approach of upgrade buying.

Our artillery agent can buy upgrades for its maximum energy, visibility range, maximum health or strength. The kind of upgrade which it buys, depends on the relative improvement this upgrade will bring to the upgrade-specific "module". For example, if the artillery agent has a maximum health of 3 and a visibility range of 1, then increasing the maximum health to 4 would be an improvement of 33 %. However, increasing the visibility range from 1 to 2 would be an improvement of 100 %. Thus in this example, the artillery agent will choose to buy an upgrade for its visibility range.

We also call in Saboteur agents to defend a zone if it gets attacked by an enemy agent as mentioned in Section 3.5.3.

Sentinel agents do not 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. Yet, we do not 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 at least once during every match to learn and store that agent's role. This is necessary to know which enemy agents are Saboteur agents, so that our agents can avoid them.

Furthermore, inspecting enemy agents is rewarded with achievement points. They are not rewarded for performing `inspect` actions on an enemy agent more than once. The only use case for inspecting an enemy agent more than once during a single match would be to learn if they have bought any upgrades since the last time they were inspected. But this year's and last years' matches showed that buying an upgrade for an agent is a rare occurrence during the actual contest. Therefore, re-inspect is not of much interest and inspecting each enemy agent only once could be sufficient. In our implementation, however, we toggle an enemy agent to be ready to be inspected again 50 turns after it was last inspected.

3.3 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 or 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 with a high zone score. To achieve this, it is mandatory to store information about the map, such as its vertices, edges, 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 3.3.1, presents our first approach and its up- and downsides. After that, a basic overview of our initial map building approach is given in Section 3.3.2. Finally, this section concludes with a description of the second approach we tried and ultimately decided to use in Section 3.3.3.

3.3.1 Cartographer Agent$^{\star,\circ}$

Very early in our development process, we decided that we wanted to store all information about the map in a central place. This means we were not interested in each agent storing all map data in its own belief base. The intention behind this decision was to reduce the effort in synchronising and maintaining data between individual agents. This section presents our initial approach, which was to install one additional, omniscient agent which we called the *cartographer agent*. It was a separate agent existing in the background and was independent from the scenario's 28 agents. The cartographer agent stored map information and had the task to calculate shortest paths between given vertices. Said map information included vertex values, edges and edge costs. Every agent told the cartographer agent about its environment-related beliefs and the cartographer agent stored these beliefs. Environment-related beliefs are given 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 1.17: 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. The agent could also query the cartographer agent to know whether a vertex was already probed or surveyed. See Figure 7 for the communication process of our first approach. Figure 8 shows the distribution of the single components related to map exploration.

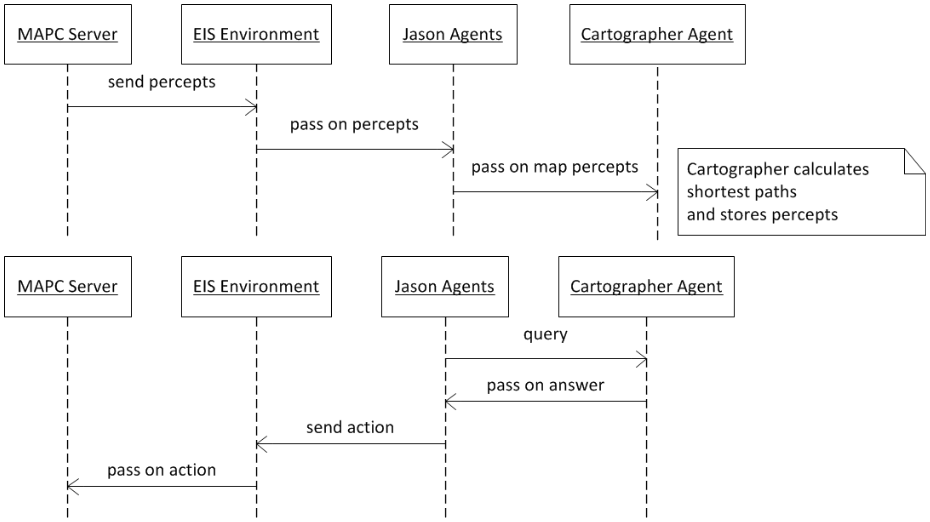


Fig. 7: Initial communication approach for map generation.

Shortly after implementing this approach, we encountered two major problems, which both resulted in serious performance issues. One problem came from our implementation of the pathfinding algorithm to determine the shortest path between two vertices. Although we used Dijkstra's algorithm, which is an efficient algorithm for this kind of task, we encountered performance issues. This was because the pathfinding algorithm had to be executed every time an agent asked for a shortest path, which led to a lot of redundant calculations and processing by the cartographer agent.

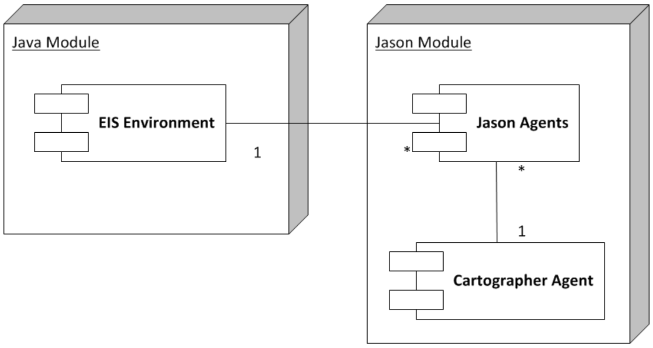


Fig. 8: Distribution of components between Jason and Java in our first approach.

The second problem was related to the communication between agents. To understand this problem, one needs to know that Jason uses a mailbox system for inter-agent communication. This means that every message by an agent is queued in the receiver's message inbox. Only one message gets processed per Jason reasoning cycle. Although a single Jason reasoning cycle is a lot shorter in terms of time than the server-side simulation step, we found that the cartographer agent's mailbox received more messages than it was able to process, resulting in a significantly delayed response to these messages. Both of these issues resulted in blocked agents, which had to wait for responses to their queries for several steps.

The following example serves to illustrate this problem. An exploring agent reaches an unvisited vertex. At that point, the agent asks the cartographer agent, whether this vertex has been surveyed at some point. Once the agents receives an answer, it either surveys the vertex if has not been surveyed before or asks the cartographer agent for the closest unsurveyed vertex, which will become the agent's next destination. Once the agent reaches this vertex, this process starts over. Ascan be seen, at least two messages are needed each time a new vertex is reached and hence there are two possible bottlenecks. The first one is the query for the state of a vertex and the second is the query for the next unsurveyed vertex, which includes calculating the shortest path to this vertex. If the agent surveyed the vertex, it will additionally have to inform the cartographer agent about any new information. This information is received and will be forwarded by the respective agent at the beginning of the next step. Due to the way inter-agent communication is handled in Jason, this results in the cartographer agent not being able to handle queries immediately. In our tests, we saw query response times of up to ten to twenty server steps. This lead to our agents idling most of the time, waiting for answers from the cartographer agent. With the agents being idle most of the time, continuing with this approach would obviously have led to poor scoring during the competition.

The next section describes our second approach, which reduces the calculation overhead for the cartographer agent when calculating shortest paths.

3.3.2 Distance-Vector Routing-Protocol$^{\star,\circ}$

This section presents our next approach, which tried to solve the problem of repeated shortest path calculations. This approach was first intended to work in tandem with the cartographer agent, but later came to be used independently. We decided to calculate all shortest paths as early as possible and store these paths together with other information in a network of what we called *node agents*. A node agent is an agent representing one vertex of the scenario map and stores information about this vertex. All percepts perceived by our 28 regular agents were passed on to the node agents. To make it easier to address node agents, we named node agents after the vertex they represented. Node agents stored all their neighbours in a neighbour-list belief `neighbours([<ListOfNeighbours>])`. Additionally, they stored all available paths to other vertices as shortest path beliefs `minStepsPath(<Destination>, <NextHop>, <Hops>, <CostToNextHop>)`. Similarly, they stored all available paths to other vertices as cheapest path beliefs `minCostPath(<Destination>, <NextHop>, <TotalCosts>, <CostToNextHop>)`. The addition of a cheapest path belief would allow an agent to travel towards a destination vertex while using the least amount of energy. The idea behind this was to allow an agent to travel to a destination vertex while reducing the amount of times the `recharge` action had to be performed by the agent, at the cost of possibly needing more in-between hops (and thus server steps) to reach the destination vertex. In regard to exploration and zoning, we decided that a node agent also had to store the probed value of the vertex, and whether it had already been probed or surveyed.

At first, we changed the cartographer agent's tasks, so that it only had to create the node agents at runtime and redirect queries to them. Hence, the cartographer agent became an intermediate agent responsible for creating and communicating with the node agents. All map-related percepts were now redirected by the cartographer agent to the respective node agents. Agents would query node agents directly when they were seeking information about how to reach a specific vertex. This only happened when agents could not see within their visibility range an unsurveyed vertex to explore next. Yet, they had to ask the cartographer agent for a list of unsurveyed vertices further away. That way, we were able to ensure the existence of a node agent prior to a regular agent communicating with this node agent. Figure 9 shows the second approach we used and Figure 10 the corresponding distribution model.

