Multi-Agent Programming Contest
MASSim Server Manual
(2016 Edition)

http://www.multiagentcontest.org/

Tobias Ahlbrecht Jürgen Dix Federico Schlesinger

June 21, 2016

New in 2016: Differences between the last year and 2016 are now marked with boxes.

Contents

1 Starting the Programs

For Microsoft Windows we suggest that you install MSYS[1] or Cygwin[2] in order to run the *MASSim*-software taking advantage of the shell scripts.

1.1 *MASSim* Server

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

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

You will then be prompted to choose a simulation. We mention the files explicitly. The server generates XML files, statistics etc. Please have a look at the folders (`output` and `backup` that were generated during a run.

1.2 *MASSim* Monitor

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

```
$ ./startMapMonitor.sh
```

The monitor has currently two different options for the visualization. Please have a look at the scenario description for details. Also note, this monitor provides you with complete information. Your agents on the other hand do not have access to complete information.

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

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

1.3 *MASSim* Web Server

For the tournament we always provide a web server that is running on an Apache using *Apache Tomcat* and RMI as well as XML and XSLT. This server is not needed for the development of your multi-agent system, however, for the sake of completeness we provide some information how to install it. A install script that explains the procedure is placed in `scripts/tools/`. If you have any questions please contact the organizers.

[1] http://www.mingw.org/wiki/MSYS
[2] http://www.cygwin.com/

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.

2.1 General Configuration

The general structure of the configuration file is depicted in Fig. 1. Note, however, that we use some additional XML features that allows us to use more than one file and reuse parts of the XML in different parts of the configuration. Therefore, you have to look into the main file as well as into the corresponding `config.dtd` file. You can set a starting time with `time` now. Also, a `debug-level` was introduced. Additionally, we describe some parameters in more detail.

```
<?xml version="1.0" encoding="UTF-8"?>
<conf   backuppath="backup"
        launch-sync-type="key"
        reportpath="./backup/"
        time="14:05"
        time-to-launch="10000"
        tournamentmode="0"
        tournamentname="City2016 "
        debug-level="normal" >
    <simulation-server>
        <network-agent backlog="10" port="12300"/>
    </simulation-server>
    <match>
        <simulation ...> ... </simulation>
        ...
        <simulation ...> ... </simulation>
    </match>
    ...
    <match>
        ...
    </match>
    <accounts>
        ...
    </accounts>
</conf>
```

Figure 1: General structure of the *MASSim* configuration-file.

Tag: `conf`. 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` or at time point (defined by `time`). The value can be `key`, `timer` or `time`.
- `reportpath` - The path where the overall tournament results are stored.
- `time` - The time point for the option `time`.
- `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. Finally, `2` allows one to set up all matches manually. You have to add some code similar to the one in Fig. 2 after `</match>` to make it work.

```
<manual-mode>
  <match team1="A" team2="B"/>
  <match team1="A" team2="C"/>
  <match team1="B" team2="D"/>
</manual-mode>
```

Figure 2: Manual mode.

- `tournamentname` - Sets the tournament name to this value.
- `debug-level` - Changes the verbosity of the output on shell. Allowed values: `debug`, `normal`, `critical`, `error`.

Tag: `simulation-server`. The `simulation-server` tag has one child which has 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.

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

Tag: `accounts`. Finally, the `accounts` tag contains the details about the agents that are allowed to take part in the matches.

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.

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

- `id` - An identifier for the simulation. 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.
- `simulationclass` - The name of the main Java class implementing the scenario. For the 2016 Mars Scenario, the class that must be used here is `massim.competition2016.MapSimulation`.
- `configurationclass` - The name of the Java class that will hold the configuration data specified in the `configuration` child tag. For the 2016 Logistics Scenario, the class to use is `massim.competition2016.configuration.MapSimulationConfiguration`.
- `rmixmlobsserverhost` - The host to which the scenario monitor should connect.
- `rmixmlobsserverport` - The port to which the scenario monitor should connect.
- `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 2016 Logistics Scenario, the class to use is `massim.competition2016.MapSimulationRMIXMLDocumentObserver`.
- `xmlstatisticsobserver` - This is needed for the *Apache Tomcat* `Connection Status` and `Results` page.
- `rmixmlobserverweb` - This is needed for the *Apache Tomcat* `Current Simulation` and `Results` page.
- `visualisationobserver` - This defines the class for the visualization.
- `visualisationobserver-outputpath` - This defines the output path for the visualization files.
- `xmlobserver` - This allows one to store the results of a simulation as xml file.
- `xmlobserverpath` - This is the path for the `xmlobserver`.

