ELMS: An Environment Description Language for Multi-Agent Simulation

Fabio Y. Okuyama[1], Rafael H. Bordini[2,1], and Antônio Carlos da Rocha Costa[3,1]

[1] Programa de Pós-Graduação em Computação, Universidade Federal do Rio Grande do Sul (UFRGS), Porto Alegre RS, Brazil
okuyama@inf.ufrgs.br

[2]* Department of Computer Science, University of Liverpool, Liverpool L69 3BX, U.K.
R.Bordini@csc.liv.ac.uk

[3] Escola de Informática, Universidade Católica de Pelotas (UCPel), Pelotas RS, Brazil
rocha@atlas.ucpel.tche.br

Abstract. This paper presents ELMS, a language used for the specification of multi-agent environments. This language is part of the MAS-SOC approach to the design and implementation of multi-agent based simulations. The approach is based on specific agent technologies for cognitive agent programming and high-level agent communication, as well as ELMS. We here concentrate on introducing ELMS, which allows the description of environments in which agents are to be situated during simulations. The ELMS language also allows the definition of the agents' perceptible properties and the kinds of (physical) interactions, through action and perception, an agent can have with the objects of the environment or the perceptible representations of the other agents in the environment.

1 Introduction

The goal of our overall project is to develop an approach and platform for the development of multi-agent based social simulations, incorporating agent technologies for specifying and running cognitive agents. When a multi-agent system is fully computational (i.e., not situated in the real world, the Internet, etc.), the specification of the (simulated) environment where agents are situated is an important task in the engineering of the system, which is not, however, normally addressed in the literature: environments are usually simply considered as "given", or sometimes environments are themselves modelled as agents. Nevertheless, the characteristics of environments are quite different from those of cognitive agents. Therefore, in our practical work, we identified the need for the use of a language specifically designed for the specification of multi-agent environments.

Based on that experience, we have developed a prototype of an interpreter for an environment definition language, presented in detail in [1] and mentioned in [2]. The

* Current affiliation: Department of Computer Science, University of Durham, Durham DH1 3LE, U.K. E-mail: R.Bordini@durham.ac.uk.

language has been designed to support the description of environments for our multi-agent based social simulations (although it may turn out to be useful more generally). Besides the basic environment properties and objects, the language provides the means for the specification of the "physical" representation of a simulated agent, which we refer to as the "body of an agent"[4], as well as the various kinds of physical interactions, through action and perception, among agents and objects or other agents in the environment.

This paper is structured as follows. Sect. 2 covers the main ideas of the MAS-SOC approach to the development of multi-agent based social simulation. We discuss the classes of environments that can be modelled with ELMS in Sect. 3. Sect. 4 presents ELMS itself, the language we introduce in this paper and that is designed specifically for the modelling of multi-agent environments. Then we describe how ELMS environments are run in Sect. 5. Besides small examples given in Sect. 4, we also give a complete example in Appendix Appendix A .

2 The MAS-SOC Project

The main goal of the MAS-SOC project (**M**ulti-**A**gent **S**imulations for the **SOC**ial Sciences) is to provide a framework for the creation of agent-based social simulations that, ideally, should not require much experience in programming from users [2]. In particular, it should allow for the design and implementation of cognitive agents and their social actions. A graphical user interface facilitates the specification of environments, agents (their beliefs and plans), and the simulation as a whole. It also helps the management of libraries of simulation components. From the information input by the user, the system generates source codes for the interpreter used for agent reasoning (from the representations of agents' mental attitudes), and for the ELMS interpreter, whereby environment objects and agents' bodies are simulated.

Agents' practical reasoning is specified in AgentSpeak(L)[3], using the ***Jason*** interpreter [4] (see also [5]). We do not discuss here the AgentSpeak(L) programming language, but one can refer to the papers mentioned above, as well as [6, 7], for a complete account of that language. We here concentrate on presenting the ELMS language and its interpreter.

The interaction between the interpreters (for agents and the environment) and the graphical interface for creating and controlling the simulations is made possible by the SACI toolkit, developed as part of the work reported in [8]. This tool also supports the interactions of agents with the environment (perception and action) as well as speech-act based agent communication, including interactions such as plan exchange[5]. SACI also provides the infra-structure that makes it possible for us to run distributed simulations, thus facilitating large-scale simulations with cognitive agents.

