TU Clausthal

Clausthal University of Technology

Multi-Agent Programming Contest 2011 Edition Documentation

Tristan Behrens, Jürgen Dix, Jomi Hübner, Michael Köster, Federico Schlesinger

IfI Technical Report Series IfI-12-01

IfI

Department of Informatics
Clausthal University of Technology

Impressum

Publisher: Institut für Informatik, Technische Universität Clausthal
Julius-Albert Str. 4, 38678 Clausthal-Zellerfeld, Germany
Editor of the series: Jürgen Dix
Technical editor: Federico Schlesinger
Contact: federico.schlesinger@tu-clausthal.de
URL: http://www.in.tu-clausthal.de/forschung/technical-reports/
ISSN: 1860-8477

The IfI Review Board

Multi-Agent Programming Contest
2011 Edition

Documentation

Tristan Behrens, Jürgen Dix, Jomi Hübner, Michael Köster, Federico Schlesinger

Abstract

This document is intended to convey all knowledge that is required to successfully participate in the Multi-Agent Programming Contest 2011. We initially provide a scenario description that contains all relevant information about the environment. Based on this scenario, it is described all the technical specifications required to develop agents to participate in the contest. Detailed instructions about using the software provided by the organization is then included. This software simplifies all the process of developing agents for the *MASSim*-platform, participants can thus focus on the development of good strategies for the contest.

1 Introduction

The Multi-Agent Programming Contest is an attempt to stimulate research in the area of multi-agent system (MAS) development and programming (MAP) by

1. identifying key problems,
2. collecting suitable benchmarks, and
3. gathering test cases which require and enforce coordinated actions,

that can serve as milestones for testing multi-agent programming languages, platforms and tools. We also expect that participating at the contest helps to debug existing systems and to identify their weak and strong aspects.

The Multi-Agent Programming Contest was initiated in 2005 and since then it has passed through three distinct phases. The first phase began in 2005 with the "food-gatherers"-scenario, where a pre-specified multi-agent system had to be implemented. These MASs were later examined in order to determine the winner. From 2006 - 2007 the "goldminers"-scenario was used. This time it is provided an environment by means of an online-architecture, and automatically determined the winner. Then from 2008 - 2010 we ran the "cows and cowboys"-scenario, again on the same online-architecture.

We noticed that most approaches used in the agent contest in the last years were *centralized*, contrary to the philosophy of multi agent programming (MAP). Even the accumulated knowledge of the agents was maintained centrally and shared by internal communication. This aspect has motivated the definition of a new scenario.

For the 2011 Contest we would like to highlight the two following key problems that the participants should keep in mind when developing their multi-agent system:

1. *agent cooperation* and *agent coordination* is encouraged even more, and
2. *team decentralization* should be taken into account.

We think, developing good strategies for these two key problems helps to perform well in the Contest.

This document is organized as follows. While the first two chapter contains the specification and the rules of the contest, the latter ones are focused on how to use the software provided to the participants. Chapter 2 contains a full description of the Multi-Agent Programming Contest 2011 scenario called *Agents on Mars*. There we describe the environment and the semantics of the environment's evolution. In the chapter 3 it is explained how you can connect your agents on the lowest level, that is that you are supposed to faithfully implement the client-server communication-protocol. This means that you are expected to establish and maintain an authenticated TCP/IP connection to the *MASSim*-server and communicate with it by exchanging XML-messages. In chapter 4 we explain how to start the server that simulates the scenario providing perception and action for the agents. Details about setting up a tournament in a local computer is also covered in this chapter. Additionally, in chapter 5 we explain different means for developing agents with the *MASSim*-platform, each one on a distinct level of abstraction. The first proposal is to use EIS[1]-compatible *environment-interface* that implements the aforementioned communication protocol. This very interface establishes and maintains authenticated connections to the server and reduces acting and perceiving to invoking Java-methods and evaluating call-backs. Finally, it is described how to develop a very simplistic *dummy*-agent team that we have implemented for testing the environment. Users can of course extend these agents with the artificial intelligence they have in mind. These dummy-agents make use of the EIS interface.

[1]`http://sf.net/projects/apleis`

2 Scenario Description

In the following, we provide a detailed description of the Multi-Agent Programming Contest 2011 scenario. The overall goal of the game is to control zones of a map (graph) by placing agents on appropriate positions.

2.1 Background Story

In the year 2033 mankind finally populates Mars. While in the beginning the settlers received food and water from transport ships sent from earth shortly afterwards – because of the outer space pirates – sending these ships became too dangerous and expensive. Also, there were rumors going around that somebody actually found water on Mars below the surface. Soon the settlers started to develop autonomous intelligent agents, so-called All Terrain Planetary Vehicles (ATPV), to search for water wells. The World Emperor – enervated by the pirates – decided to strengthen the search for water wells by paying money for certain achievements. Sadly, this resulted in sabotage among the different groups of settlers.

Now, the task of your agents is to find the best water wells and occupy the best zones of Mars. Sometimes they have to sabotage their rivals to achieve their goal (while the opponents will most probably do the same) or to defend themselves. Of course the agents' vehicle pool contains specific vehicles, some of them have special sensors, some of them are faster and some of them have sabotage devices on board. Last but not least, your team also contains special experts, the repair agents, that are capable of fixing agents that are disabled. In general, each agent has a special expert knowledge and is thus the only one being able to perform a certain action. So your agents have to find ways to cooperate and coordinate themselves.

2.2 The Challenge

In this year's Contest the participants have to compete in an environment that is constituted by a graph where the vertices have an unique identifier and also a number that determines the value of that vertex. The weights of the edges on the other hand denotes the costs of traversing the edge.

A *zone* is a subgraph (with at least two nodes) whose vertices are colored by the graph coloring algorithm introduced in Section 2.3. If the vertices of a zone are colored with a certain team color it is said that this team occupies this area. The value of a zone determined by the sum of its vertices' values. Since the agents do not know a priori the values of the vertices, only probed vertices contribute with their full value to the zone-value, unprobed ones only contribute one point.

The goal of the game is to maximize the score. The score is computed by summing up the values of the zones and the current money (cf. Section 2.7)

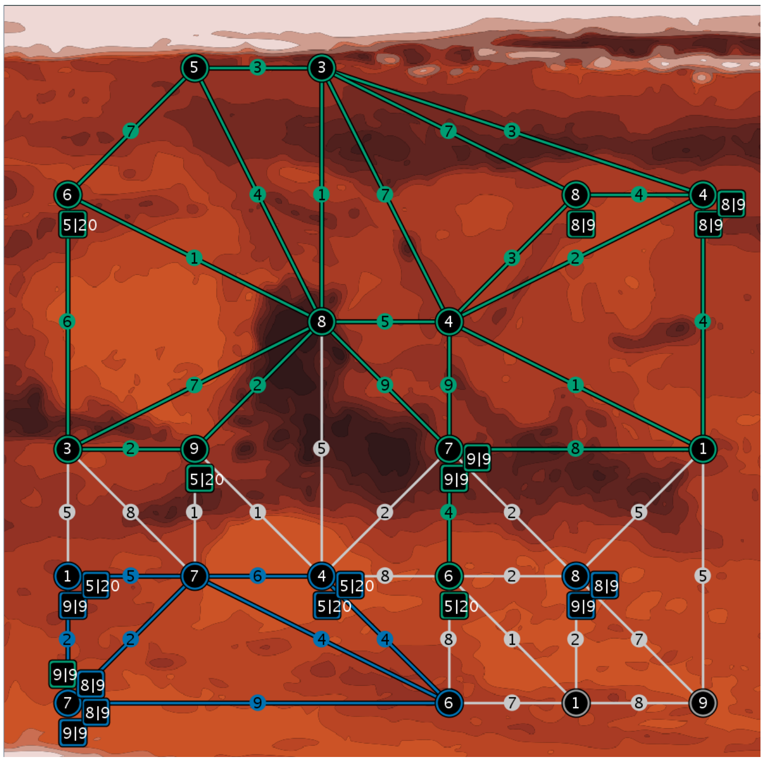

Figure 1: A screenshot.

for each simulation step:

$$\texttt{score} = \sum_{s=1}^{\texttt{steps}} (\texttt{zones}_s + \texttt{money}_s)$$

Where `steps` is the number of simulation steps, and $\texttt{zones}_s$ and $\texttt{money}_s$ are the current sum of all zone values and the current amount of money respectively.

Figure 1 shows such a scenario. The numbers depicted in the vertices describe the values of the water wells while the distance of two water wells is labeled with travel costs. The green team controls the green zone while the blue team has the smaller blue zone. The value of the blue zone, assuming that all vertices have been probed by the blue team, is 24.

2.3 Graph Coloring Algorithm

The graph coloring algorithm is used to determine the zones that a team is occupying. We firstly present the formal definition and afterwards explain it via an example.