A skeleton XML for the `simulation` tag is shown in Fig. 3. 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 2016 Logistics scenario, the `configuration` attributes are the following:

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

```
<simulation ...>
    <configuration ...>
        <roles>
            <role ...>
                <roads>
                    <road .../>
                    <road .../>
                    ...
                </roads>
                <tools>
                    <tool .../>
                    <tool .../>
                    ...
                </tools>
            </role>
        </roles>
        <facilities>
           <facility ...>
               <location .../>
               ...
           </facility>
        </facilities>
        <jobs>
           <job ...>
               <products>
                  <product ...>
                  ...
               </products>
           </job>
        </jobs>
        <products>
           <product ...>
               <requirements>
                  <product ...>
                  ...
               </requirements>
           </product>
        </products>
        <generate>
        ...
        </generate>
    </configuration>
    <agents>
        <agent ...>
            <configuration .../>
        </agent>
        <agent ...>
            <configuration .../>
        </agent>
        ...
    </agents>
</simulation>
```


Figure 3: Simulation XML structure

- `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.
- `minLon`, `minLat`, `maxLon`, `maxLat` - Max and min latitude and longitude of the map to use.
- `proximity` - Determines the max lateral and vertical distance that two elements may be from each other, to be considered to be at the same location. It is determined as a fraction of degrees of Latitude/longitude[3]
- `cellSize` - Determines the size of the unit of distance that is calculated when agent move (agent's speed determines how many times this distance is advanced in a single step). It is determined as a fraction of degrees of Latitude/longitude[3].
- `serviceTime` - The number of steps the `call_breakdown_service` action will take.
- `serviceFee` - The amount of money the `call_breakdown_service` action will cost.

2.2.1 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. The following attributes should be specified for each role:

- `name` - The name by which this role is referenced.
- `speed` - The speed at which the agent moves in a single step (a positive integer). The units of distance are determined by the configuration's attribute `cellSize`.
- `loadCapacity` - Determines the maximum volume that the agent is able to carry (a positive integer).
- `batteryCapacity` - Determines the maximum possible battery charge (a positive integer).

The `roads` section determines the kind of paths that the agents of this role can take for moving from one location to another. Each one is defined by a `road` tag with a single `name` attribute (e.g. `<roads><road name="road"/></roads>`). Currently we only define two kinds of road: `road`, which allows the agent to use regular streets, and `air`, which allows an agent to move in straight line to its destination.

[3]For simplification purposes, Latitude and Longitude are used in this regards as if its units were uniform and they formed a perfect grid.

The `tools` section determines which items can be used as tools by agents of this roles, when assembling new items. Each one is defined by a `tool` tag with a single `id` attribute (e.g. `<tools><tool id="tool1"/><tool id="tool2"/></tools>`). The available items are defined in the products section described below.

2.2.2 Products

The `products` section defines the characteristics of all the different items to be used in the simulation. The following attributes should be specified for each item:

- `id` - The id by which this item is referenced.
- `volume` - A positive integer, indicates the volume of this item, which ultimately limits the total number of items that an agent may carry, that may be stored in a storage facility, etc.
- `userAssembled` - Either `true` or `false`, determines whether agents may assemble this product from other products.

When the product may be assembled by agents, a list of requirements (products) follows. An example can be seen below. Each product in the list has three attributes:

- `id` - The id of the item.
- `amount` - The quantity of instances of this product needed to assemble one unit of the root product.
- `consumed` - Either `true` or `false`, determines whether this items are lost during the assembly of the root product (i.e., this product is used as prime matter), or not (i.e., this product is used as a tool).

```
<product id="material1" volume="10" userAssembled="true">
  <requirements>
    <product id="base1" amount="5" consumed="true"/>
    <product id="tool1" amount="1" consumed="false"/>
  </requirements>
</product>
```

2.2.3 Facilities

The `facilities` section defines all the different facilities present in the simulation. Some attributes depend on the type of the facility, but common to all are the `id` of the facility, and `type`, which can be one either `shop`, `workshop`, `storage`, `dump` or `charging`. Additionally, all facilities define their position in the map in the sub-tag `location` which includes two attributes: `lat` and `lon`, as real numbers.