[4] Note that in referring to agent's bodies we do not mean to say that our approach is only applicable to *embodied agents*. By "body" we simply mean whatever physical properties of an agent that may be *perceptible* by other agents in the environment. This is quite general: if an *environment* metaphor is present at all in the multi-agent system being developed, in all likelihood some characteristics of the agents will be perceivable by other agents.

[5] This will be available in ***Jason*** soon, as reported in [5].

In summary, when using the MAS-SOC approach to develop a simulation of a social system (where agents have cognitive features), the procedure is as follows: one first defines an environment in ELMS (specifying objects and their interactions, the "bodies" of the agents, and the ways these can interact with the objects through sensors and effectors), then one defines the agents' cognitive aspects with the use of AgentSpeak(L).

Providing mechanisms for specifying social structures explicitly (e.g. groups, organisations) is part of our objectives for future work, which should also include an attempt to reconcile cognition and emergence. This latter objective is inspired by Castelfranchi's idea that only social simulation with cognitive agents ("mind-based social simulations", as he calls it) will allow the study of agents' minds individually and the emerging collective actions, which co-evolve determining each other [9]. In others words, we aim (as a long term objective) to provide the basic conditions for MAS-SOC to be used in the study of a fundamental problem in the social sciences, which is of the greatest relevance in multi-agent systems as well: the micro-macro link problem [10].

3 Multi-Agent Environments

According to Wooldridge [11], agents are computational systems situated in some environment, and are capable of autonomous action in this environment in order to meet their design objectives. Agents perceive and interact with each other via the environment, and they act upon it so that it reaches a certain state where their goals are achieved. Therefore, environment modelling is an important issue in the development of multi-agent systems where agents do not act directly on a physical or existing environment (e.g., as robots with real sensors and effectors, or Internet agents). This applies to reactive as well as cognitive agent societies (as discussed below). Nevertheless, the multi-agent systems literature seldom considers this part of the engineering of agent-based systems, in particular when dealing with cognitive agents: environments are simply *assumed* as given.

In a reactive multi-agent system, the environment plays a major role. Since reactive agents have no memory and no high-level (i.e., speech-act based) direct communication with each other, it is only perception of the environment that allows them to make decisions on how to act. On the other hand, cognitive agents have an internal representation of the environment, yet they make decisions (e.g., to adopt new goals, or to change courses of actions) based on the changes that perception of the environment causes on that representation. Thus, environment modelling is equally important for both classes of multi-agent systems. Although some multi-agent systems may be situated in an existing environment, in agent-based simulations the environment is necessarily a computational process too, so modelling multi-agent environments is always an important issue in simulations.

In [12], a number of characteristics that can be used to classify environments is given. We refer to those classifications below so that we can characterise the classes of environments that can be defined with ELMS.

Accessible vs. inaccessible: Using the ELMS approach, agents have access only to the environment properties that the simulation designer has chosen to make perceptible to them. Thus, making an ELMS environment accessible or inaccessible is a designer's decision.

Deterministic vs. nondeterministic: As ELMS environments can be inaccessible, and given that there are multiple agents that can change the environment simultaneously, from the point of view of an agent, an ELMS environment can appear to be nondeterministic.

Episodic vs. non-episodic: In ELMS environments, the current state is a consequence of the previous one and the actions taken by the agents in it. With cognitive agents, past actions may influence future actions, so each simulation cycle is unlikely to be just an isolated episode of perceiving and acting (although it is possible to use this approach for simple reactive agents, this is not its intended use).

Static vs. dynamic: An ELMS environment is meant to be shared by multiple agents. As various agents can act on this environment, an agent's action may disable another agent's action. Thus, from the point of view of agents, the environment can seem dynamic.

Discrete vs. continuous: ELMS environments tend to be discrete, through the use of a grid to represent a physical space, although this is not compulsory.

To summarise, ELMS can be used to specify environments that are (from the point of view of the agents): inaccessible, non-deterministic, non-episodic, and dynamic; however, they are usually discrete. This class of environments is the most complex and comprehensive, except for the class of environments that are continuous besides all that. However, continuous environments are notoriously difficult to simulate; although ELMS does not prevent that, it does not give much support in that respect either. We believe that ELMS allows the definition of rather complex environments, supporting a wide range of multi-agent applications (in particular, but not exclusively, for social simulation).