Definition 2.1. Let V be the set of vertices, E the set of edges, ag the set of agents, and T the set of team names. Furthermore let $ag(v)$ denote the set of agents standing on vertex $v \in V$. A *graph coloring* is a mapping

$$c : V \rightarrow T \cup \{none\}.$$

The coloring is subject to change over time. We say that a vertex v is colored if $c(v) \neq none$. The coloring is determined by the following calculation, consisting of phases that are executed sequentially:

1. The first phase of the calculation only involves the coloring of vertices that have agents standing on them. $c(v) = t$ iff $ag(v) \neq \emptyset$ and t is the name of the team that dominates the vertex. We say that a vertex v is dominated by t if t has the majority of agents on v. If no team dominates the vertex, then $c(v) = none$.

2. The coloring is extended to empty vertices that are direct neighbors of dominated vertices. Formally, $c(v) = t$ if $ag(v) = \emptyset$, t is the name of the team that dominates the largest subset of neighbors

$$S_t = \{v_n \mid (v, v_n) \in E, c(v_n) = t, c(v_n) \neq none, ag(v_n) \neq \emptyset\}$$

   of v, with $\mid S_t \mid > 1$. Note that a team needs to dominate at least two neighboring vertices of an empty vertex to be able to color that empty vertex.

3. Some of the vertices that where colored with a team name t in the previous two steps might represent a *frontier* that isolates a part of the graph from all the other teams' agents. We say that an empty vertex v has been isolated by a team t (and thus $c(v) := t$) iff for all agents ag belonging to a team t', where $t' \neq t$, there is no path from ag_n to v that does not include a vertex v' such $c(v') = t$.

4. $c(v) := none$ iff the other conditions are not satisfied.

For the coloring algorithm, we are only consider the agents that are not `disabled`. The definition of disabled agents is given later in Section 2.6.

An example of graph coloring in an hypothetical world configuration is depicted in Figure 2. Pictures (a), (b) and (c) show the result of executing the coloring calculation phases 1, 2 and 3 respectively. For the sake of improving visibility, all edges whose two vertices are colored in the same team's color, are also shown in that same color, but internally this has neither meanings nor implications.

In detail, phase 1 colors such vertices in a certain color regarding the color of the majority of agents. For instance, in Figure (a) the top right vertex is colored in green because there are three green agents but only one red agent standing on that vertex. When there is a draw the vertex does not belong to a team.

In phase 2 (Figure (b)) we look at the direct neighbors of the already colored vertices. We color such a neighbor in a certain team color when there is an edge from this uncolored vertex to at least two other vertices that are colored in that particular team color. We are taking again the majority into account, i.e., the color of the vertex is finally determined by counting for each team color the connected vertices and choosing the best result. If there is a draw the vertex is not colored at all.

Phase 3, finally, colors all vertices that are not reachable by other teams without crossing the already colored vertices. One can see it as a border that is separating parts of the graph. After executing phase 3 we have defined the zones of all teams.

Picture (b) clearly shows how the green team has built a closed frontier around a set of empty vertices, which are then colored in picture (c). In picture (d), an agent of the red team has "broken" the frontier, making some of the vertices inside of it not isolated anymore.

2.4 Teams & All Terrain Planetary Vehicles

We define five roles (see Table 1), where each role describes the available actions (actions an agent can perform) for the All Terrain Planetary Vehicle (ATPV), its maximum energy, its maximum health, its strength and its visibility range. While the energy is important for executing actions, the health determines whether an agent is still able to perform all actions or just a small subset. The strength defines how strong a sabotage will be and the visibility range describes how far an agent can see. The concrete actions are described in Section 2.5. The teams consist of 10 agents and for each pair we assign them the following roles:

2.5 Agent Actions

An agent can perform some of the following actions regarding its role. The result of that action is perceived by the agent automatically, i.e., the information is sent to it in the next percept.

`skip` This action is always `successful`, costs no resources and the effect is that the agent does not do anything.

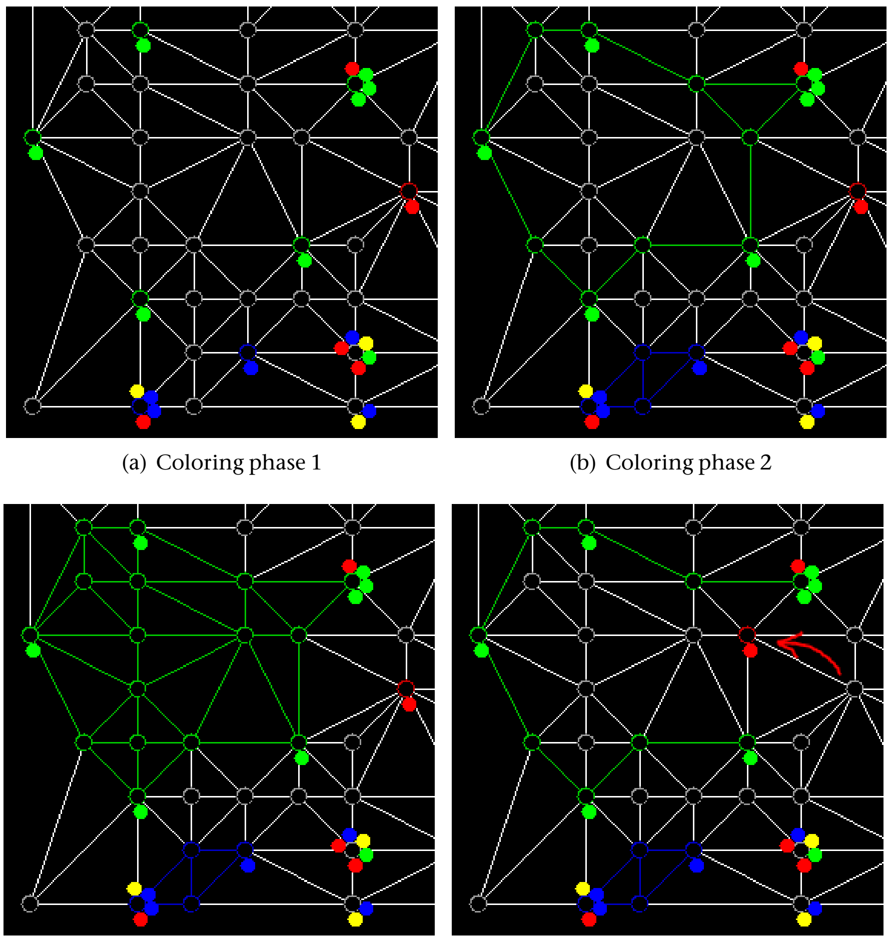
(a) Coloring phase 1
(b) Coloring phase 2
(c) Coloring phase 3
(d) Breaking a frontier.

Figure 2: Coloring phases

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

Table 1: The different roles.

`recharge` This action increases the current energy of the agent by 20 percent. The result can be `successful`, or `failed` if attacked by an opponent. Of course, it does not cost any resources.

`attack` If an agent wants to sabotage some other agent it has to perform this action. The action requires a parameter (the identifier of the target) and can be `successful`, or `failed` because of lack of energy. Also, it can be `parried`. Lastly, it can fail because of a wrong parameter, i.e., `wrongParameter` is the result. Also, the current energy is decreased about 2 points.

`parry` This action parries an attack and costs 2 points of energy points. Only if an attack is actually taking place it is `successful`, otherwise it is `useless`. The action can also fail because of too low energy, i.e., the result is `failed`.

`goto` The agent moves from one vertex to another by executing this action.

The reduction of the current energy is determined by the traveling cost, i.e., the weight of an edge. The action needs a parameter, namely the id of the vertex it wants to go to. The result of that action is, `failed` when the current energy is too low and the current energy is decreased by 1, `wrongParameter` if the parameter is incorrect or `successful` otherwise.

`probe` Without the team knowing the exact value of the node, it is set to 1 when it comes to the computation of the zone score. Only if at least one agent of the team analyzes the water well the team gets the full value of that vertex (the value is then incorporated in the next percept). The action costs 1 points of energy. A `probe` action can fail (result: `failed`) for different reasons: lack of energy or being attacked by another agent. Otherwise it is `successful`.

`survey` With this action (costs: 1) the agent can get the weights of the edges (in the next percept). If the action is not `successful` it `failed` because of too low energy or the agent was attacked in that moment.

`inspect` This action (costs: 2) inspects all opponents (the internals) on the vertex the agent is standing at the moment as well as all direct neighbors. The result can be again `failed` because of lack of energy or being attacked, and is `successful` otherwise even if there is no agent at all.