The dynamic creation of node agents during runtime was done because the number of vertices is not known until the simulation start. As the creation of a few hundred agents takes some time itself, it was not possible to create the node agents during runtime after the simulation started. But having the cartographer agent function as an intermediary made it another bottleneck. So, we later eliminated this agent completely and simply created 625 node agents before the simulation started. This is the maximum number of possible vertices used in the MAPC. For the sake of performance, we deemed it acceptable to have some idling node agents which would not engage with any other agent. We also experimented with removing unnecessary node agents once we received the

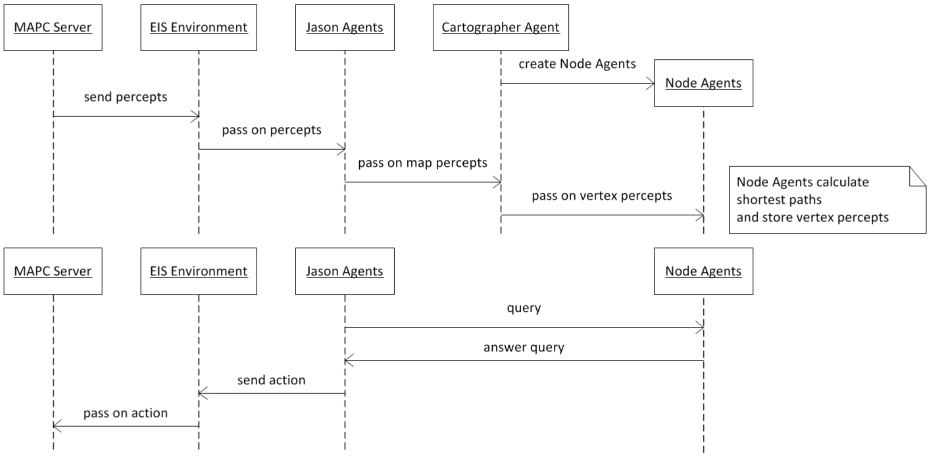


Fig. 9: Second communication approach for map generation with using the cartographer agent.

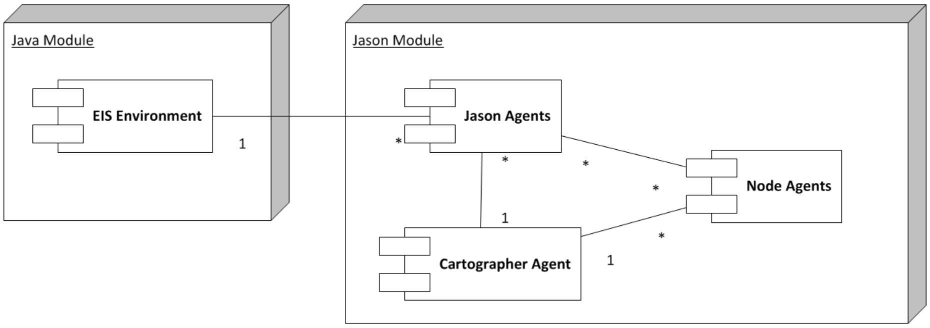


Fig. 10: Distribution of map components in our second approach with using the cartographer agent.

information about the exact number of vertices for a given map. However, this had no noticeable impact on the performance or stability of the system, which is why it was ultimately removed.

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

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

A query for a shortest path to `v8` would then look like this:

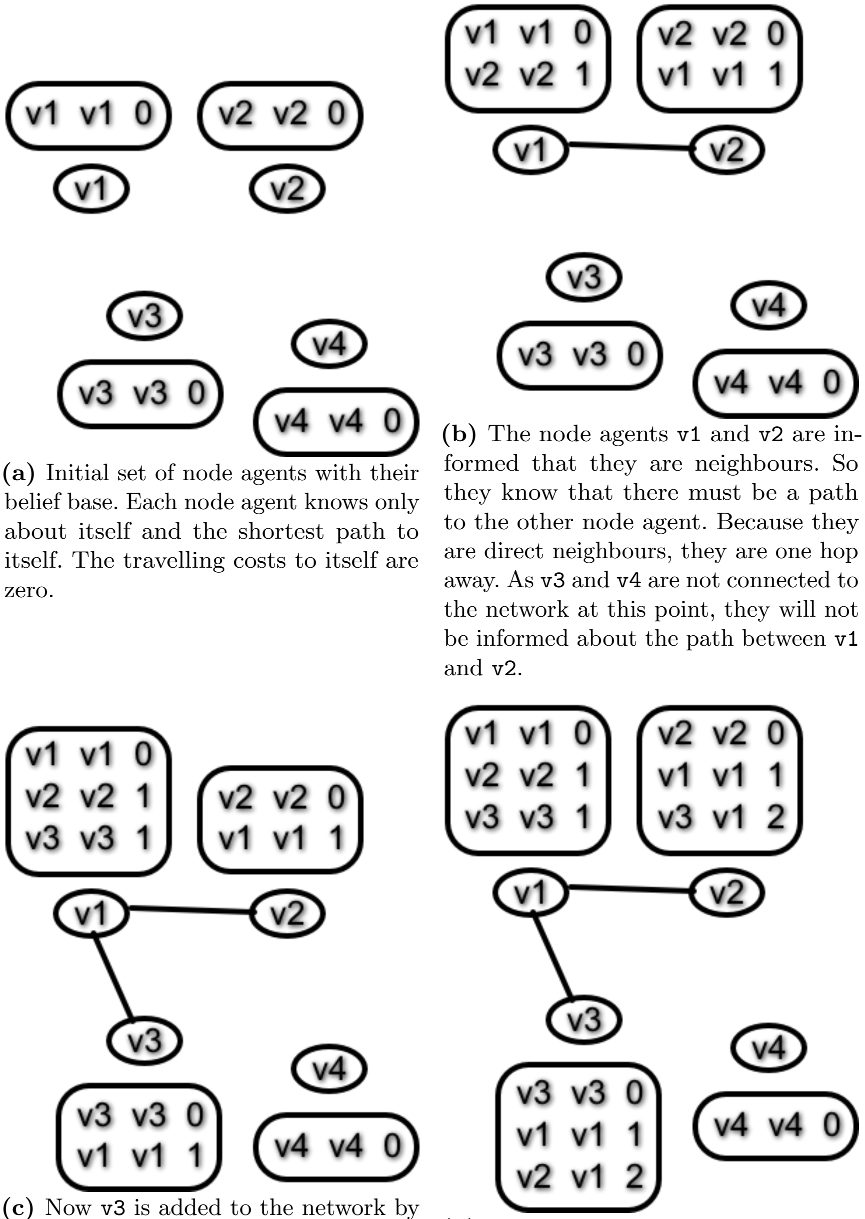
```
  .send(v1, askOne, minStepsPath(v8, NextHop, _, CostToNextHop)).
```

Listing 1.18: 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 send this as a response to the querying agent.

For propagating data between these node agents, we used a modified *Distance-Vector Routing Protocol* [18] (short: DV). DV is a routing protocol based on the Bellman-Ford algorithm and normally has its use in packet-switched networks. It 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 their neighbours and so on. Because this algorithm has no loop detection for larger networks, we had to implement some kind of break condition for the algorithm. We used the value of the calculated shortest path or the calculated cheapest path respectively as a termination condition. If a newly calculated path to another node agent was shorter than the shortest known path, then the node agent informed its neighbours about this new shortest path. Otherwise it would do nothing. At some point, all information and paths will have been propagated through the entire network and all node agents will have have a consistent belief base. Figure 11 illustrates the algorithm for a small set of four neighbour nodes.

Fig. 11: Executing the Distance-Vector Routing Protocol algorithm as described in Section 3.3.2 on a small network of four nodes to calculate the shortest paths. 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. For more compact display, we left out the parameter showing the cost of the edge to the next hop.

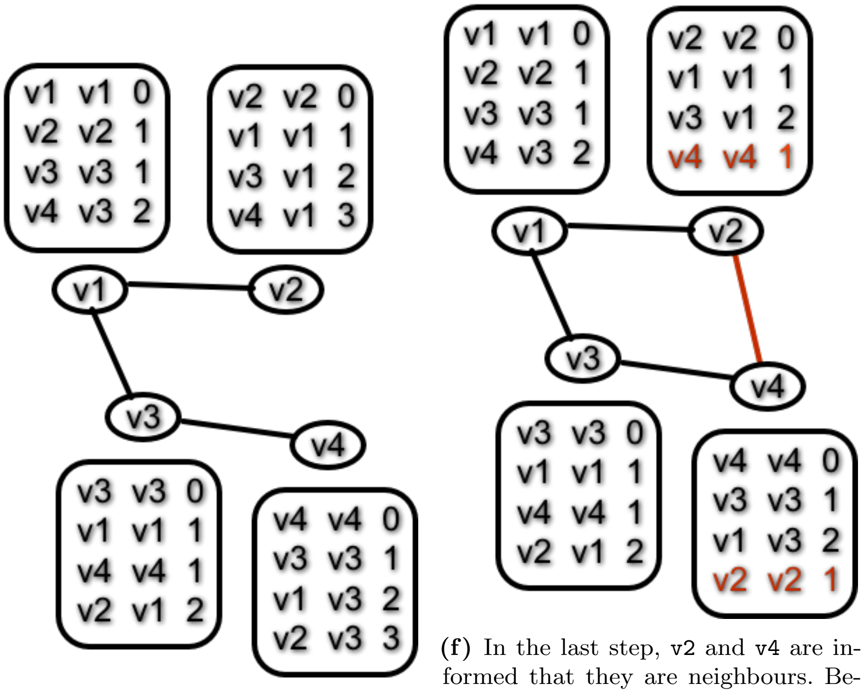


(a) Initial set of node agents with their belief base. Each node agent knows only about itself and the shortest path to itself. The travelling costs to itself are zero.

(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. As `v3` and `v4` are not connected to the network at this point, 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`. The shortest path to `v3` from `v2` is now known to be 2 hops away, going over `v1`.