4 The ELMS Language

Agents in a multi-agent system interact with the environment in which they are situated and interact with each other (possibly through the shared environment). Therefore, the environment has an important role in a multi-agent system, whether the environment is the Internet, the real world, or some simulated environment. ELMS is intended as a specification language for the latter form of environments.

We understand as environment modelling, the modelling of external aspects that an agent needs as input to its reasoning and for deciding on its course of action. Also, there is the need to model explicitly the physical actions and perceptions that the agents can do on the environment, as will be seen in Sect. 4.1.

This section introduces the main aspects of the language we defined for the specification of the simulated environment that is to be shared by the agents in a multi-agent system. The language is called ELMS (**E**nvironment Description **L**anguage for **M**ulti-Agent **S**imulation).

4.1 Modelling Environments with ELMS

An environment description is a specification of the properties and behaviour of the environment. In our approach, we also include in such specification the definition of the features of the simulated "bodies" of the agents. The modelling of such "physical" aspects of an agent (or agent class, more precisely) includes the definitions of its properties that may be perceived by others agents, the definitions of the kinds of perceptions that are available for that agent, and the actions that the agent is able to perform in the environment.

The definition of the environment includes mainly sets of: objects, to which we interchangeably refer as *resources* of the environment; reactions that objects display when agent actions affect them; an (optional) grid to allow the explicit handling of the spatial positioning of agents and objects in the environment; and the properties of the environment to which external observers (e.g., the users) have access.

The objects that are part of an environment can be modelled as a set of properties and a set of actions that characterise the object's behaviour in response to stimuli. That is, objects can *react*—only agents are pro-active. Agents can be considered components of the environment insofar as, from the point of view of one agent, any other agent is a special component of the environment (however, only certain properties of an agent can be perceived by other agents, and this must be specified by designers of agent-based simulations). Thus, to define agents from this point of view, it is necessary to list all properties that define their perceptible aspects, a list of actions that they are able to execute (pro-actively), and a list of the types of perception to which they have access. From the point of view of the environment, the deliberative activities of an agent are not relevant, since they are internal to the agent, i.e., they are not observable to the other agents in the environment. As mentioned before, in the MAS-SOC approach the mental aspects of agents are described with the AgentSpeak(L) language.

Quite frequently, spatial aspects of the environment are modelled in agent simulations by means of a grid. Our approach provides a number of features for dealing with grids, if the designer of the environment chooses to have one. In the constructs that make reference to the grid, positions can be accessed by absolute or relative coordinates. Relative coordinates are prefixed by '$+$' and '$-$' signs, so $(+1, -1, +0)$, for example, refers to the position at the upper right diagonal from the agent's current position. However, the grid definition is optional, as some simulations may not require any spatial representation. Clearly, there are simulations where the topology resulting from specific types of agent and object positioning is the main issue of interest for the investigation for which the simulations are being used. In contrast, there are also simulations where the existence of a topology is not relevant at all as, e.g., in a stock market simulation, where the main issue under consideration relate to the agent interactions themselves, and perhaps agents' interactions with some types of resources. In order to make ELMS as general as possible, we chose to make the grid an optional feature.

For the definition of the types of perception to which each agent class has access, it is necessary to define which properties of the environment, agents, and objects are to be perceived. The conditions associated with each perceptible property can be specified as well. That is, environment designers can control: which properties of objects will be accessible to the "minds" of the agents that are given access to a certain perception type, and under which conditions each (potentially perceivable) property will be effectively perceived. An action is defined as a sequence of changes in properties (of the environment in general, its resources, or agents) that it causes, along with the preconditions that must be satisfied for the action to be actually executed in the environment.

Note that our approach allows for quite flexible environment definitions. It is the environment designer who decides which properties of the environment can be perceptible by agents, and which are observable by external users (as well as defining how actions change the environment). Any properties associated with objects or with agents themselves can potentially be specified as perceptible/observable properties.