`buy` The `buy` action (costs: 2) is more complex. It's purpose is to increase your agent's maximum health, maximum energy, visibility range or maximum strength buy spending money (cf. Section 2.7) on extension packs. The possible values for the parameter are: `battery` (increases maximum energy and current energy by 1), `sensor` (increases visibility range by 1), `shield` (increases maximum health and current health 1) or `sabotageDevice` (increases the strength by 1). Of course, it fails if your agent is not allowed (determined by the role) to wear a `sabotageDevice`. Also, it fails when being attacked or if the current energy is too low. Finally, if the parameter is syntactically wrong the result is `wrongParameter` and otherwise it is `successful`.

`repair` This action (costs: 2) repairs a teammate. Note that an agent cannot repair itself. The parameter determines which agent gets repaired. If the value is syntactically wrong the result is `wrongParameter`. If there is no such agent it is `failed`. It also fails because of too low energy. Otherwise it is `successful`.

In general, an action can fail with a certain probability (1 percent). In this case the action is considered as the `skip` action (and the perceived result will be `failed`). Actions that are performed by an agent but do not correspond to the agent's role fail as well.

2.6 Disabled Agents

Agents whose health drops to zero, are disabled, i.e., only the action `goto`, `repair`, `skip` are executable (if the role allows that). The `recharge` action is also allowed to be performed, but its recharge rate is set to 10 percent.

2.7 Money

If a team reaches a milestone, its money is increased. We have different achievements, for example:

- having zones with fixed values, e.g. 10 or 20,
- fixed numbers of probed vertices, e.g. 5 or 10,
- fixed numbers of surveyed edges, e.g. 10 or 20,
- fixed numbers of inspected vehicles, e.g. 5 or 10,
- fixed numbers of successful attacks, e.g. 5 or 10, or
- fixed numbers of successful parries, e.g. 5 or 10.

Note that the exact achievements are defined in the configuration. The exact details about achievements are still object to change, in the upcoming phase of balancing. We are going to announce them later.

2.8 Percepts

In every step, the agents get these percepts:

- state of the simulation, i.e. the current step,
- state of the team, i.e. the current scores and money,
- state of the vehicle, i.e. its internals as described above,
- visible vertices, i.e. identifier and team,
- visible edges, i.e. its vertices' identifiers,
- visible vehicles, i.e. its identifier, vertex, team,
- probed vertices, i.e. its identifier and its value,
- surveyed edges, i.e. its vertices' identifiers and weight, and
- inspected vehicles, i.e. its identifier, vertex, team and internals.

Please refer to the protocol description for the details about percepts.

We also have the notion of *shared percepts*. Agents of the same team that are in the same zone share their percepts, that is visible vertices, edges and vehicles, and probed vertices, surveyed edges and inspected vehicles.

2.9 Simulation state transition

The simulation state transition is as follows:

1. collect all actions from the agents,
2. let each action fail with a specific probability,
3. execute all remaining `attack` and `parry` actions,
4. determine disabled agents,
5. execute all remaining actions,
6. prepare percepts,
7. deliver the percepts.

3 Agent-Server Communication

The agents from each participating team will be executed locally (on the participant's hardware) while the simulated environment, in which all agents from competing teams perform actions, runs on the remote contest simulation server.

Agents communicate with the contest server using standard TCP/IP stack with socket session interface. The Internet coordinates (IP address and port) of the contest server (and a dedicated test server) will be announced later via the official contest mailing list.

Agents communicate with the server by exchanging XML messages. Messages are well-formed XML documents, described later in this document. We recommend using standard XML parsers available for many programming languages for generation and processing of these XML messages. Note that ill-formed messages, that is messages that do not comply to the message-syntax outlined here, are ignored.

3.1 Communication Protocol Overview

Logically, the tournament consists of a number of matches. A match is a sequel of simulations during which several teams of agents compete in several different settings of the environment. However, from agent's point of view, *the tournament consists of a number of simulations in different environment settings and against different opponents*.

The tournament is divided into three phases:

1. the initial phase,

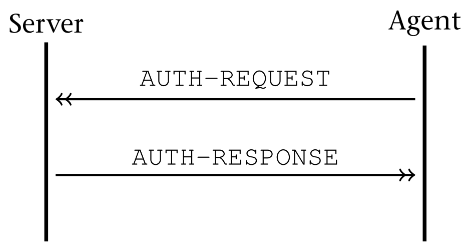

Figure 3: The initial phase.

2. the simulation phase, and
3. the final phase.

During the initial phase, agents connect to the simulation server and identify themselves by providing their username and password (`AUTH-REQUEST` message). Credentials for each agent will be distributed in advance via e-mail. As a response, agents receive the result of their authentication request (`AUTH-RESPONSE` message) which can either succeed, or fail. After successful authentication, agents should wait until the first simulation of the tournament starts.

Fig. 3 shows a picture of the initial phase (UML-like notation).

At the beginning of each simulation, agents of the two participating teams are notified (`SIM-START` message) and receive simulation specific information.

In each simulation step each agent receives a perception about its environment (`REQUEST-ACTION` message) and should respond by performing an action (`ACTION` message).

The agent has to deliver its response within the given deadline. The action message has to contain the identifier of the action, the agent wants to perform, and action parameters, if required.

Fig. 4 shows a picture of the simulation phase.

When the simulation is finished, participating agents receive a notification about its end (`SIM-END` message) which includes the outcome of the simulation.

All agents which currently do not participate in a simulation should wait until the simulation server notifies them about either 1) the start of a simulation, they are going to participate in, or 2) the end of the tournament.

At the end of the tournament, all agents receive a notification (`BYE` message). Subsequently the simulation server will terminate the connections to the agents.

Fig. 5 shows a picture of the final phase.

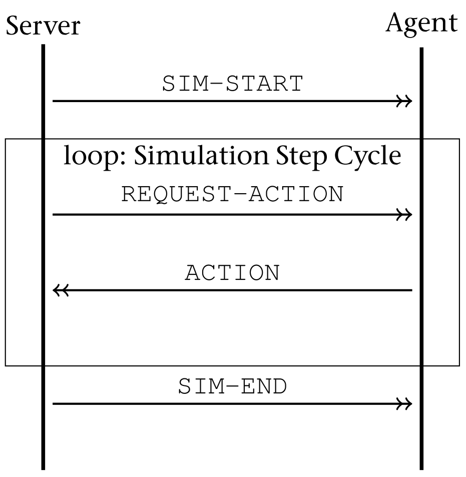

Figure 4: The simulation-phase.

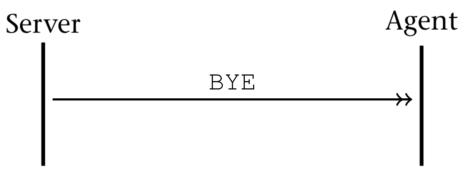

Figure 5: The final phase.

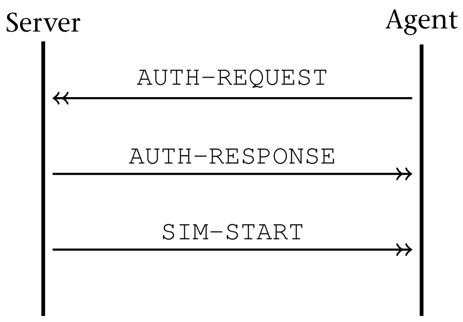

Figure 6: Reconnecting.

3.1.1 Reconnection

When an agent loses connection to the simulation server, the tournament proceeds without disruption, only all the actions of the disconnected agent are considered to be empty (*skip*). Agents themselves are responsible for maintaining the connection to the simulation server and in a case of connection disruption, they are allowed to reconnect.

Agents reconnect by performing the same sequence of steps as at the beginning of the tournament. After establishing the connection to the simulation server, an agent sends an `AUTH-REQUEST` message and receives an `AUTH-RESPONSE`. After successful authentication, the server sends the agent the `SIM-START` message. If the agent participates in a currently running simulation, the `SIM-START` message will be delivered immediately after the `AUTH-RESPONSE`. Otherwise the agent will wait until the start of the next simulation in which it participates. In the subsequent step when the agent is picked to perform an action, it receives the standard `REQUEST-ACTION` message containing the perception of the agent at the current simulation step, and the simulation proceeds in a normal mode.

Fig. 6 shows a picture of the reconnection.

3.1.2 XML Messages Description

XML message structure The XML messages exchanged between server and agents are zero terminated UTF-8 strings. Each XML message exchanged between the simulation server and agent consists of three parts:

- Standard XML header: Contains the standard XML document header

```
<?xml version="1.0" encoding="UTF-8"?>
```

- Message envelope: All XML messages have `<message>` as the root element. Its attributes are the timestamp and a message type identifier.
- Message separator: Note that because each message is a UTF-8-zero-terminated string, messages are separated by nullbyte.

The timestamp is a numeric string containing the status of the simulation server's global timer at the time of message creation. The unit of the global timer is milliseconds and it is the result of standard system call "time" on the simulation server (measuring number of milliseconds from January 1st, 1970 UTC). The message type identifier is one of the following values: `auth-request`, `auth-response`, `sim-start`, `sim-end`, `bye`, `request-action`, `action`.