(e) Adding the knowledge about an edge between `v3` and `v4` will make all node agents update their 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 unchanged, because they cannot improve.

By changing the distribution of map-related information from the single cartographer agent to a large network of node agents, we also distributed the load from one agent between the respective node agents. But we still had queries which were not answered immediately, because now we had a lot of communication between node agents. Around 400 Jason agents were calculating shortest and cheapest paths in parallel within the node agent networks. At the same time, they received new information by the exploring agents. Although this information was redundant in most cases, it still had to be processed. This led to a high load on our system. To reduce the load, we decided to throw out the cheapest path calculation and only resort to calculating the shortest paths between vertices. We made this decision because we believed that losing a step due to taking a longer path with more hops was worse than having to recharge more often on the shorter path. The reason for this is that recharging returns half of the maximum energy to the agent, which is enough to be able to pass multiple hops most of the time.

Next, we reduced communication between node agents and real agents. At all times, the necessary map information was first received from the Java EIS

Environment module. The Java EIS Environment module is our interface to communicate with the MAPC server. Through this interface, we got agent percepts from the server and transmitted actions to the server. Originally, the map information was blindly transmitted to the agents who perceived this information. They then themselves passed the information on to the node agents. Our change here was to add a filter to this module which would only transmit new information and would send it directly to the respective note agents. Since the message boxes were no longer flooded, communication between agents improved significantly.

However, two reasons made us discard this approach as well and change to a solution based entirely on Java. First, we were not able to reduce the load on our system sufficiently, even with these changes. Further, we observed that some beliefs were not received by the node agents. Even worse, over time beliefs disappeared from node agents' belief bases. Due to the constant high work load on our system, we saw that agents did not send actions to the server in time, which led again to a lot of idle steps for our agents. The next section presents the solution which we used for the competition.

3.3.3 JavaMap$^{\star,\circ}$

We decided to implement the entire map module in Java. This solved all of the previously described problems. The JavaMap module gets its map information directly from the Java EIS Environment module. It is queried through internal actions by the Jason agents which were presented in Section 2.4.6. Internally the JavaMap creates and manages a list of vertex-objects which are similar to the node agents from our previous approach. Every vertex stores a list of all paths it knows and all vertex information. This includes the one-hop neighbourhood and the probed value of the vertex. During our tests we often encountered the problem that multiple agents were sent to the same node. To solve this, we extended the functionality of our JavaMap with a mechanism to prevent mutliple agents being sent towards the same node when exploring, probing, repairing or attacking. Internally, we used a reservation list for each task. In this list we kept the information about which agent was doing which task, e.g. we used the survey list to store the information that an agent reserved a node for surveying. If another agent asked for a not already surveyed vertex, the nearest not surveyed vertex was first checked against the reservation list. When the vertex was not in the list, it was assigned to the agent and the list was updated respectively. Otherwise, the next nearest not surveyed vertex was checked, and so on. The list was reset at the beginning of each step. This allowed agents to react to environment changes in each step, such as vertices changing their surveyed state or enemies in their path to the vertex.

Figure 12 shows how the map percepts are now passed to the JavaMap and how Jason agents query the JavaMap.

In Figure 13 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

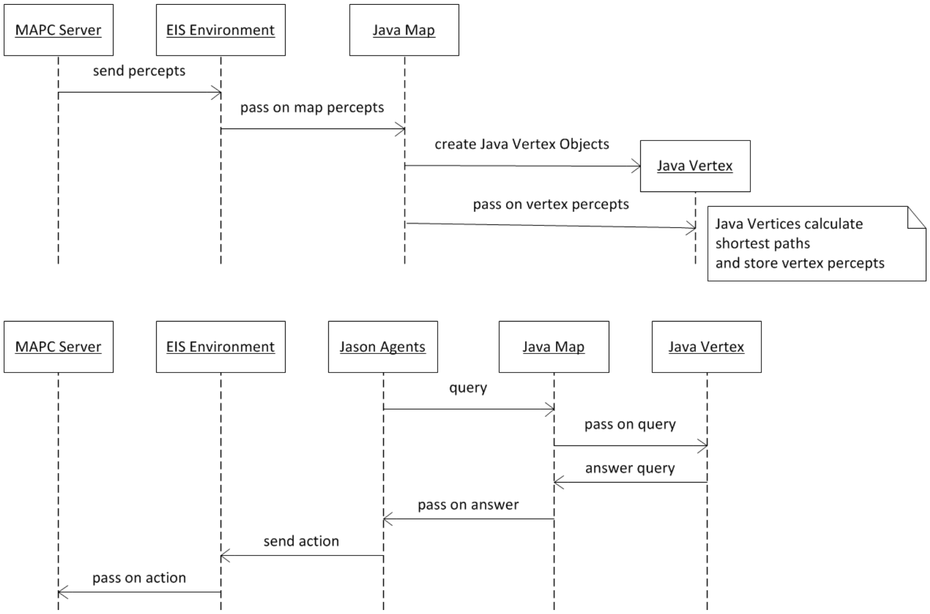


Fig. 12: Final communication approach for map generation.

and calculation is done entirely in Java and agent related actions, planning and communication entirely in Jason.

With JavaMap we found an exploration approach which is fast, stable and has a high coverage of the actual map. Furthermore, we could ensure that all information is accessible by every Jason agent in reasonable time. We were also able to reduce the time to receive answers to rather complex queries like paths between distant vertices.

3.4 Repairing$^{\diamond,\dagger}$

As was already mentioned in the scenario description, any agent can become disabled after being attacked by an enemy Saboteur agent. To become disabled in the scenario means to lose all of the agent's health points. Naturally, we have implemented several supporting strategies for avoiding enemy Saboteur agents when possible and parry when there is a Saboteur agent nearby. However, following these strategies does not guarantee that the agent will never become disabled, mostly because an "escape route" that would put an agent out of the attack range of an enemy Saboteur agent does not always exist and not all agents are able to perform the `parry` action. When an agent becomes disabled, it loses most of its functionality: only the `skip`, `recharge` and `goto` actions can be performed. Repairer agents can also perform the `repair` action when disabled, although repairing costs more energy to perform when disabled. Disabled agents

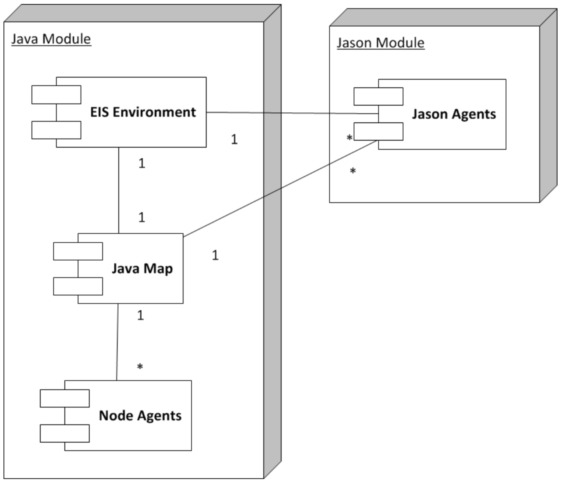


Fig. 13: Distribution of map components in our final solution.

also do not count towards establishing team ownership of a map vertex, and thus do not contribute to zone scoring.

In Section 3.2 it was said that the primary task of Repairer agents is to repair others, and that they should prioritize the `repair` action whenever they see a disabled friendly agent within their visibility range. It is important that disabled agents are repaired quickly, and for this to happen the disabled agent needs to be brought within repairing distance of a Repairer agent. In our implementation, every time an agent becomes disabled, it receives the high priority goal *getRepaired*. Following the plan of this goal, an agent requests an available Repairer agent and its position from the *MapAgent*. If the returned Repairer agent position is the same the disabled agent's position, then the disabled agent only recharges and waits to get repaired. Otherwise, the disabled agent simply moves towards the returned Repairer agent position. If there is no Repairer agent available within the part of the map that the disabled agent knows how to reach, i.e. the explored subgraph the disabled agent is currently on, the disabled agent will instead attempt to expand the knowledge of the map by moving towards the closest unexplored vertex.

Assignment of agents to their Repairer agents is done inside our Java MapAgent. To be more flexible, we decided to perform these assignments on every step. This allows the system to adapt to constantly changing situations, such as when agents are moving, some other agent becomes disabled or a previously disabled agents is repaired, freeing up a previously agent Repairer agent. The assignment itself is done based on the hop distances between agents. First, all the distances between all disabled agents and all Repairer agents are calculated. Then, the paths with the shortest distance are selected and the agents belonging to these paths are assigned to each other. If all Repairer agents are assigned and there are still some unassigned disabled agents, they get assigned to the closest Repairer agent, even though that Repairer agent is already assigned to another disabled agent. his assigning approach in most of the cases led to fast and effective repairing.

In addition to disabled agents moving towards their assigned Repairer agents, Repairer agents can also move towards their assigned disabled agents. This behaviour is only possible during the exploration phase of the simulation, because in zoning mode Repairer agents moving will often to lead to the zone they help maintain being broken up, which we would like to avoid. We implemented this by making Repairer agents explore the map in the direction of their assigned disabled agents, i.e. if the vertex the Repairer currently occupies on the way towards its assigned disabled agent is not surveyed, they survey it, otherwise they will continue on their path towards their assigned disabled agent.

What happens if a Repairer agent becomes disabled? We decided to not treat this as a special case, but instead use the standard procedure of disabled agent to Repairer agent assignment. For this to work, for Repairer agents the goal of repairing a disabled agent is higher on their priority list than waiting to get repaired themselves. This helps to use all the repairers more effective and prevents the situation when all the Repairer agents are disabled and waiting to get repaired.