4.2 Language Constructs in ELMS

The ELMS language uses an XML syntax, which can be somewhat cumbersome to be used directly. However, recall that environment specifications are to be obtained from a graphical interface, so users do not need to bother about the language syntax. Still, environment specifications can be written directly in XML with a simple text editor, or some other tool, if the user prefers to do so. The use of XML provides various advantages, for example because of the wide range of XML tools currently available, and it can be useful for the future development of visualisation mechanisms for ELMS-based simulations, particularly if they are to be web-based.

An environment specification in ELMS can make use of constructs of nine main types, and several other constructs that may appear within some of the main ones. There is no special order for the constructs to appear in a specification. The main types of ELMS constructs are listed below.

1. Defining agent bodies:

   Agent Body: This construct defines a class of agent bodies for the agents that may join a simulation with that environment. A specification of an agent-body class contains its name, a list of attributes, a list of actions, and a list of perception types. The list of attributes is defined as before; it characterises the observable properties of this class of agent bodies, from the point of view of the environment and other agents. It is then necessary to specify a list of names for the actions that agents of this type are able to perform in the environment. The set of perceptions is a list of the names of perception types (see below) that are available to agents of this class (i.e., the information that will be accessible to the agent's mind at every reasoning cycle). Note that the same perception and action names can appear in any number of agent-body definitions; that is, they can be used in all the different classes of agent bodies that may execute that type of perception/action (the same applies to reactions for resources). The code sample below defines an agent-body class named `worker_robot` which has as attributes an integer and a boolean value. It is able to perform actions `walk_right`, `walk_left`, `load`, and `unload`. The perceptions that are available for agents belonging to the `worker_robot` class are `vision` and `audition`.

   ```
   <AGENT_BODY  NAME = "worker_robot">
       <INTEGER NAME = "id"> "SELF" </INTEGER>
   ```

```
        <BOOLEAN NAME = "functional"> "TRUE" </BOOLEAN>
        <ACTIONS>
            <ITEM NAME = "walk_right"/>
            <ITEM NAME = "walk_left"/>
            <ITEM NAME = "load"/>
            <ITEM NAME = "unload"/>
        </ACTIONS>
        <PERCEPTIONS>
             <ITEM NAME = "vision"/>
             <ITEM NAME = "audition"/>
        </PERCEPTIONS>
    </AGENT_BODY>
```

Perception: This construct allows the specification of perception types to be listed in agent-body specifications. A perception type definition is formed by a name, an optional list of preconditions, and a list of properties that are perceptible. The listed properties can be any of those associated with the definitions of resources, agents, cells of the grid, or simulation control variables. If all the preconditions (e.g., whether the agent is located on a specific position of the grid) are all satisfied, then the values of those properties will be made available to the agent's reasoner as the result of its perception of the environment. Note that perception can be based on the spatial position of the agent, but this is not mandatory; any type of perception can be defined by the environment designer.

```
<PERCEPTION NAME = "vision">
   <PRECONDITION>
     <EQUAL>
        <OPERAND>
            <ELEMENT_ATT NAME = "SELFCLASS" ATTRIBUTE = "functional">
                <INDEX>"SELF"</INDEX>
            </ELEMENT_ATT>
        </OPERAND>
        <OPERAND> "TRUE" </OPERAND>
     </EQUAL>
   </PRECONDITION>
   <CELL_ATT ATTRIBUTE = "CONTENTS">
        <X> +0 </X>   <Y> +0 </Y>
   </CELL_ATT>
</PERCEPTION>
```

The code sample above defines a perception called `vision`. This perception has as its precondition that the agent must have its `functional` attribute equals to `TRUE`. If the precondition is satisfied, the agent will receive the information about the contents of the cell on the grid where it is currently positioned.