Messages sent from the server to an agent contain all attributes of the root element. However, the timestamp attribute can be omitted in messages sent from an agent to the server. In the case it is included, server silently ignores it.

Example of a server-2-agent message:

```
<message timestamp="10001980000000" type="request-action">
  <!-- optional data -->
</message>
```

Example of an agent-2-server message:

```
<message type="auth-request">
  <!-- optional data -->
</message>
```

Depending on the message type, the root element `<message>` can contain simulation specific data.

`AUTH-REQUEST` **(agent-2-server)** When an agent connects to the server, it has to authenticate itself using the username and password provided in advance by the contest organizers. This way we prevent the unauthorized use of connections belonging to a contest participant. `AUTH-REQUEST` is the very first message an agent sends to the contest server.

The message envelope contains one element `<authentication>` without subelements. It has two attributes `username` and `password`.

Example:

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<message type="auth-request">
  <authentication password="1" username="a1"/>
</message>
```

AUTH-RESPONSE **(server-2-agent)** Upon receiving an AUTH-REQUEST message, the server verifies the provided credentials and responds by a message AUTH-RESPONSE indicating success, or failure of authentication. It has one attribute timestamp that represents the time when the message was sent.

The envelope contains one <authentication> element without subelements. It has one attribute result of type string and its value can be either "ok", or "fail". Example:

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<message timestamp="1297263037617" type="auth-response">
  <authentication result="ok"/>
</message>
```

SIM-START **(server-2-agent)** The simulation starts by notifying the corresponding agents about the details of the starting simulation. This notification is done by sending the SIM-START message.

The data about the starting simulation is contained in one <simulation> element with the following attributes:

- the number of edges,
- the number of vertices,
- the id of the simulation, and
- the number of steps the simulation will last.

One step involves all agents acting at once. Therefore if a simulation has n steps, it means that each agent will receive n REQUEST-ACTION messages during the simulation (assuming a stable connection to the server).

Example:

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<message timestamp="1297263004607" type="sim-start">
  <simulation edges="47" id="0" steps="500" vertices="20"/>
</message>
```

SIM-END **(server-2-agent)** Each simulation lasts a certain number of steps. At the end of each simulation the server notifies agents about its end and its result.

The <sim-result>-tag has two attributes. ranking is the ranking of the team and score is the final score.

Example:

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<message timestamp="1297269179279" type="sim-end">
  <sim-result ranking="2" score="9"/>
</message>
```

`BYE` **(server-2-agent)** At the end of the tournament the server notifies each agent that the last simulation has finished and subsequently terminates the connections. There is no data within the message envelope of this message.
Example:

```
<?xml version="1.0" encoding="UTF-8"?>
<message timestamp="1204978760555" type="bye"/>
```

`REQUEST-ACTION` **(server-2-agent)** In each simulation step the server asks the agents to perform an action and sends them the corresponding perceptions.

This message, due to its complexity, is best explained using an example. Note, however, that the following message is an artificial one, which has never been sent by the server:

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<message timestamp="1297263230578" type="request-action">
  <perception deadline="1297263232578" id="201">
    <simulation step="200"/>
    <self energy="19" health="9" lastAction="skip"
    lastActionResult="successful" maxEnergy="19"
    maxEnergyDisabled="9" maxHealth="9" position="vertex4"
    strength="5" visRange="5" zoneScore="27"/>
    <team lastStepScore="27" money="1" score="4270"
    zonesScore="26">
      <achievements>
        <achievement name="area20"/>
        ...
      </achievements>
    </team>
    <visibleVertices>
      <visibleVertex name="vertex19" team="none"/>
      ...
    </visibleVertices>
    <visibleEdges>
      <visibleEdge node1="vertex0" node2="vertex11"/>
      ...
    </visibleEdges>
    <visibleEntities>
      <visibleEntity name="b5" team="B" node="vertex0"
                     status="normal"/>
      ...
    </visibleEntities>
    <probedVertices>
      <probedVertex name="vertex18" value="4"/>
    </probedVertices>
    <surveyedEdges>
      <surveyedEdge node1="vertex3" node2="vertex7" weight="2"/>
      ...
    </surveyedEdges>
    <inspectedEntities>
```

```
      <inspectedEntity energy="8" health="9" maxEnergy="8"
      maxHealth="9" name="b5" node="vertex10" role="role2"
      strength="6" team="B" visRange="2"/>
      ...
    </inspectedEntities>
  </perception>
</message>
```

Now, it is not necessary to elaborate on the nesting of the tags, which is obvious from the example. We will only focus on the relevant tags.

- `<perception>` has two attributes
  - `deadline` denotes the latest moment in time when the server will accept an action, and
  - `id` represents the action-id, that is the id, that is supposed to be added to the action-message.
- `<simulation>` has a `step`-attribute, that denotes the current step of the simulation.
- `<self>` represents the state of the vehicle, with the attributes
  - `energy`, which is the current energy,
  - `health`, which is the current health,
  - `lastAction`, which is the last action that has been performed,
  - `lastActionResult`, which is the outcome of the last action,
  - `maxEnergy`, which is the maximum energy,
  - `maxEnergyDisabled`, which is the maximum energy, when the vehicle is disabled,
  - `maxHealth`, which is the maximum health,
  - `position`, which is the vehicle's current position,
  - `strength`, which is the strength,
  - `visRange`, which is the visibility range, and
  - `zoneScore`, which is the value of the zone that the vehicle is part of.
- `team` represents the state of the vehicles team, with the attributes
  - `lastStepScore`, which is the score of the team in the last step,
  - `money`, which is the current amount of money the team has,
  - `score`, which is the overall score of the team, and

  - `zonesScore`, which is the sum of the values of all zones occupied by the team.

  Note, at this point, that `lastStepScore` is the addition of `money` and `zonesScore` from the last step. Note also that `score` is the sum of all `lastStepScore`s

- `<achievement>` is an achievement, whose name is indicated by the `name`-attribute.

- `<visibleVertex>` represents a visible vertex, whose name is indicated by the `name`-attribute, which denotes its identifier, and by the `team`-attribute, representing the team occupying the vertex.

- `<visibleEdge>` denotes a visible edge, its vertices are represented by the attributes `node1` and `node2`.

- `<visibleEntity>` represents a visible entity, denoted by the `name`-attribute. The `status` of the agent can be either `normal` or `disabled`.

- `<probedVertex>` is a probed vertex, the `name`-attribute is the vertex's name and the `value` is the vertex's value.

- `<surveyedEdge>` is a surveyed edge, `node1` and `node2` denote the adjacent vertices, and `weight` represents the weight.

- `<inspectedEntity>` represents an inspected vehicle, the attributes are

  - `energy`, which is the current energy of the vehicle,
  - `health`, which is the current health of the vehicle,
  - `maxEnergy`, which is the maximum energy of the vehicle,
  - `maxHealth`, which is the maximum health,
  - `name`, which is the vehicle's name
  - `node`, which is the name of the vertex the vehicle is standing on,
  - `role`, which is the vehicles role,
  - `strength`, which is the vehicle's strength,
  - `team`, which is the vehicle's team, and
  - `visRange`, which is the vehicle's visibility range.

`ACTION` **(agent-2-server)** The agent should respond the `REQUEST-ACTION` message with the action it chooses to perform.

The envelope of the `ACTION` message contains one element `<action>` with the attributes `type` and `id`. The attribute `type` indicates an action the agent wants to perform. It contains a string value which can be one of the following strings:

- `"goto"` with an obligatory attribute `param`, moves the entity to another vertex, whereas the attribute denotes the vertex,
- `"attack"` with an obligatory attribute `param`, attacks another entity, whereas the attribute denotes the entity-to-be-attacked,
- `"parry"` parries any attack,
- `"probe"` probes the current vertex,
- `"survey"` surveys some visible edges,
- `"inspect"` inspects some visible entities,
- `"repair"` with an obligatory attribute `param`, repairs another entity, whereas the attribute denotes the entity-to-be-repaired,
- `"buy"` with an obligatory attribute `param`, buys an item, whereas the attribute denotes the item-to-be-bought,
- `"recharge"` recharges, and
- `"skip"` does nothing.

Note, however, that the scenario description contains the precise semantics of the actions.

Here is an example of a `goto`-action:

```
<?xml version="1.0" encoding="UTF-8"?>
<message type="action">
  <action type="goto" param="vertex1">
</message>
```

Here is an example of a `attack`-action:

```
<?xml version="1.0" encoding="UTF-8"?>
<message type="action">
  <action type="attack" param="a2"/>
</message>
```

Here is an example of a `probe`-action:

```
<?xml version="1.0" encoding="UTF-8"?>
<message type="action">
  <action type="probe"/>
</message>
```

Here is an example of a `survey`-action:

```
<?xml version="1.0" encoding="UTF-8"?>
<message type="action">
  <action type="survey"/>
</message>
```

Here is an example of a `inspect`-action:

```
<?xml version="1.0" encoding="UTF-8"?>
<message type="action">
  <action type="inspect"/>
</message>
```

Here is an example of a `parry`-action:

```
<?xml version="1.0" encoding="UTF-8"?>
<message type="action">
  <action type="parry"/>
</message>
```

Here is an example of a `recharge`-action:

```
<?xml version="1.0" encoding="UTF-8"?>
<message type="action">
  <action type="recharge"/>
</message>
```

Here is an example of a `repair`-action:

```
<?xml version="1.0" encoding="UTF-8"?>
<message type="action">
  <action type="repair" param="b2"/>
</message>
```

Here is an example of a `buy`-action:

```
<?xml version="1.0" encoding="UTF-8"?>
<message type="action">
  <action type="buy" param="battery"/>
</message>
```

The attribute `id` is a string which should contain the `REQUEST-ACTION` message identifier. The agents must plainly copy the value of `id` attribute in `REQUEST-ACTION` message to the `id` attribute of `ACTION` message, otherwise the action message will be discarded.

Note that the corresponding `ACTION` message has to be delivered to the time indicated by the value of attribute `deadline` of the `REQUEST-ACTION` message. Agents should therefore send the `ACTION` message in advance before the indicated deadline is reached so that the server will receive it in time.

Example:

```
<?xml version="1.0" encoding="UTF-8"?>
<message type="action">
  <action id="70" type="skip"/>
</message>
```

4 Simulation Server

This chapter describes how to use the scenario simulator provided by the contest organization (available at `http://www.multiagentcontest.org/2011`). Although some pre-defined configurations for the scenario are provided, participants can create new scenarios to evaluate specific strategies for their teams. All these aspects are covered in the next sections.

4.1 Starting *MASSim*

You can start the *MASSim* server by invoking this:

```
$ ./startServer.sh
```

You will then be prompted to choose a simulation.

In parallel you can also start the monitor which will allow you to observe the current simulation. The monitor can be invoked like this:

```
$ ./startMarsMonitor.sh
```

Note, however, this monitor provides you with complete information. The agents taking part on the simulation, on the other hand, only have access to a subset of this information.

The monitor also stores the match on hard disk. You can view these files by invoking:

```
$ ./startMarsFileViewer.sh /path/where/the/files/are
```

For Microsoft Windows we suggest that you install Cygwin[2] in order to run the *MASSim*-software taking advantage of the shell scripts.

[2]http://www.cygwin.com/

```
<?xml version="1.0" encoding="UTF-8"?>
<conf   backuppath="backup"
        launch-sync-type="key"
        reportpath="./backup/"
        time-to-launch="10000"
        tournamentmode="0"
        tournamentname="Mars2011">
    <simulation-server>
        <network-agent backlog="10" port="12300"/>
    </simulation-server>
    <match>
        <simulation ...>
            ...
        </simulation>
        ...
        <simulation ...>
            ...
        </simulation>
    </match>
    ...
    <match>
        ...
    </match>
    <accounts>
        ...
    </accounts>
</conf>
```

Figure 7: General structure of the *MASSim* configuration-file.

4.2 Configuring *MASSim*

When starting *MASSim*, you must provide a configuration file to the server. Configuration files are XML-based, and a set of configuration files is already available in the `scripts/conf` sub-folder of your *MASSim* installation. A detailed explanation of the configuration file is given next.

4.2.1 General Configuration

The general structure of the configuration file is depicted in Fig. 7.
The attributes of the `conf` tag are the following:

- `backuppath` - The path where important information of each simulation step is stored.

- `launch-sync-type` - Determines whether the server is started by pressing ENTER or after a certain time defined in `time-to-launch`. The value can be `key` or `timer`.
- `reportpath` - The path where the overall tournament results are stored.
- `time-to-launch` - The time for the option `timer`.
- `tournamentmode` - Defines the structure of the tournament. `0` sets it to a round robin tournament. `1` is used when only one team should play against all others.
- `tournamentname` - Sets the tournament name to this value.

The `simulation-server` tag has one child with two attributes. `backlog` defines the time intervals (in milliseconds) for printing the debug messages to `stdout` or `stderr` respectively. The attribute `port` sets the port of the server.

A configuration can have one or more `match` tags, that will be instantiated depending on the `tournamentmode` attribute.

4.2.2 Simulation Configuration

The simulation tag is used to specify the scenario to be run, along with all the parameters that affect the simulation.

The attributes available for the `simulation` tag are the following:

- `id` - An identifier for the simulation.[3]
- `simulationclass` - The name of the main Java class implementing the scenario. For the 2011 Mars Scenario, the class that must be used here is `massim.competition2011.GraphSimulation`.
- `configurationclass` - The name of the Java class that will hold the configuration data specified in the `configuration` child tag. For 2011 Mars Scenario, the class to use is `massim.competition2011.GraphSimulationConfiguration`.
- `file-simulationlog` - The path where simulation logs are stored.
- `rmixmlobsserverhost` - The host to use to connect the scenario monitor.
- `rmixmlobsserverport` - The port to use to connect the scenario monitor.

[3]To distinguish among different instances of the simulation executed during a tournament, this identifier will be appended to the names of the teams taking part in that instance.

- rmixmlobserver - The name of the Java class that will translate the current scenario state into XML data, and send it via RMI to the scenario monitor when connected. For the 2011 Mars Scenario, the class to use is
massim.competition2011.
GraphSimulationRMIXMLDocumentObserver.

A skeleton XML for the simulation tag is shown in Fig. 8. It has two children: configuration and agents. The configuration part is scenario-specific, and must be in correspondence with the configurationclass specified in the simulation attributes. For the 2011 Mars scenario, the attributes of the configuration tag are the following:

- maxNumberOfSteps - The number of steps that the simulation must run until determining a winner.

- numberOfAgents - The total number of agents that take part in the simulation run.

- numberOfTeams - The number of teams that take part in the simulation run.

- agentsPerTeam - The number of agents composing each team in the simulation run.

- numberOfNodes - The size of the randomly generated map, in terms of number of nodes (vertices).

- gridWidth, gridHeight - Affect the map-generation algorithm. Internally, nodes are created as being situated on a grid, and then edges are calculated according to this grid. gridWidth $*$ gridHeight must be greater than numberOfNodes.

- cellWidth - This parameter is not used from *MASSim* itself, but is given to the monitor to facilitate the visualization. It stands for the distance (measured in pts) between two adjacent points in the grid.

- minNodeWeight, maxNodeWeight - The minimum and maximum possible value for the weights of the nodes (randomly assigned).

- minEdgeCost, maxEdgeCost - The minimum and maximum possible value for the costs of the edges (randomly assigned).