3.5 Zone Forming$^{\circ}$

Zone forming is the most important part in the “Agents on Mars” scenario [1].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 3.5.1. Said algorithm is used in the process of finding and negotiating a zone to build, which is described in Section 3.5.2. After a zone that can be built has been found, agents are 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 3.5.3.

3.5.1 Zone Calculation$^{\dagger,\circ}$

The graph colouring algorithm used by the MAPC server to determine occupied zones is described in detail in the MAPC 2014 scenario description [1] 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 14 shows an example. To position themselves in an optimally-scoring way, agents *could*

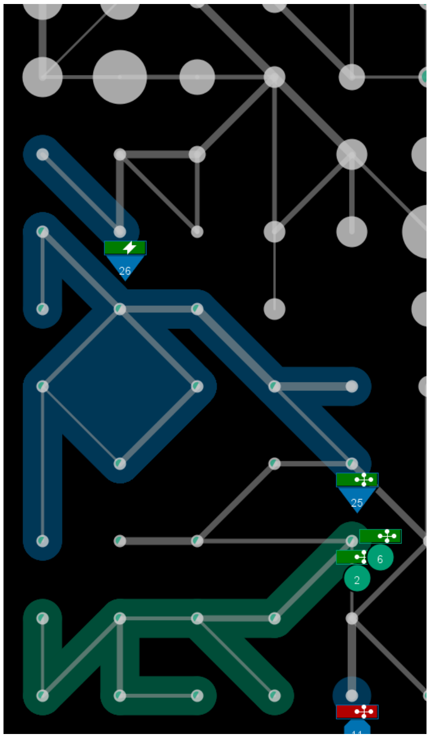


Fig. 14: By occupying an articulation point, a single agent can establish a high-scoring zone—provided that there are no enemy agents inside the subgraph that is split off from the main graph.

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 $C\left(n+r-1,r-1\right)=\frac{(n+r-1)!}{n!(r-1)!}$, a number that increases rapidly with n and k. In particular, there are $C\left(28+600-1,600-1\right)=3.75\times10^{48}$ 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 this vertex such that:

– All vertices that share an edge with the centre vertex will be included in the zone.

- Besides the centre vertex itself, agents should only be placed on the centre vertex's two-hop neighbours. These are those vertices whose shortest path to the centre node has a length of two. Later, we will illustrate that there are rare cases in which agents must be placed on one-hop neighbours as well.
- The constructed zone's value per agent, that is, the sum of the values of each vertex in the zone divided by the number of agents required to establish that zone, should be high. Ideally, it would be maximal, but the heuristic we use does not guarantee this.

Figure 15 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 easiest illustrated graphically, as in Figure 16.

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 centred 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: $V_v^2 = \left\{u \middle| (v,w) \in E, (w,u) \in E, u \notin V_v^1 \cup \{v\}\right\}$. 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 [1]. 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.*

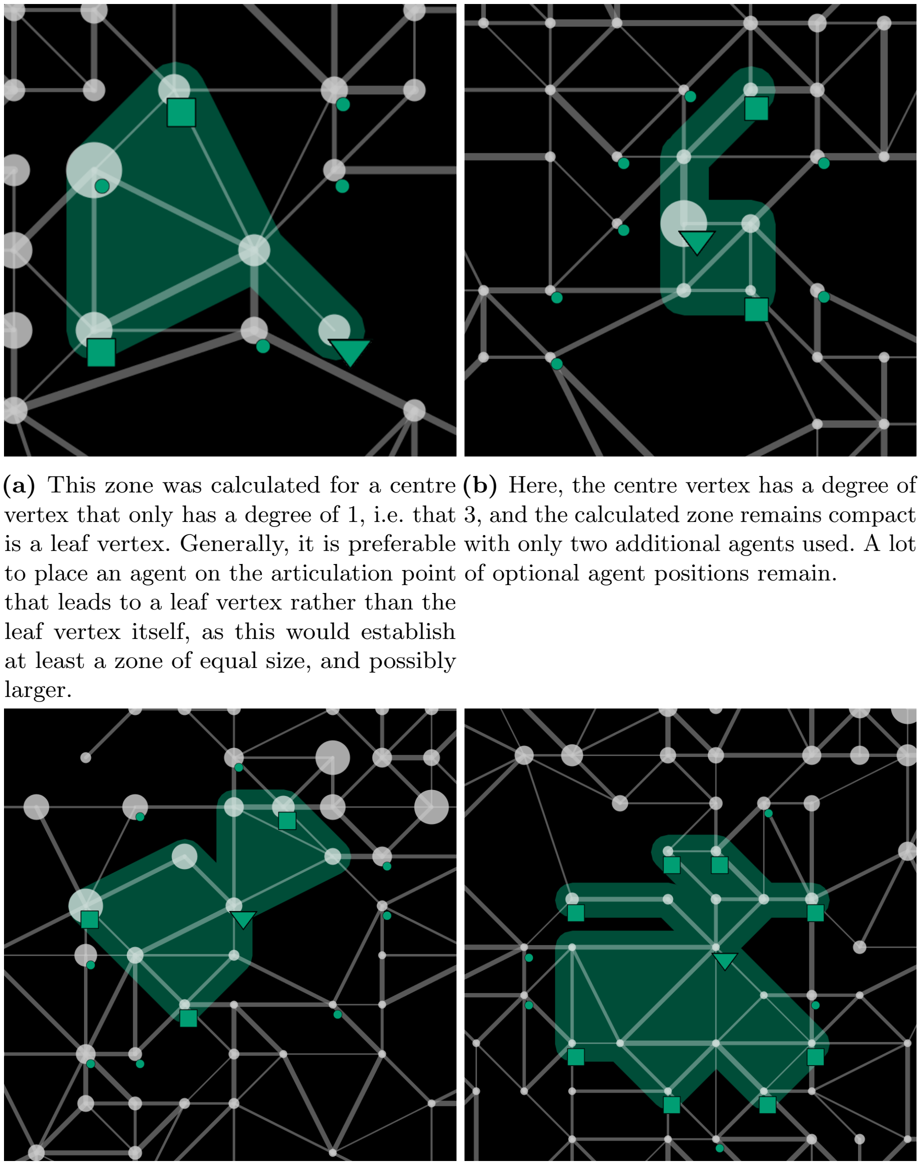
If we happen to explain this roughly in the MAPC section, we can further refer to that one

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's two-hop neighbourhood V_v^{2+} is discovered during map exploration or changes its known value. Value changes happen when a vertex is probed by an Explorer agent. These two events are the only ones that can lead to possible changes in A_v and thus the zone.

The zone centred 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 directly to 2 or more vertices in V_v^1, add it to A_v and remove it from V_v^2:

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



(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 articulation point 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. 15: Four examples of zones calculated by the heuristic algorithm described in Section 3.5.1. The green squares and triangles represent the placement of agents, where the triangle is the agent on the centre 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 3.5.3. The green-coloured 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:

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

   The reasoning behind this is that those vertices in V_v^1 that are neighbours of those in A_v will definitely be included in the zone. Moreover, 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 bridge and the in-between vertex unoccupied:

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

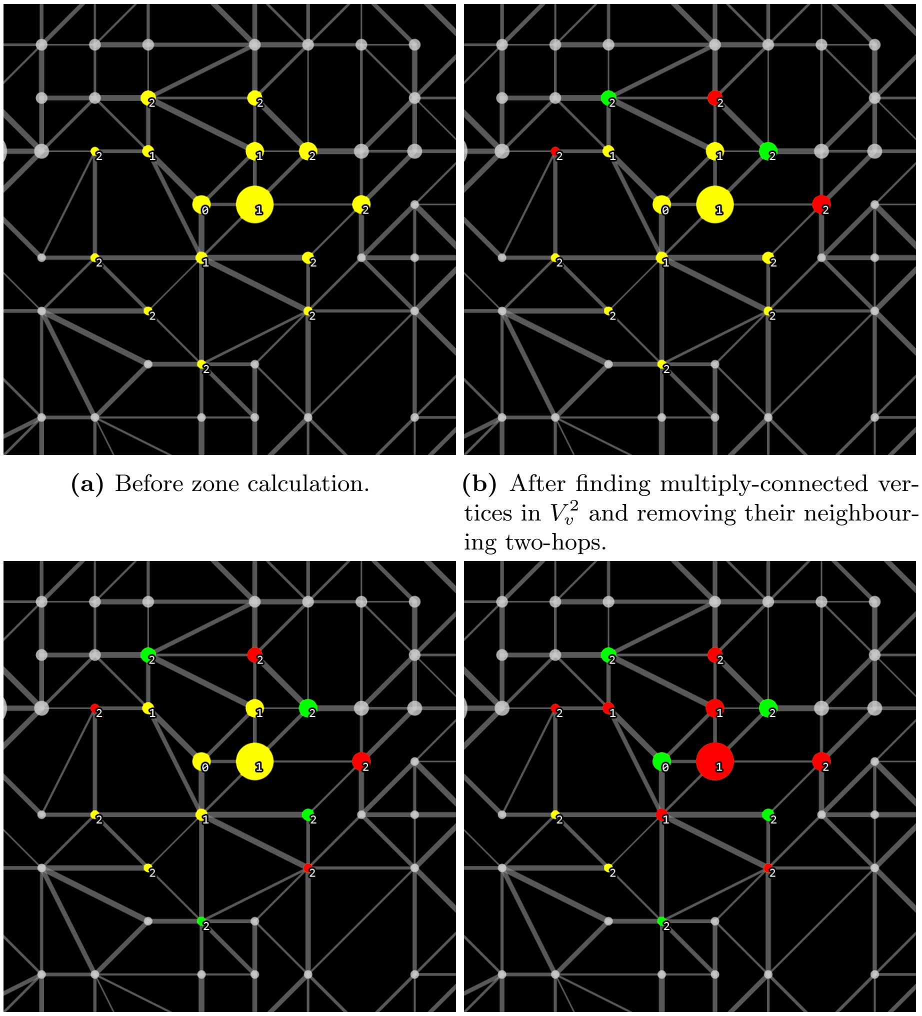
   Since three vertices can be captured in the zone for the "cost" of two agents, we consider this a good exchange to make.