The additional attributes for each type of facility are described next:

- `workshop`:
  - `cost` - An integer indicating the cost associated to using this facility on a single step.
- `storage`:
  - `cost` - An integer indicating the cost (per unit of volume) associated to storing items at this facility.
  - `capacity` - An integer indicating the max possible total sum of the volumes of the items stored at this facility.
- `dump`:
  - `cost` - An integer indicating the cost associated to using this facility on a single step.
- `charging`:
  - `cost` - An integer indicating the cost per unit of battery charge at this charging station.
  - `rate` - An integer indicating the charging rate, i.e., the number of units of battery charge by which the charge of an agent is increased on a single step (when the agent is indeed charged and its max charge is not reached).
  - `concurrent` - An integer indicating the max number of agent that may be effectively charging their batteries at this station the same time (when these number is reached, following agents are temporarily placed into a queue).
- `shop`: Shops do not define extra attributes. Instead, they define a list of products (available for buying at that shop) as shown in the example below. The attributes for each product in this list are:
  - `id` - The id of the item.
  - `amount` - The quantity of instances of this product that the shop holds in stock at the beginning of the simulation.
  - `cost` - An integer indicating the price of a single unit of this item in this shop.
  - `restock` - The number of simulation steps after which a new unit of this item is added to the shop's stock (0 means never).

```
<facility type="shop" id="shop1">
  <location lat="52.3619" lon="9.7299"/>
  <products>
    <product id="item1" cost="5" amount="500" restock="2" />
    <product id="item2" cost="17" amount="50" restock="3"/>
  </products>
</facility>
```

2.2.4 Jobs

The `jobs` section defines the system-created jobs for the simulation. Some attributes depend on the type of job, but most of them are common. All are described below:

- `id` - The id by which the job will be identified.
- `type` - The type of job: either `priced` or `auction`.
- `firstStepAuction`[4] - The number of the simulation step at which this Job shall begin its auction period.
- `firstStepActive` - The number of the simulation step at which this Job shall become active (for auction jobs this means the end of the auction period).
- `lastStepActive` - The number of the simulation step at which this Job shall be finalized if it wasn't completed.
- `reward`[5] - The reward to give that completes this priced job (integer).
- `maxReward`[4] - The biggest bid amount for this job that shall be accepted if no better bid is placed (integer).
- `fine`[4] - The amou
- `storageId` - the id of the storage facility where the items must be delivered in order to complete this job.

The target of the job is defined as a list of products (under the sub-tag `<products>`) where each product (sub-tag `<product>`) has just two attributes: The `id` of the product and the `amount` required. Here is an example of an `auction` job:

```
<job id="job1" type="auction" firstStepAuction="50" firstStepActive="60"
   lastStepActive="200" fine="10000" maxReward="20000" storageId="storage1">
  <products>
    <product id="material1" amount="3"/>
    <product id="material2" amount="3"/>
  </products>
</job>
```

[4]Only for `auction` jobs.
[5]Only for `priced` jobs.

2.2.5 Agents

The `agents` part 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 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 2016 Mars scenario, the class to use is `massim.competition2016.GraphSimulationAgent`.
- `agentcreationclass` - The name of the Java class that will hold the configuration parsed from the `configuration` child tag. For the 2016 Mars scenario, the class to use is `massim.competition2016.GraphSimulationAgentParameter`.

The `configuration` child tag for the 2016 Logistics scenario has three attribute: `roleName`, which refers to the name of one of the previously defined roles; and `lat` and `lon` that define the initial location of the agent.

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 timout for messages.
- `username` - The user name of the agent.