```
<simulation ...>
    <configuration ...>
        <actions>
            <action .../>
            <action .../>
            ...
        </actions>
        <roles>
            <role ...>
                <actions>
                    <action .../>
                    <action .../>
                    ...
                </actions>
                <actionsDisable>
                    <action .../>
                    <action .../>
                    ...
                </actionsDisable>
            </role>
        </roles>
        <achievements>
            <achievement .../>
        </achievements>
    </configuration>
    <agents>
        <agent ...>
            <configuration .../>
        </agent>
        <agent ...>
            <configuration .../>
        </agent>
        ...
    </agents>
</simulation>
```

Figure 8: Simulation XML structure

Actions The actions section is used to specify the costs that actions may imply for the agents attempting to execute them. There must be one action tag for each action. Thus, the name attribute must be one of the following: recharge, goto, attack, parry, probe, survey, inspect, repair or buy.

In the general case, the rest of the attributes to be specified here represent the costs of attempting to execute that action in different situations. An action can cost energy, health, and achievement points (money). The costs can vary depending on the success or failure of the action, and also on whether the agent is in a normal or disabled[4] state, so attributes for all the combinations can be specified.[5] The names of these attributes are:

- energyCost
- healthCost
- pointsCost
- energyCostFailed
- ...
- energyCostDisabled
- ...
- energyCostFailedDisabled

Two special cases are the actions recharge and goto. For the recharge action, the values represent the percentage of the maximum energy and health that gets recovered. "Failure" in this particular case means that the agent has been attacked, and thus the health and energy recovering rates can be specified to be different.

The energy cost of a successful goto action is actually determined by the cost of the traversed edge. Therefore, in this particular case the energyCost and energyCostDisabled specified here are considered as factors, that are multiplied by the edge cost. The cost for the Failed cases, on the other hand, are constants, as with the rest of the actions.

A thing to note here is that some costs can be specified to be negative values, e.g. if an agent should recover some energy when it was not able to perform a particular action.

[4]An agent is considered to be in disabled state when its current health is 0

[5]Not all combinations make sense, and some of them may be just ignored by the server. Nevertheless, they are provided for notation consistency

Roles The `roles` section defines the different roles that agents participating in the simulation will assume. A `role` encompasses all the internal characteristics of the agent and the set of actions that the agent is allowed to perform, both when in normal state and when disabled. The following attributes should be specified for each role:

- `name` - The name by which this role is referenced.
- `maxEnergy` - The initial upper limit for the `energy` of the agent.
- `maxBuyEnergy` - The upper limit for `maxEnergy` that can be reached when attempting to perform the `buy` action with `param="battery"`.
- `rateBuyEnergy` - The amount by which `maxEnergy` is increased when successfully performing the `buy` action with `param="battery"`.
- `maxEnergyDisabled` - The initial upper limit for the `energy` of the agent when `disabled`.
- `rateBuyEnergyDisabled` - The amount by which `maxEnergyDisabled` is increased when successfully performing the `buy` action with `param="battery"`.
- `maxHealth` - The initial upper limit for the `health` of the agent.
- `maxBuyHealth` - The upper limit for `maxHealth` that can be reached when attempting to perform the `buy` action with `param="shield"`.
- `rateBuyHealth` - The amount by which `maxHealth` is increased when successfully performing the `buy` action with `param="shield"`.
- `strength` - The initial strength of the agent.
- `maxBuyStrength` - The upper limit for `strength` that can be reached when attempting to perform the `buy` action with `param="sabotageDevice"`.
- `rateBuyStrength` - The amount by which `strength` is increased when successfully performing the `buy` action with `param="sabotageDevice"`.
- `visRange` - The initial visibility range of the agent.
- `maxBuyVisRange` - The upper limit for `visRange` that can be reached when attempting to perform the `buy` action with `param="sensor"`.
- `rateBuyVisRange` - The amount by which `visRange` is increased when successfully performing the `buy` action (with `param="sensor"`).

The `actions` and `actionsDisable` sections of the role definition expect a list of `action` tags with only one attribute: the `name` of an action. The actions listed in these sections are the only actions that will be enabled for agents having this role when in `normal` or `disabled` state respectively.

Achievements The achievements that can yield `achievement points` for the teams are defined here. Each achievement has four attributes: a (preferably unique) `name`, a `class` stating the type of achievement, a `quantity` needed to reach the achievement, and the amount of `achievement points` that the achievement yields. Six different classes of achievements are implemented:

- `probedVertices` - The `quantity` means the number of different nodes that a team needs to probe.
- `surveyedEdges` - The `quantity` means the number of different edges that a team needs to survey.
- `inspectedAgents` - The `quantity` means the number of different opponent agents that a team needs to inspect.
- `successfulAttacks` - The `quantity` means the number of successful attacks that a team needs to perform.
- `successfulParries` - The `quantity` means the number of successful parries that a team needs to perform (only counted when the parrying agent is actually attacked by an opponent).
- `areaValue` - The `quantity` means the score of a zone that a team needs to build.

Agents The `agents` section of the simulation configuration is where it is defined how server-side teams are to be composed during the simulation. Agents defined here will be matched with agents defined in the `accounts` section to be controlled externally by the participants. This matching of agents varies in function of the `tournamentmode` parameter explained in 4.2.1.

The attributes for the `agent` tag are:

- `team` - The server-side name of the team.
- `agentclass` - The name of main Java class implementing the agents. For the 2011 Mars scenario, the class to use is `massim.competition2011.GraphSimulationAgent`.

- `agentcreationclass` - The name of the Java class that will hold the configuration parsed from the `configuration` child tag. For the 2011 Mars scenario, the class to use is `massim.competition2011.GraphSimulationAgentParameter`.

The `configuration` child tag for the 2011 Mars scenario only has one attribute: `roleName`, which refers to the name of one of the previously defined roles.

4.2.3 Accounts Configuration

In the `accounts` section of the configuration file, one can configure the developers' team that will participate in the tournament, and with which credentials each developer-side agent will connect to *MASSim* to control its server-side counterpart.

The `actionclassmap` has one attribute `name` and defines all available action classes for the agent accounts. Each `actionclass` has a `class` attribute and an `id`. An `account` is structured as follows:

- `actionclassmap` - Refers to the actionclassmap name that is used for this account.
- `auxtimeout` - Additional timeout for messages. The purpose of this parameter is to give the agents some additional time to allow the server to process the message.
- `defaultactionclass` - Sets the default action class.
- `maxpacketlength` - Defines the maximal length of on message.
- `password` - The password for the agent.
- `team` - The team name for the agent.
- `timeout` - The timeout for messages.
- `username` - The user name of the agent.

5 Agent Development

Although participants can develop their own interface with the server, implementing the client-side of the protocol described in chapter 3, a high level interface is provided by the organisation of the contest. It hides all the details of the protocol and gives to the user a suitable API. This interface is based on the EISMASSim which allows agents coded in several languages (2APL, GOAL, Jason, Java, ...) to straightforwardly integrate their agents with the simulation server.

EISMASSim is based on EIS[6], which is a proposed standard for agent-environment interaction. It maps the communication between the *MASSim*-server and agents (sending and receiving XML-messages) to Java-method-calls and call-backs. On top of that it automatically establishes and maintains connections to a specified *MASSim*-server. Additionally it is intended to also gather statistics about the execution of your agents. EISMASSim and EIS both come as a jar-files which are included in the software-package.

5.1 Using EISMASSim

In order to use EISMASSim with your project, you have to perform a couple of steps, which we will outline here.

1. Setting up the class-path: The first thing you have to do is to add EIS and EISMASSim to the class-path of your project. Please use the jar-files `eis-0.3.jar` and `eismassim-1.0.jar`. The first jar contains the *generic* environment-interface, the second one contains the *specialized* one.

2. Creating an instance of the environment interface: It is not intended to instantiate EIS-compliant environment-interfaces directly, that is calling the constructor of the respective class. Instead it is advised to use the *class-loader* `eis.EILoader`. Here is an example for instantiating the environment-interface-class via this very class-loader[7]:

```
EnvironmentInterfaceStandard ei = null;
try {
    String cn = "massim.eismassim.EnvironmentInterface";
    ei = EILoader.fromClassName(cn);
} catch (IOException e) {
    // TODO handle the exception
}
```

3. Registering your agents: Now that the environment-interface is instantiated you need to register your agents to it. That is, that you are required to register every single agent that is supposed to interact with the environment via the environment-interface using its name or any unique identifier. For each of your agents please do this:

[6] Available at `http://sf.net/projects/apleis/`.

[7] There is also a method called `fromJarFile`, which firstly add a jar-file to the class-path, secondly looks up the main-class attribute from the jar's manifest-entry, and thirdly instantiates the environment-interface. This works for EISMASSim as well.

```
try {
  ei.registerAgent(agentName);
} catch (AgentException e1) {
  // TODO handle the exception
}
```

4. Associating your agents with the vehicles: At this moment you have to associate your agents with the available entities. An entity is a connection to a vehicle, which is part of a simulation executed by the *MASSim*-server. You can associate one of your agents with an entity (vehicle) by using the entity's name. The names of the entities however are specified in the configuration XML-file (see below). As soon as you associate an agent with an entity, a connection to the *MASSim*-server is established. Here is an example how to associate an agent with an entity:

```
try {
    ei.associateEntity(agentName,entityName);
} catch (RelationException e) {
    // TODO handle the exception
}
```

5. Starting the execution: The next step is to start the overall execution. This is how it is done:

```
try {
    ei.start();
} catch (ManagementException e) {
    // TODO handle the exception
}
```

6. Perceiving the environment: Perceiving is facilitated either by 1. getting all percepts, that is calling the `getAllPercepts`-method or 2. by handling percepts-as-notifications, that is every time there is a new percept a listener's method is called in order to trigger a reaction to the percept. Note that this is EIS's usual policy about perceiving. Here is an example for retrieving all percepts[8]:

```
try {
    Collection<Percept> ret = getAllPercepts(getName());
    // TODO interpret the percepts
} catch (PerceiveException e)  {
    // TODO handle the exception
} catch (NoEnvironmentException e) {
    // TODO handle the exception
}
```

7. Acting: Executing an action means invoking the `performAction`-method and passing 1. the name of the agent, that intends to execute an action, and 2. an action-object that represents the action-to-be-executed. This is an exemplary execution of an action:

```
Action = new Action(...);
try {
    ei.performAction(agentName, action);
} catch (ActException e) {
    // handle the exception
}
```

5.2 Configuring EISMASSim

The EISMASSim environment-interface can be configured using the configuration-file `eismassimconfig.xml` which is automatically loaded and evaluated when the environment-interface is instantiated. Fig. 9 shows an exemplary configuration-file for EISMASSim.

The attributes of the `<interfaceConfig>`-tag are:

[8]For an introduction on how to use percepts-as-notifications, see the manual that accompanies the EIS software package.