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 directly 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:

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

5. Finally, we include the centre 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 They could be used to extend the zone by otherwise idle agents. But unlike the vertices in A_v the vertices left in V_v^2 are not required to establish the initial smallest zone that we calculated.

While we consider our algorithm to find zones of acceptable 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 Furthermore,



(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. 16: 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.

it is not difficult to 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 16d were an articulation point whose inclusion in the list of agent positions would add more than that single vertex to the zone. Then, this would probably be a good vertex to place an agent on. Yet, it would not be discovered by our heuristic algorithm.

3.5.2 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 successfully, 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 3.3, 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 3.5.3.

In the beginning, all agents have to centrally register themselves when they are ready for zone building to indicate their availability. Next, each agent uses the algorithm presented in Section 3.5.1 to determine the best zone in its neighbourhood. The algorithm will only return zones which need at most as many agents to be built as there are registered agents. This is to ensure that agents will not try to build zones for which there are not sufficiently many agents available. The algorithm further uses a range parameter k. It 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 built 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 3.5.3.

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.

3.5.3 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 building 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 3.5.2 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 agent. 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 3.5.2.

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 surroundings 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 3.5.1. 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 [1]. Moreover, enemy Saboteur agents can disable agents inside a zone, which similarly destroys the zone in its original form [1]. 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 forever 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 built which take the information of the newly probed vertices into account, this also allows for agents to start the zone finding process in a less asynchronous fashion.

Although the zone calculation algorithm was able to calculate additional zone spots to place agents, we did not make use of it. The main reason was because we changed the strategy of zoning agents. Earlier approaches allowed for agents forming a zone to leave their vertex. They would be able to do so when an enemy agent was nearby or when a repair agent forming a zone would have to move towards a disabled agent. In these cases, being able to extend zones by more agents would have been beneficial to keeping up the zone. If an agent on an optional zone vertex would have to leave, the zone would still be kept up. But if an agent on a mandatory zone vertex would have to leave, an agent on an optional zone vertex could quickly take its place as it is already nearby. Changing the strategies so that no agent forming a zone would ever leave its vertex unless it got disabled made these additional zone spots less relevant. Lastly, we decided to abandoned the concept of extending zones by placing agents on optional zone spots. This was because it seemed more promising to make idle agents which were not able to build a zone just now move onto a highly valuable vertex in their neighbourhood. It was also simpler to realise than dealing with all edge cases extending a zone and holding it would bring. For example, we otherwise would have had to account for coaches leaving a zone and an agent on an optional zone spot filling in for it. Said agent would then have to inform all minions about the change. If this information did not get through to all minions fast enough, there would be the chance of them trying to interact with the former coach. As there was a shortage of leftover development time until the competition, we stayed with this decision.

4 The BDI Model in Jason and our Implementation$^{\blacktriangle,\circ}$

This section presents the interpretation cycle of an AgentSpeak(L) program as implemented in Jason. It shows a more extensive picture on the reasoning process of a Jason BDI agent also with regard to our implementation.

AgentSpeak(L) is an agent-oriented programming language built around the BDI model [9]. This language is based on a restricted first-order language with events and actions [42]. The behaviour of an agent such as the interaction of that agent with the environment is implemented in AgentSpeak(L). In other words, AgentSpeak(L) encodes beliefs, desires, and intentions in a way processable by an agent.

As explained by Rao [42] beliefs model the current state of an agent, expressing its knowledge. They may be modified on environment changes due to some internal or external events. Rao furthermore describes states the agent wants to reach are desires as in the earlier presented BDI approach. Analogously, he continues to define intentions as an agent's attempt to reach such a state by the commitment to concrete plans.

As Jason is an interpreter for AgentSpeak(L) with Java-base extensions [8], it it suitable for practical multi-agent systems. Some details on the functioning of an AgentSpeak(L) interpreter are presented in Figure 17.

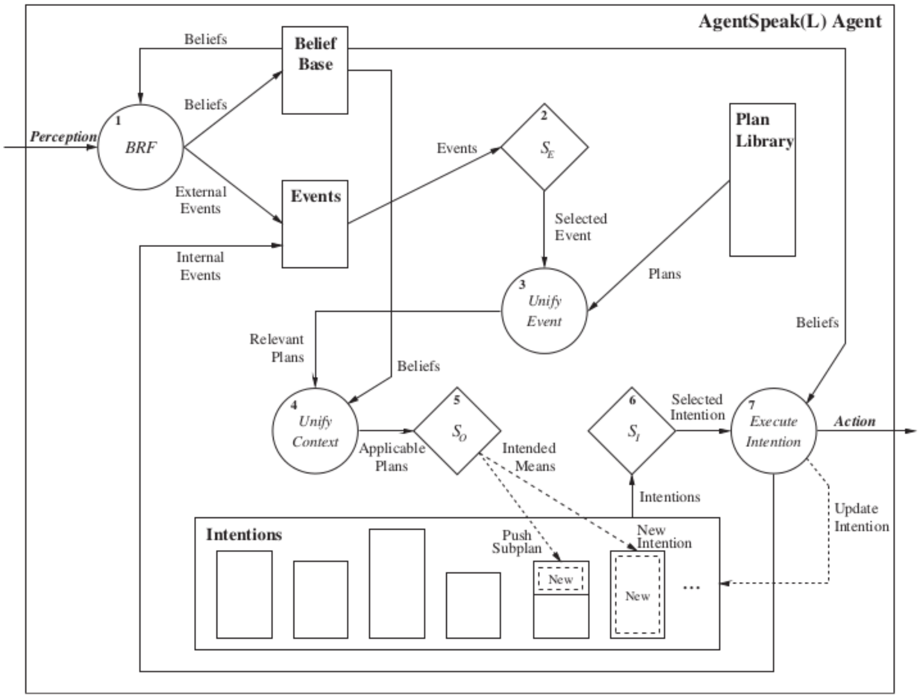


Fig. 17: An interpretation cycle of an AgentSpeak(L) program [9].

An AgentSpeak(L) agent consists of its initial knowledge which is encoded in its belief base and a plan library [9]. Figure 17 does not visualise the source of the beliefs in the belief base properly. The beliefs do not only come from external perceptions of the environment. Instead, they can also be the result of plan executions by the agents which only have internal effects. The beliefs from perceptions are annotated by `source(percept)`. In our implementation, there are 34 different types of perceptions received from the server, which an agent would store as beliefs. Initial beliefs and generally all internal beliefs are annotated with `source(self)`. The external beliefs describe the current states of each agent such as its energy or its role. These beliefs are not immutable and can be used and modified in plans.

In the interpretation cycle shows that events also play an important role. After the selection of the events by the *event selection function* $\mathcal{S}_\mathcal{E}$ the events will be unified with the triggering events in the head of the plans from the plan library. In our system, there are 184 plans in total. We have 88 plans with discrete triggering events meaning that our own plans handle 88 different events.

Desires in AgentSpeak(L) are expressed through test or achievement goals. We explicitly implemented achievement goals in our program. For example, `!doParry` is an achievement goal that will lead to the execution of the `parry` action when a certain plan matches the event. However, querying the belief base in the context of a plan is converted by Jason into an achievement goal automatically.

For some tasks, we have multiple plans as there are many different contexts to consider. As an example, attacking is covered in more plans than other actions because of our complex strategy there. A variety of situations must be considered before attacking like determining if there is an enemy nearby, if the agent should move towards the enemy, if it has enough energy, what to do if there is no enemy and so on. On the other hand, other actions do not need excessive planning like the `buy` action. When the specified Saboteur agent does not see any enemy agents nearby, it will upgrade its visibility range and health without considering more than just its energy and the available money. Similarly, 31 percent of all plans are associated with zoning. The reason for this is mainly due to the complexity of communication between agents and the many special cases to consider.

Figure 18 illustrates the amount of plans used for different actions. Similarly, the amount of plans executable only if the agent has a particular role vary a lot. Figure 19 illustrates this and shows that the Saboteur agent had more role-specific plans than e.g. for the Sentinel agent. The reason for this is that we have a complex, offensive strategy for our Saboteur agents but there are not

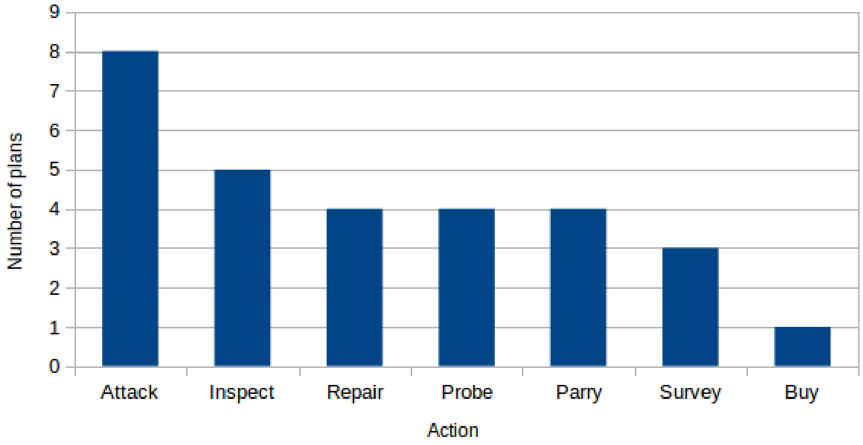


Fig. 18: Plan distribution for actions.

even Sentinel-exclusive actions. This figure does not count plans executable by all roles such as going to another vertex.