Action: With this construct, the actions that may appear in agent-body definitions are described. An action definition includes its name, an optional list of parameters, an optional list of preconditions, and a sequence of commands which determine what changes in the environment the action causes. The list of parameters specifies the data that will be received from the agent for further guiding the execution of that type of action. The possible commands for defining the consequences of executing an action are assignments of values to attributes (i.e., properties of agents, resources, etc.), and allocations or repositioning of instances of agents or resources within the grid. Resources can also be instantiated or removed by commands in an action. If the preconditions are all satisfied, then all the commands in the sequence of commands will be executed, changing the environment accordingly. To avoid consistency problems, actions are executed atomically. For this reason, they should be defined so as to follow the concept of an atomic action (although this is again not mandatory); recall that more complex courses of actions are meant to be part of agents' internal reasoning[6].

```
<ACTION NAME = "walk_right">
    <PARAMETER NAME="STEPS" TYPE="INTEGER"/>
    <PRECONDITION>
        <LESSTHAN>
            <OPERAND> "STEPS" </OPERAND>
            <OPERAND> 3 </OPERAND>
        </LESSTHAN>
    </PRECONDITION>
    <MOVE>
        <ELEMENT NAME = "SELFCLASS">
                <INDEX>"SELF"</INDEX>
        </ELEMENT>
        <FROM>
            <CELL>
                <X>+0</X> <Y>+0</Y>
            </CELL>
        </FROM>
        <TO>
            <CELL>
                <X>STEPS</X> <Y>+0</Y>
            </CELL>
        </TO>
    </MOVE>
</ACTION>
```

In the example above, an action named `walk_right` is defined. It has as parameter an integer referred to as `STEPS`. The precondition defines that this parameter must be lower than 3. As a result of the execution of this action, the agent will walk, to the right, the number of steps specified by the parameter (i.e., the agent's body location will be moved within the environment representation).

2. Defining the environment:

Grid Options: This is used for a grid definition, if the designer has chosen to have one. The grid can be two or three dimensional, the parameters being the sizes of the grid on the X, Y, and Z axes. Still within the grid definition, a list of cell attributes can be given: the attributes defined here will be replicated for each cell of the grid. Also as part of the cell definition, a list of reactions can be defined for them[7]. The code below exemplifies a definition of a two-dimensional grid that has twenty columns and twenty rows, where each cell has an integer that represents its colour (which defaults to 0) and a boolean variable that keeps the information about whether the cell is occupied (e.g., by

[6] Since agents are constantly perceiving, reasoning, and acting, the actions they execute in the environment should normally be atomic. That is, it is known before the next reasoning cycle whether the previous action was successfully executed, and if it was, its perceptible effects will be noticed by the agent when it does belief revision just before the next reasoning cycle. Although it is possible to make alternative design choices where actions are not atomic, it seems that simulations in particular should be more easily and appropriately engineered this way.

[7] Although the list of reactions is the same for all cells, this does not imply they all have the same behaviour at all times, as reactions can have preconditions on the specific state of the individual cells.

an agent) or not. Each of those cells can have the reactions named `reaction1` and `reaction2`.

```
<DEFGRID SIZEX="20" SIZEY="20" SIZEZ="1">
    <INTEGER NAME = "cellcolour"> 0 </INTEGER>
    <BOOLEAN NAME = "ocuppied"> "FALSE" </BOOLEAN>
    <REACTIONS>
         <ITEM NAME = "reaction1"/>
         <ITEM NAME = "reaction2"/>
     </REACTIONS>
</DEFGRID>
```

Resources: This construct is used to define the objects in an environment (i.e., all the entities of the environment that are not pro-active). A definition of a resource class includes the class name, a list of attributes, and a set of reactions. The attributes are defined in the same way as for the cell attributes (i.e., by the specification of its name, type, and initial value). The reactions that a class of resources can have is given by a list of the names identifying those reactions (see below how reactions are defined).

```
<RESOURCE NAME = "water">
    <STRING  NAME = "state" VALUE = "liquid"/>
    <INTEGER NAME = "temperature"> 23 </INTEGER>
    <INTEGER NAME = "quantity"> 10 </INTEGER>
    <REACTIONS>
      <ITEM NAME = "solidify"/>
      <ITEM NAME = "melt"/>
    </REACTIONS>
</RESOURCE>
```

The code sample above defines a resource class named `water`. It has a string attribute that records its `state` value, and there are two integer values that represent its temperature and quantity. This resource can have the `solidify` and `melt` reactions (i.e., the expected reactions to actions changing its temperature).