2.4 Random generation

It is now possible to let some values of the simulation be randomly generated. This can be done via the `generate` section an example of which could look like the following snippet:

```
<generate products="true" facilities="true" jobs="true" agentLoc="true"
  mapCenterLat="51.4885438" mapCenterLon="-0.1112036">

<products
  min="14" max="20"
  minVol="10" maxVol="30"
  assembled="0.6"
  minReq="1" maxReq="3" reqAmountMin="1" reqAmountMax="3"
  valueMin="100" valueMax="150" assembledValueAddMin="80" assembledValueAddMax="100"
  toolPercentage="50"
/>

<facilities quadSize="0.04">
<chargingStations density="0.9" rateMin="50" rateMax="150" costMin="1" costMax="3"
  concurrMin="1" concurrMax="5"/>
<shops
  density="0.8" minProd="3" maxProd="10" priceAddMin="110" priceAddMax="140"
  amountMin="5" amountMax="20" restockMin="1" restockMax="5"
  assembleAddMin="5" assembleAddMax="15"
/>
<dumps density="0.6" costMin="1" costMax="2"/>
<workshops density="0.6" costMin="50" costMax="300"/>
<storages density="0.8" costMin="1" costMax="6" capacityMin="7500"
  capacityMax="15000"/>
</facilities>

<jobs
  rate="0.2" auctionPerc="40"
  productTypesMin="3" productTypesMax="5"
  timeMin="70" timeMax="200"
  valueMin="2000" valueMax="5000"
  rewardSub="0" rewardAddMin="130" rewardAddMax="160"
  badJob="0">
<auction
  auctionTimeMin="2" auctionTimeMax="10"
  fineSub="50" fineAdd="50"
  maxRewardAdd="50"/>
<priced/>
</jobs>
</generate>
```

If anything is configured to be generated, the respective manual configuration entries are ignored.

The parameters can be explained as follows:

- `generate` - the whole block
  - `products` - true, if products should be generated
  - `facilities` - true, if facilities should be generated
  - `jobs` - true, if jobs should be generated
  - `agentLoc` - true, if agents should be positioned randomly
  - `mapCenterLat` - latitude of one point of the reachable part of the graph (used for positioning)
  - `mapCenterLon` - longitude of one point of the reachable part of the graph (used for positioning)
  - `products` - block describing the generation of products
    - `min` - minimum number of products
    - `max` - maximum number of products
    - `minVol` - minimum volume of a product
    - `maxVol` - maximum volume of a product
    - `assembled` - probability of a product needing assembly ($p \in [0, 1]$)
    - `minReq, maxReq` - bounds for numbers of required items for assembly
    - `toolPercentage` - probability of product being a tool ($p \in \mathbb{N}, 0 \leq p \leq 100$)
    - `valueMin, valueMax` - bounds for the (internal) value of the product
    - `reqAmountMin/Max` - bounds for the amount per other product required for assembly
    - `assembledValueAddMin/Max` - percentage (bounds) to add to a product's value if it is assembled
  - `facilities` - block describing the generation of facilities
    - `quadSize` - cellsize of a grid that is used for positioning (in °)
    - `chargingStations` - block describing the generation of charging stations
      - `density` - probability of placing a charging station per quad (or number of charging stations to place if > 1)
      - `rateMin/Max` - bounds for charging rate
      - `costMin/Max` - bounds for facility cost
      - `concurrMin/Max` - bounds for charging slots
    - `shops` - block describing the generation of shops

        - `density` - same as density above
        - `min/maxProd` - bounds for number of different available products per shop
        - `priceAddMin/Max` - bounds for the percentage a shop adds to a product's price (can vary between products of the same shop) (integer values from 0)
        - `amountMin/Max` - bounds for the starting amount a shop sells of a product
        - `restockMin/Max` - bounds for a shop's restock interval (different between products of the same shop)
        - `assembleAddMin/Max` - bounds for the percentage to add to a product's price if it needs assembly (integer values between 0 and 100)
    * `dumps` - block describing the generation of dump locations
        - `density` - same as density above
        - `costMin/Max` - bounds for the facility's cost
    * `workshops` - block describing the generation of workshops
        - `density` - same as density above
        - `costMin/Max` - bounds for the facility's cost
    * `storages` - block describing the generation of storages
        - `density` - same as density above
        - `costMin/Max` - bounds for the facility's cost
        - `capacityMin/Max` - bounds for the storage's capacity
- `jobs` - block describing the generation of jobs
    * `rate` - the exponential arrival rate of jobs
    * `auctionPerc` - probability of a job being an auction (integer values between 0 and 100)
    * `productTypesMin/Max` - minimum/maximum number of different products required for a job
    * `timeMin/Max` - bounds for the time a job is active
    * `valueMin/Max` - bounds for a job's value
    * `rewardSub/AddMin/AddMax` - how much to subtract from (max) or add (bounds) to the job's reward
    * `badJob` - probability of a job being bad (amount is subtracted from reward)
    * `auction` - block describing the characteristics of an auction
        - `auctionTimeMin/Max` - bounds for the duration of the auction part
        - `fineSub/Add` - how the fine can be modified at most
        - `maxRewardAdd` - how much to add at most to the maximum reward (the highest value that can be bid)