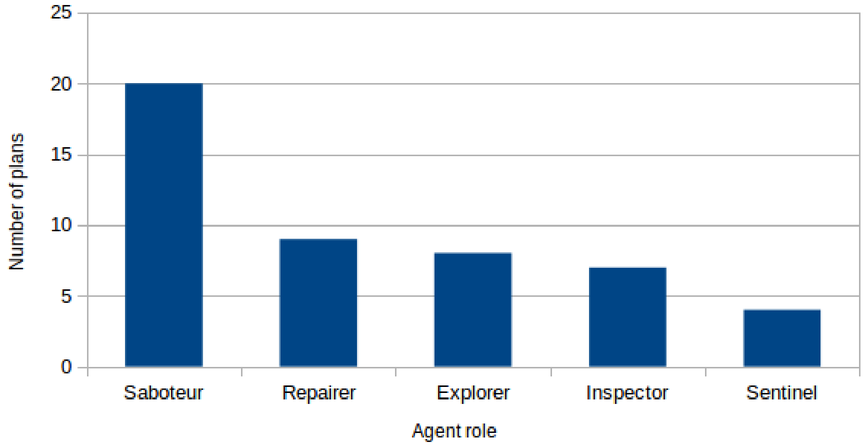


Fig. 19: Plan distribution for agents.

All those plans are loaded into the plan library. To determine when a plan is actually applicable, the *option selection function* $\mathcal{S}_{\mathcal{O}}$ is needed. It receives the plans whose triggering event match and matches them against the belief base. This ensures that it only reasons about plans which are executable in the current situation by inspecting their context. As a result, $\mathcal{S}_{\mathcal{O}}$ draws an applicable plan for an intention. The plans for the intentions are stored as a new intention if the plan was drawn as a first reaction to an event. Subgoals, triggered from within the bodies of other plans, are added to existing intentions. As the drawn plans match the contexts, they can be seen as partially instantiated [8].

Beliefs, desires and intentions have been introduced above in context of the Jason interpretation cycle. There are various agent languages implementing the BDI model. However, one of the reasons to choose Jason as programming language was that Jason provides an already implemented set of internal actions. These internal actions are furthermore extensible by user-defined internal actions, which are programmed in Java [8]. Besides the original internal actions like `.print` or `.send`, we developed 32 internal actions. Most of these internal actions are devoted to receiving information about the map, such as finding high value vertices. Hence, they do not play a big part in our development from a BDI perspective.

This section gave a more in-depth overview of the reasoning in Jason in the context of the BDI model. Jason's and AgentSpeak(L)'s implementation of the BDI agent architecture manage to close the gap between the theoretic model and a practically suitable language for multi-agent development. Nevertheless, we faced several problems during implementation which are further discussed in Section 6.2.

5 Team Organisation and Individual Tasks$^{\blacktriangle/\circ}$

This part presents the organisation of our team work. Over the course of about seven months, the team worked towards competing in the Multi-Agent Programming Contest 2014. Section 5.1 gives an overview on the team structure and interaction. It also describes the software tools used to communicate and cooperate. After this basic introduction, Sections 5.2 to 5.7 cover the individual work of each member in greater detail.

5.1 Basic 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. His tasks are explained in more detail in Section 5.6. Once every week, the team met in person to discuss the current progress and the upcoming course of action. 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 time slots 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 in real-time. If there was need for reconciliation, for example when tasks of different groups were closely interrelated or dependent on one another, group voice over IP calls were held. Towards 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 our versioning system. No team member had any prior experience with GitHub, although some had worked with Apache Subversion as a version control 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. [32]). 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] 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 "Agents on 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 summary, 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-

[1] `https://github.com/` – last accessed 24 October 2014

[2] `https://www.google.com/+/learnmore/hangouts/` – last accessed 24 October 2014

[3] `https://www.skype.com/` – last accessed 24.10.2014

[4] `http://jason.sourceforge.net/mini-tutorial/eclipse-plugin/` – last accessed 24 October 2014

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.

5.2 Artur Daudrich$^{\star,\circ}$

This section lists the work done by Artur Daudrich. Artur's contribution to the project was mainly the development and implementation of ideas and strategies for our map exploration and zoning strategies. At the beginning he implemented the interface between the MAPC server and our implementation. This included the passing of percepts from the server to our agents and the passing of actions from our agents to the server. He implemented a basic Java agent class to store data received from the server, like information about an agent's role, their basic energy value or their basic health value. As we at first worked with beliefs in Jason we did not use that until it was later revisited and used for our implementation of the map component in Java. After implementing the server-client communication and the server interface, Artur started working on percept reception and percept storage in the belief base of our agents. He developed some basic Jason plans for exploration which execute actions like `goto`, `probe` or `survey`.

Additionally, a first approach of reactive plans was developed by him to allow agents to react on external events. Such plans would e.g. consider a nearby enemy agent, so that our agent would then either defend itself, attack it or avoid enemy agents. Or they would observe their current energy to determine whether further actions were executable. As some team members implemented the cartographer agent approach, Artur adapted the plans to this new approach.

We later realised that the cartographer agent approach alone could not solve our performance issues. Artur then introduced the team to the idea of using a Distance-Vector Routing Protocol as shown in Section 3.3.2. While one group implemented the Distance-Vector Routing Protocol in AgentSpeak(L), Artur and Manuel Mittler worked on improving the storing and communication of agent percepts. They came up with the idea of bypassing the percept passing to the respective agents. Instead, the percepts would be sent from the server interface directly to the cartographer agent.

Artur then worked on his own approach for the map component presented in Section 3.3.3. His idea was to fully implement the map component in Java to benefit from the speed increase of pure Java. This happened in parallel to the part of the team which further developed map related tasks in AgentSpeak(L). As Artur's approach boosted our performance a lot, the map team active back then was disbanded. Artur and Michael Sewell then formed the new map team together. Our team always worked in small groups of two or three people, Artur and Michael Sewell worked together on most of the following objectives. They finished the map component in Java and adapted the Jason agents to work with the new map component. Also, they implemented many internal actions to allow Jason agents to communicate with the JavaMap.

In the meantime the whole team started to develop ideas for our zoning approach in our weekly team meetings. After defining the outlines of our zoning approach, Artur and Michael Sewell implemented the zone calculation as explained in Section 3.5.1. Accompanied by this was the realisation of representing and storing the information about currently built zones in the JavaMap component.

After that Artur was mainly working on the internal actions to allow Jason agents query zoning and map details. In the final part of our development Artur's task was to fix bugs and implement new features which were associated with the Java part of our code. This includes zoning, exploration, agents classes in Java and the interface to the server. In this documentation Artur wrote the sections he was mainly involved, which are Section 2.2, Section 3.1 and Section 3.3.

5.3 Sergey Dedukh$^\diamond$

This section lists the work done by Sergey Dedukh.

Sergey's contribution to the project was mainly related to development and testing of agent behaviour and communication in Jason. He started his work together with Manuel Mittler and Michael Ruster implementing the cartographer agent which is presented in Section 3.3.1 of this report. He primarily worked on the tasks related to correct percept handling, communicating and secure storing of beliefs in the cartographer agent and also development of auxiliary functions used in other agent strategies.

Continuing his work on the cartographer agent functionality he proposed and implemented the first exploring strategy based on a depth-first search algorithm. Later this algorithm was replaced by a Distance-Vector Routing algorithm during the development of the JavaMap agent, which is described in Section 3.3. During further development of exploring strategies Sergey together with Manuel Mittler implemented and tested the bidding algorithm of communication and argumentation. This approach allowed stable negotiation between agents and was used later on in several other agent strategies in our project. The theoretical background behind the bidding algorithm is described in Section 2.3.3.

While other team members implemented the exploration algorithms in Java he implemented and tested the zone defending strategy of saboteur agents. It used internal negotiation between saboteurs, making decisions and performing zone defensive tasks. This strategy was used in the zoning mode of the simulation and is described in more detail in Section 3.5.3 of this report.

After introducing the JavaMap, together with Artur Daudrich, he developed the repairing strategy, which included optimal assignment of repairers to disabled agents, moving towards each other and the repairing itself. More information on repairing strategy can be found in Section 3.4.

The last algorithm implemented by Sergey before the tournament, was detecting the “dead ends” of the graph (nodes with degree of one), and then probing them remotely by explorer agents during the exploring phase. This change allowed to save several steps for exploring the map and get achievement points for exploring faster, especially on sparse graphs, which were also present during the tournament.

During the competition Sergey actively participated in debugging of the issues that were found while competing against other teams.