Reactions: This part of the specification is where the possible reactions of the objects in the environment are defined. For each type of reaction, its name, a list of preconditions, and a sequence of commands is given. The commands are exactly as described above for actions. All expressions in the list of preconditions must be satisfied for the reaction to take place. Differently from actions, where only one action (per agent) is performed, all reactions that satisfy their preconditions will be executed "simultaneously" (i.e. in the same simulation cycle). In the code sample below, the reaction `melt` is defined. As precondition, the `temperature` attribute must greater than 273 (Kelvin scale) and the `state` attribute must be equal to `solid`. This reaction results in changing the `state` attribute to `liquid`. Note the use of the reserved keyword `SELFCLASS`, which refers to the class of whatever resource type the reaction is associated with, and is useful for programming and code reuse.

```
<REACTION NAME = "melt">
    <PRECONDITION>
        <GREATERTHAN>
            <OPERAND>
                <ELEMENT_ATT NAME = "water" ATTRIBUTE = "temperature">
                    <INDEX> "SELF" </INDEX>
                </ELEMENT_ATT>
           </OPERAND>
           <OPERAND> 273 </OPERAND>
        </GREATERTHAN>
    <EQUAL>
        <OPERAND>
        <ELEMENT_ATT NAME = "SELFCLASS" ATTRIBUTE = "state">
            <INDEX> "SELF" </INDEX>
        </ELEMENT_ATT>
            </OPERAND>
            <OPERAND> "solid" </OPERAND>
    </EQUAL>
    </PRECONDITION>
        <ASSIGN>
            <ELEMENT_ATT NAME = "SELFCLASS" ATTRIBUTE = "state">
                <INDEX> "SELF" </INDEX>
            </ELEMENT_ATT>
            <EXPRESSION> "liquid" </EXPRESSION>
        </ASSIGN>
</REACTION>
```

3. Some operational aspects of a simulation are specified using the following constructs:

Observables: This is how the user defines which properties of the agents, resources, and the environment itself will be sent to the MAS-SOC interface as the result of a simulation cycle; that is, the users specify the particular properties of the simulated "world" which they are interested in observing from the simulator interface. The properties to be selected as observable can be any of those associated with instances of resources and agents, cells of the grid, and simulation control variables. The observable items are defined in the same way as the perceptible items in a perception definition.

Initialisation: This part of the specification allows resources in the environment to be instantiated and allocated to grid positions in the initial state of the simulation (resources can also be created in the environment or allocated to the grid dynamically during simulation). All commands in this section are only executed before the start of the simulation. The initialisation is defined in the same way as command sequences in action definitions.

Simulation Values: In this section of an ELMS definition, the values for the attributes of instances of resources and agents that are currently part of a simulation can be defined. The environment controller process (see Sect. 5) can generate a snapshot of a running simulation by filling in such values from those contained in its data structures. With the constructs described above, the classes of agents and resources are simply defined; instantiations can be made in the initialisation section, or in this one for a simulation that is already running. Also in this section, the position of instances of agents and resources on the grid can be defined. The values for environment control variables can be defined by assignment commands over predefined variable names (e.g., the current simulation step number). This feature allows the user to save the simulation state for later execution, or to make on-the-fly changes in the environment (via the interface or by changing the ELMS code manually) to induce various different situations in a simulation. Such simulation snapshots may also be useful for complex forms of visualisation of multi-agent simulations.

Next, we show some of the constructs that are used in ELMS to define commands, expressions, and attributes.

Attribute Definition: The types of attributes supported by ELMS are: boolean, integer, float, and string. Attributes are defined by a specific XML tag for each type and an initial value. The initial value can be a constant or an expression (except for string expressions, which are currently not allowed).

Expressions: In ELMS, some mathematical, logical, and relational operators are available. The available relational operators are `EQUAL`, `UNEQUAL`, `GREATERTHAN`, and `LESSTHAN`. For mathematical expressions, the following constructs are available: `ADD`, `SUBTRACT`, `MULTIPLY`, `DIVIDE`, `MOD`, `SUM` (summatory), and `PROD` (product). The available logical operators are: `AND`, `OR`, and `NOT` (negation). Operands of relational operators can be another operation, a constant, and a cell, resource, or agent attribute. It is also possible to use the commands `RAND` and `RANDOM`. The former command generates a pseudo-random number between 0 and 1, while the latter command has as parameters a minimum value (inclusive) and a maximum value (exclusive), generating a pseudo-random integer in this range. These commands can be used in all parts of the code, except within the "simulation values" section (where they are not required).