```
<?xml version="1.0" encoding="UTF-8"?>
<interfaceConfig scenario="mars2011" host="localhost" port="12300"
scheduling="yes" times="no" notifications="no">
  <entities>
    <entity name="vehicle1" username="a1" password="1" xml="yes"/>
    <entity name="vehicle2" username="a2" password="1" xml="yes"/>
    <entity name="vehicle3" username="a3" password="1" xml="yes"/>
    <entity name="vehicle4" username="a4" password="1" xml="yes"/>
    <entity name="vehicle5" username="a5" password="1" xml="yes"/>
    <entity name="vehicle6" username="a6" password="1" xml="yes"/>
    <entity name="vehicle7" username="a7" password="1" xml="yes"/>
    <entity name="vehicle8" username="a8" password="1" xml="yes"/>
    <entity name="vehicle9" username="a9" password="1" xml="yes"/>
  </entities>
</interfaceConfig>
```

Figure 9: An exemplary EISMASSim-configuration-file.

- `scenario` specifies the Contest-scenario that is supposed to be handled. For the time being the only value that is accepted is `"mars2011"`.
- `host` specifies the URL of the *MASSim*-server that runs the simulations. This can be for example `localhost`, a valid IP-address, or the fully-qualified hostname of one of our Contest-servers.
- `port` specifies the port-number of the *MASSim*-server.
- `scheduling` enables/disables scheduling. Enabled scheduling means that an action-message is not sent unless there is a valid action-id (see the protocol-description for details on the action-ids). This mechanism makes sure that a single action-id is used only once. Note that an attempt to send an action-message times out after 5 seconds. The default value is `yes` for scheduling enabled. **Warning:** note, however, that disabling scheduling in the interface leaves you with the responsibility of scheduling, that is to ensure that the server is not strained with more than one action per connection and simulation-step.
- `times` enables/disables time-annotations. If enabled this will annotate each percept with a time-stamp, that indicates when the percept has been generated by the server (see the protocol-description for details on time-stamps).
- `notifications` denotes whether percepts are to be provided as notifications. The default-value is `no`.

Each `<entity>`-tag specifies a single connection to the *MASSim*-server. The attributes are:

- `name` specifies the name of the connection. This is a requirement for acting and perceiving, and needs to be unique.

- `username` and `password` specify the credentials that are required by *MASSim*'s authentication-mechanism (provided either by the organizers, or specified in your very own server-configuration-file).
- `xml` enables/disables printing incoming/outgoing XML-messages to the console. This is useful for debugging-purposes.

5.3 Actions and Percepts for the Mars-Scenario

In the following, we will elaborate on actions and percepts. Each action and each percept consists of a name followed by an optional list of parameters. A parameter is either an identifier (`<Identifier>`), that is a String, or a numeral(`<Numeral>`).

Here is the list of actions that can be performed in the course of each simulation (see the scenario-description for the precise semantics of the actions):

- `attack(<Identifier>)` attacks a vehicle.
- `buy(<Identifier>)` buys an item.
- `goto(<Identifier>)` moves to a vertex.
- `inspect` inspects some visible vehicles.
- `parry` parries all attacks.
- `probe` probes the current vertex.
- `recharge` recharges the vehicle.
- `repair(<Identifier>)` repairs a vehicle.
- `skip` does nothing.
- `survey` surveys some visible edges.

Creating an action-object that is to be passed as a parameter to the method `performAction` is very straightforward:

```
Action attack = new Action("attack", new Identifier("a2"));
```

In the following we will consider a list of percepts that can be available during a tournament. Note that during a simulation, data from the respective `sim-start`-message will be available as well as data from the current `request-action`-message (see the protocol description for details about the messages):

- `achievement(<Identifier>)` denotes an achievement.

- `bye` indicates that the tournament is over.
- `deadline(<Numeral>)` indicates the deadline for sending a valid action-message to the server in Unix-time.
- `edges(<Numeral>)` represents the number of edges of the current simulation.
- `energy(<Numeral>)` denotes the current amount of energy of the vehicle.
- `health(<Numeral>)` indicates the current health of the vehicle.
- `id(<Identifier>)` indicates the identifier of the current simulation.
- `lastAction(<Identifier>)` indicates the last action that was sent to the server.
- `lastActionResult(<Identifier>)` indicates the outcome of the last action.
- `lastStepScore(<Numeral>)` indicates the score of the vehicle's team in the last step of the current simulation.
- `maxEnergy(<Numeral>)` denotes the maximum amount of energy the vehicle can have.
- `maxEnergyDisabled(<Numeral>)` denotes the maximum amount of energy the vehicle can have, when it is disabled.
- `maxHealth(<Numeral>)` represents the maximum health the vehicle can have.
- `money(<Numeral>)` denotes the amount of money available to the vehicle's team.
- `position(<Identifier>)` indicates the current position of the vehicle. The identifier is the vertex's name.
- `probedVertex(<Identifier>,<Numeral>)` denotes the value of a probed vertex. The identifier is the vertex's name and the numeral is its value.
- `ranking(<Numeral>)` indicates the outcome of the simulation for the vehicle's team, that is its ranking.
- `requestAction` indicates that the server has requested the vehicle to perform an action.
- `score(<Numeral>)` represents is the overall score of the vehicle's team.

- `simEnd` indicates that the server has notified the vehicle about the end of a simulation.
- `simStart` indicates that the server has notified the vehicle about the start of a simulation.
- `step(<Numeral>)` represents the current step of the current simulation.
- `steps(<Numeral>)` represents the overall number of steps of the current simulation.
- `strength(<Numeral>)` represents the current strength of the vehicle.
- `surveyedEdge(<Identifier>,<Identifier>,<Numeral>)` indicates the weight of a surveyed edge. The identifiers represent the adjacent vertices and the numeral denotes the weight of the edge.
- `timestamp(<Numeral>)` represents the moment in time, when the last message was sent by the server, again in Unix-time.
- `vertices(<Numeral>)` represents the number of vertices of the current simulation.
- `visRange(<Numeral>)` denotes the current visibility-range of the vehicle.
- `visibleEdge(<Identifier>,<Identifier>)` represents a visible edge, denoted by its two adjacent vertices.
- `visibleEntity(<Identifier>,<Identifier>,<Identifier>, <Identifier>)` denotes a visible vehicle. The first identifier represents the vehicle's name, the second one the vertex it is standing on, the third its team and the fourth and final one indicates whether the entity is disabled or not.
- `visibleVertex(<Identifier>,<Identifier>)` denotes a visible vertex, represented by its name and the team that occupies it.
- `zoneScore(<Numeral>)` indicates the current score yielded by the zone the vehicle is part of.
- `zonesScore(<Numeral>)` indicates the current score of the vehicle's team yielded by zones, that is the sum of scores of all zones.

Note, however, that the percepts look a little different, when annotations (see the section on configuring EISMASSim) are activated.

5.4 Programming Java Agents

Again, we provide a set of very simple agents that function as a proof-of-concept and as dummy-agents for testing. On top of that we also show how to use the environment-interface EISMASSim (see the EISMASSim description for further details).

This document will show you two things: 1. how configure and execute our dummy agents, and 2. how to create new agents. Note, however, that we do not encourage you to create your own agent team from scratch using the agent-infrastructure we are providing. Our agent-infrastructure is rather limited and we are not optimistic that we will have enough time to add new features, if the need for such features arises. Nevertheless, if you accept this, feel free to use the agent-infrastructure.

5.4.1 Running Our Dummy-Agents

In the software package we have included a single agent-configuration (see below). It sets up two teams `A` and `B`. Each team has 10 agents.

In order to run the dummy agents, navigate to the `javaagents/scripts` directory and execute

```
./startAgents.sh
```

You will then be asked to select a configuration. At the time of the release you can only select a single configuration. If you add new ones you can select them as well. After selecting a configuration, the environment-interface will immediately establish the connections to the *MASSim*-server, which must be already up and running, as specified in the environment-interface configuration-file, and execute the agents.

5.4.2 Changing the Configuration

If you desire another configuration that is different from ours, please perform these steps:

1. create a new subfolder at `javaagents/scripts/conf`, you can define an arbitrary name, which will be made selectable by the `startAgents.sh` shell-script,
2. copy the two XML-files from `javaagents/scripts/conf/dummyteam/` to the directory you have just created,
3. if necessary adapt the `eismassimconfig.xml`-file (see the EISMASSim description for instructions), and