5.4 Manuel Mittler$^{\odot}$

This section lists the work done by Manuel Mittler. He started in the project, together with Rahul and Sun, by familiarising himself with the Jason programming language and to implement initial actions like `survey` and `probe`. Afterwards, simple plans were developed to execute the actions. Additionally, he came up with the idea of following an aggressive strategy. The next thing he looked into was inter-agent communication to exchange information and to delegate tasks. This insight was used during the development of the cartographer agent together with Michael Ruster and Sergey. Also some effort was put into the map exploration task. Following this, Manuel worked together with Artur on improving the storing and allocating of agent percepts, because the team encountered that percepts frequently weren't in the belief base of agents that should have been there. They came up with the idea of bypassing the map-related percepts to the respective agents. Instead of forwarding the percepts from one agent to another, they were sent from the server interface directly to the cartographer agent. The next problem that Artur and Manuel tackled was that agents didn't always perform an action in every step. A hierarchy of actions for every agent type and fallback plans were developed that are executed if the agent is not capable to execute the designated plan, e.g. because of a lack of required information like waiting for a reply from another agent. After encountering issues when we started to communicate and negotiate between agents a lot, especially when it came to zone building, Manuel proposed to switch to another approach. He suggested to let the JavaMap class decide/calculate which are the best zones and then assign the agents to these zones instead of letting the agents find the best zones by themselves with a lot of negotiation. Best in this context means simply the best ratio of potential zone score to the number of agents which are necessary to build that zone. When he found out that disabled agents can still move, Manuel proposed to let the disabled agent move towards the repairer. This is beneficial in case the repairer agent is already involved in building a zone, because this zone would then not be destroyed. Later on, he and Michael Ruster started working on zone forming, maintenance and destruction. Manuel, together with Michael Ruster, also prepared and presented the final presentation.

5.5 Michael Sewell†

This section lists the work done by Michael Sewell, referred to as MS rather than Michael in the rest of this section to avoid confusion with Michael Ruster.

In the presentation phase at the beginning of the lab, MS gave a presentation on possible extensions to BDI methods, and he was the one to set up the GitHub git repository used by the team as their version control system and wiki. Once the coding phase of the project began, he was mostly responsible for implementing the Java classes and functions used to facilitate communication between agents in AgentSpeak/Jason, and for the AgentSpeak logic used by agents in general, or specific agents.

For a large part of the development time, MS worked together with Artur Daudrich in a kind of pair programming scenario using online screen sharing software. Most of this paired programming time was spent on implementing the MapAgent and other classes in Java and getting them to communicate with the AgentSpeak-based agents through AgentSpeak's internal actions. Generally, many of the functions in the Java MapAgent, Vertex and Agent classes were either created or modified by MS, and most of the internal actions were also written by him. He was responsible for writing a lot of the JavaDoc documentation for the Java-based part of the agent system and the comments in the AgentSpeak files that explained the code there. Artur and MS were also mostly responsible for thinking of and implementing the node-based zone calculation heuristic described in Section 3.5.1.

Besides the paired programming tasks, MS wrote many of the basic AgentSpeak plans used by agents, such as the parrying, repairing, attacking, and exploring-related actions. He also noticed and fixed issues that were related to Jason's inadequacies, such as that sometimes due to the Jason cycles taking too long, not only would agents sometimes miss out on sending an action to the server for a given step, but instead send multiple actions (the action for the previous and this turn) in one step, which would cause the MAKo team's agents to be "out of sync" with the server. He also decided on much of the order of the AgentSpeak plans in the `agent.asl` and other AgentSpeak files that define the agent behaviour, prioritizing such actions as getting repaired and moving away from enemy saboteur agents. MS implemented all inspecting-related behaviour for the Inspector agents, and was responsible for implementing the aggressive saboteur attacking behaviour used by the MAKo team. MS was also responsible for the MAKo team's agent upgrade buying behaviour, trying out different upgrade buying strategies by having the MAKo agents play against each other. During these matches, restricting the buying of upgrades to a single saboteur agent (the artillery agent) seemed to outperform any other approaches in terms of the final score, and this upgrade buying strategy is the one that made it into the final implementation used in the competition phase. During the actual tournament phase, the MAKo team was faced with a number of critical bugs that only became apparent when playing against other teams, such as an issue caused by dashes in enemy team names, which would cause AgentSpeak to interpret any enemy-related strings as arithmetic expressions. MS managed to fix many of these issues in the short time in-between matches, which ultimately allowed the MAKo team to score as well as it did.

In the preparation of this document, MS was responsible for primary or secondary proof-reading of most sections, in addition to writing the sections indicated by the † sign.

5.6 Michael Ruster°

This section lists the work done by Michael Ruster. He was the designated project leader after he proposed himself and was approved by the rest of the team. His leadership followed a democratic management style. Hence, decisions were made by the team as a whole through majority decisions. This helped team creativity while still keeping a structured working process in contrast to e.g. a laissez-faire approach. The encouragement of creativity was of special interest due to the inexperience of the team in this field. It supported new ideas and approaches which could then be immediately discussed and further developed as a group. An autocratic style on the other hand would have needed an expert in this field for wise delegations as well as him having some means of exerting pressure.

The project leader carried out the external communication with the supervisors and contest organisers. This includes e.g. the writing of the contest participation registration document. Initiated by Manuel Mittler, Michael worked on the slides for our final presentation which was held by both together. He ran and monitored the test simulations together with the organisers of the MAPC as well as the final simulations. Here, he also configured and maintained the server. Michael was also responsible for configuring the server and running test simulations. Furthermore, he regularly controlled the quality of the minutes and improved them when necessary. Similarly, Michael maintained the structure of the GitHub repository by introducing various folders, renaming and moving files. He wrote the guidelines and the table of contents of this report. Moreover, he incorporated and adapted the Springer lecture notes in computer science template[5] used for this report. In the beginning, he also started off with requirements engineering for a more formal approach to the software development. This effort was discarded by the team as many requirements were only discovered during the development and others changed frequently. Michael licensed the code after investigating the options given limited by the used external libraries.

In our development phase, Michael was chiefly concerned with map-related tasks. There, he initially implemented the cartographer agent which is presented in Section 3.3.1. He then continued maintaining this approach together with Sergey Dedukh and Manuel Mittler. When Artur Daudrich proposed the Distance-Vector Routing Protocol approach, Michael Ruster and Michael Sewell implemented, tested and improved the node agents. Both are explained in Section 3.3.2. Manuel and Michael Ruster later started working on zone forming, maintenance and destruction. After Manuel had to take a pause due to personal matters, Michael continued implementing the zone logics as presented in Section 3.5.2 and Section 3.5.3. As agent plans for the zone building phase used various internal actions accessing the JavaMap shown in 3.3.3, Michael was also active in fixing bugs in the Java code. We were not able to properly handle the end of a simulation which is indicated by the receipt of the `SIM-END` message [2]. The problem was that it was a complex task to fully reset our local simulation and restoring the initial state of agent knowledge. Here, Michael implemented a

[5] `https://www.springer.com/computer/lncs/lncs+authors` – last accessed 17 November 2014

workaround. It made our multi-agent system shut down completely on receipt of the `SIM-END` message. The system was then started anew after giving the MAPC server some time to restart the simulation. This was done by using a bash shell script.

5.7 Yuan Sun[▲,◇/○]

This section lists the work done by Yuan Sun. Yuan mainly focused on implementing actions of agents in this project. At the beginning, she started to learn how to use Jason programming language together with Manuel Mittler and Rahul Arora. Rahul later left the research lab.

In addition, there was a Java factory class called `ActionHandler` which was used to instantiate a corresponding `Action` object according to a given identifier of type string. It aimed to separate the creation and representation of concrete `Action` objects. However, the actions represented as objects were not diverse enough to justify the creation of subclasses for each action. Hence, a dedicated factory class was not necessary. Yuan deleted this Java factory class and then encapsulated the progress of object instantiation inside the `Action` constructor. The constructor was able to return an `Action` object according to a given parameter, `entityName`, so that no complex switch statement would be necessary every time a new `Action` object was needed. Therefore, the degree of coupling of the system was decreased.

In the initial phase of agent strategy development in Jason, the implementation of different actions was done in a single file. Yuan rewrote and reorganised some of the code in different files with names corresponding to the roles of different agents.

Yuan implemented initial actions like `probe`, `attack`, `repair` and `parry` with simple strategies while the other team members were working on map- and zone-related tasks. This enabled the other team members sooner to monitor and analyse the functioning of their code. She also improved the contexts of plans which lead to the execution of these actions. For example, she added the checks to determine a sufficiently high energy level to all agents before action execution as well as checks for the visibility range limitation for ranged actions.

Moreover, with further project development, Yuan continued implementing some more complex agent strategies after discussing them with other team members. For instance, previously, sentinel agents only executed the `parry` action when they were attacked by enemies. After some discussion, Yuan implemented a better strategy for the these sentinel agents. If a sentinel agent was occupying a node in order to form a zone, it would execute the `parry` action when it saw enemies nearby although it had not yet been attacked by the enemies.

In the final technical report, Yuan wrote the sections related to the Belief-Desire-Intention model. She did a presentation about Belief-Desire-Intention model before the project start. In the report, she did not only introduce the theory of the Belief-Desire-Intention model (see Section 2.3.1), but also did some statistics to analyse the use of the Belief-Desire-Intention model in our project (see Section 4).

6 Discussion and Conclusion $^{\circ}$

In this last part, we reflect on the actual competition as well as our work. First, Section 6.1 presents the setup and procedure of the competition. The results of the competition are presented. Moreover, some general observations we made while monitoring the behaviour of our agents throughout the matches are given. Section 6.2 builds upon these observations by explaining in more detail what we saw and which conclusions can be drawn from this. Besides, the Section also discusses more general lessons we have learned while developing our multi-agent system.

Fill this part

6.1 Competition Results$^{\odot/\circ}$

This section focuses on the actual competition and presents its setup as well as its results. The 2014 MAPC took place on the 15th and 17th of September. Each team had to play three simulations against all other teams. Every simulation ran for 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.

if you proofread this, make sure to also proofread the next subsection and decide if there is a great overlap and whether we need to merge them.