Preconditions: The preconditions for actions, reactions, and perceptions are defined through a sequence of logical operations. If a logical operator is not explicitly defined, `AND` is assumed (as it is most commonly used). For example, the following code:

```
<PRECONDITION>
    <EQUAL>...</EQUAL>
    <GREATERTHAN>...</GREATERTHAN>
</PRECONDITION>
```

has the same effect as:

```
<PRECONDITION>
    <AND>
        <OPERAND>
            <EQUAL>...</EQUAL>
        </OPERAND>
        <OPERAND>
            <GREATERTHAN>...</GREATERTHAN>
        </OPERAND>
    </AND>
</PRECONDITION>
```

Commands: Below, we use *element* to refer to both resources and agents. The commands available in ELMS are: assignment (`ASSIGN`), allocation of an element on the grid (`IN`), random allocation of an element on the grid (`IN_RAND`), element removal from the grid (`OUT`), changing the position of an element on the grid (`MOVE`), instance creation (`NEW`), instance exclusion (`DELETE`).

The `MOVE` command has as parameters an element, its original position, and the destination. Note that one element can occupy more than one position on the grid, but elements have a reference point used for relative position calculation: the cell to which it was first allocated. When using the `MOVE` command, the whole element is moved by changing its reference point.

5 Running ELMS Environments

The simulation of the environment itself is done by a process that controls the access and changes made to the data structure that represents the environment (in fact, only that process can access the data structure); the process is called the *environment controller*. The data structure that represents the environment is generated by the ELMS interpreter for a specification in ELMS given as input. In each simulation cycle, the environment controller sends to all agents currently taking part in the simulation the percepts to which they have access (as specified in ELMS). Perception is transmitted in messages as a list of ground logical facts. After sending perception, the process waits for the actions that the agents have chosen to perform in that simulation cycle.

The execution of a synchronous simulation in ELMS, from the point of view of the environment controller, follows the steps below:

1. execute the commands in the initialisation section before the start of the simulation;
2. check which percepts from the agent's perception list are in fact available at that time (check which perceivable properties satisfy the specified preconditions);
3. send the resulting percepts (those that satisfied the preconditions) to the agents;
4. wait until the chosen actions (to be performed in that cycle) have been received from all agents[8];
5. the order of the actions in the queue of all received actions is changed randomly to allow each agent to have a chance of executing its action first;
6. check if the first action in the queue satisfies its precondition for execution;
7. execute the action, if the precondition was satisfied;
8. if not, send a message with "`@fail`" as content to the agent;
9. remove the action at the front of the queue;
10. if there are any actions left in the queue, go to step 6;
11. check and execute all reactions defined for resources in the environment which had their preconditions satisfied;
12. send the set of properties defined as "observables" to the interface or to an output file previously specified;
13. if the step counter has not yet reached the maximum value defined by the user, go to the step 2.

Note that this corresponds to the (default) synchronous simulation mode. An asynchronous mode is also available.

For the communication between the agents, the SACI (Simple Agent Communicaiton Infrastructure) [8] toolkit is used. It supports KQML-based communication and provides an infrastructure for managing distributed agents. All agents participating in a simulation are registered to a SACI society. Through it, every member of the society can communicate with other members by simply sending messages addressed with that member's name in the society (regardless of the host where the agent interpreter is running). This way, it is possible for any SACI-based agent to interact within a simulation, so that, for example, we can make available an interface for human "agents" to interact within a MAS-SOC simulated society (although this is not currently one of the main goals of the MAS-SOC project). This feature (of open SACI societies) can also be very useful for simulation debugging and analysis (e.g., "observer" agents can be introduced to monitor aspects of a simulation).

[8] Agents send a message with "`true`" as its content if they have chosen not to execute an action in that cycle.

SACI is available as free software at `http://www.lti.pcs.usp.br/saci/`. The ELMS interpreter too will be made available as free software in the near future.

6 Conclusion

This paper introduced the ELMS language, used for the specification of the characteristics of agent "bodies" and the environment to be shared by agents in a multi-agent social simulation. Although the ELMS interpreter is tailored for social simulation implemented according to the MAS-SOC approach, it could be useful for other symbolic approaches as well. The MAS-SOC approach consists of a distinct combination of multi-agent techniques that we consider as the most adequate for the construction of multi-agent based social simulations. We believe that MAS-SOC allows for quite flexible definitions of multi-agent social simulations, taking into considerations not only cognitive agents but also the environment shared by them.