```
<?xml version="1.0" encoding="UTF-8"?>
<javaAgentsConfig>
  <agents>
    <agent name="A1" entity="cA1" class="massim.javaagents.agents2011.SimpletonBuyerAgent" team="A"/>
    <agent name="A2" entity="cA2" class="massim.javaagents.agents2011.RechargingFighterAgent" team="A"/>
    <agent name="A3" entity="cA3" class="massim.javaagents.agents2011.RechargingFighterAgent" team="A"/>
    <agent name="A4" entity="cA4" class="massim.javaagents.agents2011.RechargingFullExplorerAgent" team="A"/>
    <agent name="A5" entity="cA5" class="massim.javaagents.agents2011.RechargingFullExplorerAgent" team="A"/>
    <agent name="A6" entity="cA6" class="massim.javaagents.agents2011.RechargingFullExplorerAgent" team="A"/>
    <agent name="A7" entity="cA7" class="massim.javaagents.agents2011.RechargingFullExplorerAgent" team="A"/>
    <agent name="A8" entity="cA8" class="massim.javaagents.agents2011.RechargingFullExplorerAgent" team="A"/>
    <agent name="A9" entity="cA9" class="massim.javaagents.agents2011.RechargingFullExplorerAgent" team="A"/>
    <agent name="B1" entity="cB1" class="massim.javaagents.agents2011.SimpletonBuyerAgent" team="B"/>
    <agent name="B2" entity="cB2" class="massim.javaagents.agents2011.RechargingFighterAgent" team="B"/>
    <agent name="B3" entity="cB3" class="massim.javaagents.agents2011.RechargingFighterAgent" team="B"/>
    <agent name="B4" entity="cB4" class="massim.javaagents.agents2011.RechargingFullExplorerAgent" team="B"/>
    <agent name="B5" entity="cB5" class="massim.javaagents.agents2011.RechargingFullExplorerAgent" team="B"/>
    <agent name="B6" entity="cB6" class="massim.javaagents.agents2011.RechargingFullExplorerAgent" team="B"/>
    <agent name="B7" entity="cB7" class="massim.javaagents.agents2011.RechargingFullExplorerAgent" team="B"/>
    <agent name="B8" entity="cB8" class="massim.javaagents.agents2011.RechargingFullExplorerAgent" team="B"/>
    <agent name="B9" entity="cB9" class="massim.javaagents.agents2011.RechargingFullExplorerAgent" team="B"/>
  </agents>
</javaAgentsConfig>
```

Figure 10: An exemplary agents configuration-file.

4. adapt the `javaagentsconfig.xml`-file, according to the following instructions.

Figure 10 shows an exemplary agents configuration-file. Here, the `<agent>` tag is the most relevant one. Its attributes are:

- `name` is the agent's name, which is required for communication and registering to the environment-interface,
- `entity` is the name of the connector/vehicle the agent is supposed to control, as specified in the configuration file of the environment-interface,
- `class` is the class-name of the agent, which will be dynamically loaded via Java-reflection, and
- `team` is the team-name of the agent, which is required for communication[9].

The `class`-tag is the most relevant one, because this is the place to plug in different agents. There are a couple of unmentioned agents in the package `massim.javaagents.agents2011` (see the accompanying javadoc for their descriptions and the code of their implementations). On top of that you can also specify your own agents here.

5.4.3 Creating Your Own Agents

In order to create and use your own agents you are required to perform these steps:

[9]An agent can only receive messages from and send messages to its team-members.

```
public class YourAgent extends Agent {

        public YourAgent(String name, String team) {
                super(name, team);
                // TODO do something if necessary
        }

        @Override
        public void handlePercept(Percept p) {
                // TODO handle percepts if necessary
        }

        @Override
        public Action step() {
                // TODO deliberate and return an action
                return null;
        }

}
```

Figure 11: A new agent without any functionality.

1. create a new agent-class that inherits from `massim.javaagents.Agent`,
2. implement a constructor and a couple of required methods,
3. incorporate your new agent-class into the `javaagentsconfig.xml`,
4. make sure that your new agent-class is in the class-path, and
5. execute.

Figure 11 shows a very rudimentary agent without any functionality. It is not necessary to extend the constructor, unless you have to add some useful things. You have to add code to the `handlePercept`-method if you intend to handle percepts-as-notifications. Please note two things: 1. handling percepts-as-notifications is optional, that is there are other means to retrieve percepts, and 2. you have to explicitly activate percepts-as-notifications in the environment-interface configuration file (see the EISMASSim description) if you intend to use them. The `step`-method is automatically called by the interpreter that executes all agents. It is supposed to return an action, which will then be executed automatically. Note that if this method returns `null`, no message is sent to the server. The `step`-method is the place where you are supposed to add your agent's intelligence.

Finally, we will introduce a couple of methods that might be useful (see the javadoc of `massim.javaagents.Agent` for the full overview):

- `Collection<LogicBelief> getBeliefBase()` yields the current belief-base (immutable),

- `Collection<LogicGoal> getGoalBase()` yields the current goal-base (immutable),
- `Collection<Percept> getAllPercepts()` yields all percepts that are currently available (the EISMASSim description contains an overview),
- `Collection<Message> getMessages()` yields all messages that have been sent to the agent,
- `void sendMessage(LogicBelief belief, String receiver)` sends a message to a recipient,
- `LinkedList<LogicBelief> getAllBeliefs(String predicate)` returns all beliefs that have a given predicate,
- `void removeBeliefs(String predicate)` removes all beliefs that have a given predicate,
- `void removeGoals(String predicate)` removes all goals that have a given predicate,
- `void addBelief(LogicBelief belief)` adds a belief to the belief-base,
- `void addGoal(LogicGoal goal)` adds a goal to the goal-base,
- `boolean containsBelief(LogicBelief belief)` returns true if the belief-base contains a given belief,
- `boolean containsGoal(LogicGoal goal)` returns true if the goal-base contains a given goal,
- `void clearBeliefs()` empties the belief-base, and
- `void clearGoals()` empties the goal-base.

6 Conclusion

In this document we have provided the complete technical documentation of the Multi-Agent Programming Contest 2011. The outcomes of the contest will be analyzed in depth in an upcoming Technical Report.

References

[Behrens et al., 2010] Behrens, T., Dastani, M., Dix, J., Köster, M., and Novak, P., editors (2010). *Special Issue about Multi-Agent-Contest*, volume ?? of *Annals of Mathematics and Artificial Intelligence*. Springer, Netherlands.

[Behrens et al., 2009] Behrens, T. M., Dastani, M., Dix, J., and Novák, P. (2009). Agent contest competition: 4th edition. In Hindriks, K. V., Pokahr, A., and na, S. S., editors, *Programming Multi-Agent Systems, 6th International Workshop (ProMAS 2008)*, volume 5442 of *Lecture Notes in Computer Science*, pages 211–222. Springer.

[Dastani et al., 2005] Dastani, M., Dix, J., and Novák, P. (2005). The first contest on multi-agent systems based on computational logic. In Toni, F. and Torroni, P., editors, *Computational Logic in Multi-Agent Systems, 6th International Workshop, CLIMA VI*, volume 3900 of *Lecture Notes in Computer Science*, pages 373–384. Springer.

[Dastani et al., 2006a] Dastani, M., Dix, J., and Novák, P. (2006a). The first contest on multi-agent systems based on computational logic. In Toni, F. and Torroni, P., editors, *Computational Logic in Multi-Agent Systems (CLIMA VI)*, volume 3900 of *Lecture Notes in Artificial Intelligence*, pages 373–384. Springer. 6th International Workshop.

[Dastani et al., 2006b] Dastani, M., Dix, J., and Novák, P. (2006b). The second contest on multi-agent systems based on computational logic. In Inoue, K., Satoh, K., and Toni, F., editors, *Computational Logic in Multi-Agent Systems, 7th International Workshop, CLIMA VII*, volume 4371 of *Lecture Notes on Computer Science*, pages 266–283. Springer.

[Dastani et al., 2007] Dastani, M., Dix, J., and Novák, P. (2007). The second contest on multi-agent systems based on computational logic. In Inoue, K., Satoh, K., and Toni, F., editors, *Proceedings of CLIMA '06, Revised Selected and Invited Papers*, Lecture Notes in Artificial Intelligence, pages 266–283. Springer.

[Dastani et al., 2008a] Dastani, M., Dix, J., and Novák, P. (2008a). Agent contest competition - 3rd edition. In Dastani, M., Ricci, A., El Fallah Seghrouchni, A., and Winikoff, M., editors, *Proceedings of ProMAS '07, Revised Selected and Invited Papers*, Lecture Notes in Artificial Intelligence. Springer.

[Dastani et al., 2008b] Dastani, M., Dix, J., and Novák, P. (2008b). Agent contest competition - 4th edition. In *Proceedings of Sixth international Workshop on Programming Multi-Agent Systems, ProMAS'08*, volume 5442 of *LNAI*. Springer Verlag.