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. Section 20 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 Section 21 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 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 in the appendix.

As Matthias commented: you can't see a thing on this graphic! Redo it or improve it.

As Matthias commented: you can't see a thing on this graphic! Redo it or improve it.

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 fully restarted automatically after each round and there were no such problems

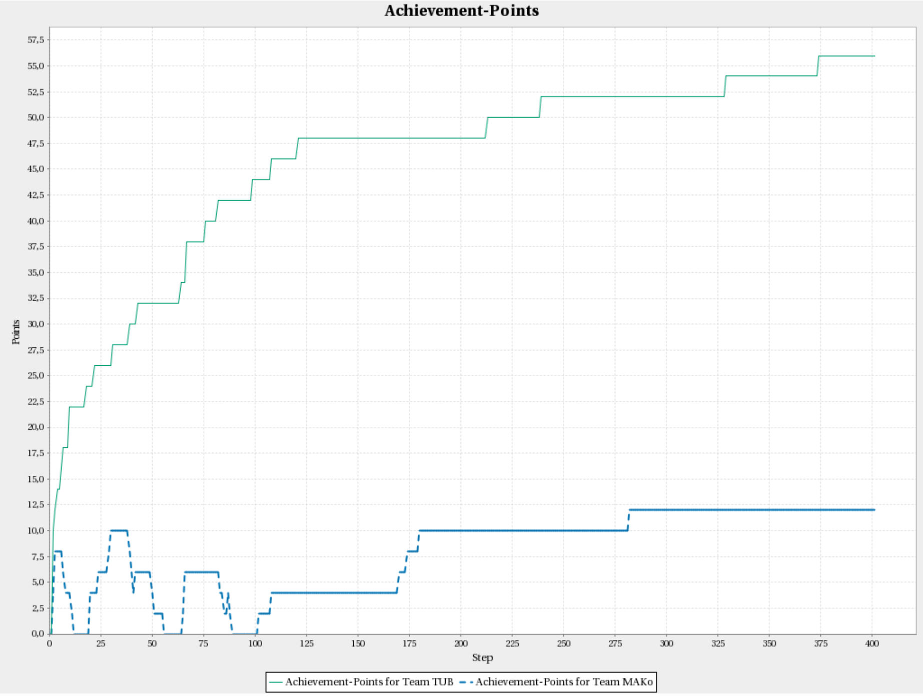


Fig. 20: Achievement points from the third match TUB against MAKo.

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

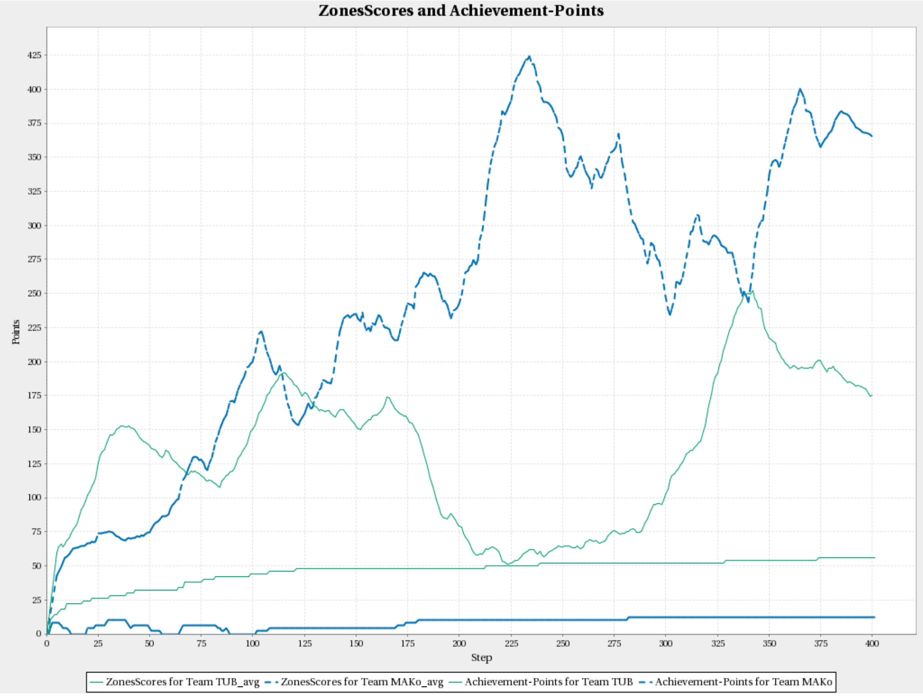


Fig. 21: Combined achievement and zone scores from the third match TUB against MAKo.

detect articulation points. In the course of the matches, we saw that multiple maps favoured such an approach. **??** 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 articulation points would have enabled us to form greater zones with fewer agents than what our algorithm calculated. Nevertheless, the general idea regarding small zone forming proved 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.

6.2 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 members had any experience with Jason as a programming language before the research lab. Similarly, we

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.

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 imperatively programmed 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 3.3.1. 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 was 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 3.3.2, 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 next day of the competition, 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 the programming for this project, 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 the conclusion that all our major problems were somehow connected to Jason. In the end, it is the opinion of the team that we would have profited from starting from scratch using e.g. an entirely Java-based approach rather than working with Jason. But in the end, we definitely learned a lot in terms of logic programming, multi-agent systems and their development and of course Jason in particular. Throughout the development phase, we quickly tried to store the shared graph information in a central place. Although there were still problems caused by communication overhead, we learned that this was easier to manage than having agents communicate graph information amongst themselves.

Considering the "Agents on Mars" scenario scenario, we saw that our aggressive attacking strategy featuring the artillery agent was effective. Furthermore, we realised that a completely different zoning approach could have been taken if we had focused on exploiting articulation points. In the end, we are happy with our result in the competition and the gathered experience.

@manuelmittler: add the references and link to the appendix

References

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

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

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

[4] Carlos E. Alchourron, Peter Gardenfors, and David Makinson. "On the Logic of Theory Change: Partial Meet Contraction and Revision Functions". In: *Journal of Symbolic Logic* 40.2 (1985), pp. 510–530.

[5] Krumnack Antje et al. "Efficiency and Minimal Change in Spatial Belief Revision". In: *Proceedings of the 33rd Annual Conference of the Cognitive Science Society.* 2011, pp. 2270–2275.

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

[7] Rafael H. Bordini. *Lecture on Programming Multi-Agent Systems in AgentSpeak Using Jason.* `http://community.dur.ac.uk/p.h.shaw/teaching/mas/lectures/rafael/MAS-Lecture05-6up.pdf`. [Online; accessed 02 December 2014]. 2008.

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

[9] 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.

[10] 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.

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

[12] 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.

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

[14] Michael E. Bratman, David J. Israel, and Pollac Martha E. "Plans and resource-bounded practical reasoning". In: *Computational Intelligence* 4.3 (1988), pp. 349–355.

[15] Lars Braubach, Alexander Pokahr, and Kai Jander. *BDI User Guide: Chapter 3 Agent Specification*. http://www.activecomponents.org/bin/view/BDI+User+Guide/07+Goals. [Online; accessed 22 October 2014]. 2010.

[16] Lars Braubach, Alexander Pokahr, and Kai Jander. *BDI User Guide: Chapter 3 Agent Specification*. http://www.activecomponents.org/bin/view/BDI+User+Guide/03+Agent+Specification. [Online; accessed 22 October 2014]. 2010.

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

[18] Rutgers University C. Hedrick. *Routing Information Protocol*. http://www.ietf.org/rfc/rfc1058.txt. [Online; accessed 20 October 2014]. 1988.

[19] FIPA TC Communication. *FIPA ACL Message Structure Specification — Version G*. http://www.fipa.org/specs/fipa00061/SC00061G.html. [Online; accessed 24 October 2014]. 2002.

[20] James P. Delgrande. "Revising by an Inconsistent Set of Formulas". In: *IJCAI'11*. 2011, pp. 833–838.

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

[22] Mark D'Inverno et al. "The dMARS Architecture: A Specification of the Distributed Multi-Agent Reasoning System". In: *Autonomous Agents and Multi-Agent Systems* 9 (2004), pp. 5–53.

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

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

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

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

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

[28] 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.

[29] Francois F. Ingrand, Anand S. Rao, and Michael P. Georgeff. "An architecture for real-time reasoning and system control". In: *IEEE Expert* 7 (1992), pp. 34–44.

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

[31] 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.

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

[33] 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.

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

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

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

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

[38] Patrick D. O'Brien and Richard C. Nicol. "FIPA—towards a standard for software agents". In: *BT Technology Journal* 16.3 (1998), pp. 51–59.

[39] 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.

[40] 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.

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

[42] Anand S. Rao. "AgentSpeak(L): BDI Agents Speak out in a Logical Computable Language". In: *Proceedings of the 7th European Workshop on Modelling Autonomous Agents in a Multi-agent World : Agents Breaking Away*. MAAMAW '96. Einhoven, The Netherlands: Springer-Verlag New York, Inc., 1996, pp. 42–55. ISBN: 3-540-60852-4.

[43] Anand S. Rao and Michael P. George. "BDI Agents: From Theory to Practice." In: *Proceedings of the First International Conference on Multi-Agent Systems ICMAS*. 1995, pp. 312–319.

[44] Anand S. Rao and Michael P. Georgeff. "Modeling rational agents within a BDI-architecture". In: *KR* 91 (1991), pp. 473–484.

[45] 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.

[46] 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.

[47] 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.

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

[49] 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.

[50] 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).

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

[52] 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.

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

[54] 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.

[55] 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.