As future work, there are several improvements to the platform that we plan to carry out. In particular, we plan to concentrate on higher-level aspects of agent-based simulations which are particularly important for social simulation, such as the specification of social structures within agent societies, as well as using the ideas of exchange values from [13] to support social interactions. In the long term, we aim at investigating the necessary mechanisms for reconciling cognition and emergence following the ideas of [9], and incorporating such mechanisms into MAS-SOC, thus allowing it to be used in investigations of the micro-macro link problem. We are currently considering the implementation of various social simulation applications.

References

1. Okuyama, F.Y.: Descrição e geração de ambientes para simulações com sistemas multiagente. Dissertação de mestrado, PPGC/UFRGS, Porto Alegre, RS (2003). In Portuguese.
2. Bordini, R.H., Okuyama, F.Y., de Oliveira, D., Drehmer, G., Krafta, R.C.: The MAS-SOC approach to multi-agent based simulation. In Lindemann, G., Moldt, D., Paolucci, M., eds.: Proceedings of the First International Workshop on Regulated Agent-Based Social Systems: Theories and Applications (RASTA'02), 16 July, 2002, Bologna, Italy (held with AAMAS02) — Revised Selected and Invited Papers. Number 2934 in the LNAI Series, Berlin, Springer-Verlag (2004), 70–91.
3. Rao, A.S.: AgentSpeak(L): BDI agents speak out in a logical computable language. In Van de Velde, W., Perram, J., eds.: Proceedings of the Seventh Workshop on Modelling Autonomous Agents in a Multi-Agent World (MAAMAW'96), 22–25 January, Eindhoven, The Netherlands. Number 1038 in the LNAI Series, London, Springer-Verlag (1996), 42–55.
4. Bordini, R.H., Hübner, J.F., et al.: ***Jason***: A Java-based agentSpeak interpreter used with saci for multi-agent distribution over the net. Manual, first release edn. (2004) `http://jason.sourceforge.net/`.
5. Ancona, D., Mascardi, V., Hübner, J.F., Bordini, R.H.: Coo-AgentSpeak: Cooperation in AgentSpeak through plan exchange. In Jennings, N.R., Sierra, C., Sonenberg, L., Tambe, M., eds.: Proceedings of the Third International Joint Conference on Autonomous Agents and Multi-Agent Systems (AAMAS-2004), New York, NY, 19–23 July, New York, NY, ACM Press (2004), 698–705.

6. Moreira, Á.F., Vieira, R., Bordini, R.H.: Extending the operational semantics of a BDI agent-oriented programming language for introducing speech-act based communication. In Leite, J., Omicini, A., Sterling, L., Torroni, P., eds.: Declarative Agent Languages and Technologies, Proceedings of the First International Workshop (DALT-03), held with AAMAS-03, 15 July, 2003, Melbourne, Australia (Revised Selected and Invited Papers). Number 2990 in the LNAI Series, Berlin, Springer-Verlag (2004), 135–154.
7. d'Inverno, M., Luck, M.: Engineering AgentSpeak(L): A formal computational model. Journal of Logic and Computation **8** (1998), 1–27.
8. Hübner, J.F.: Um Modelo de Reorganização de Sistemas Multiagentes. PhD thesis, Universidade de São Paulo, Escola Politécnica (2003).
9. Castelfranchi, C.: The theory of social functions: Challenges for computational social science and multi-agent learning. Cognitive Systems Research **2** (2001), 5–38.
10. Conte, R., Castelfranchi, C.: Cognitive and Social Action. UCL Press, London (1995).
11. Wooldridge, M.: Intelligent agents. In Weiß, G., ed.: Multiagent Systems—A Modern Approach to Distributed Artificial Intelligence. MIT Press, Cambridge, MA (1999), 27–77.
12. Russel, S., Norvig, P.: Artificial Intelligence — A Modern Approach. Prentice-Hall, Englewood Cliffs, NJ (1995).
13. Rodrigues, M.R., da Rocha Costa, A.C., Bordini, R.H.: A system of exchange values to support social interactions in artificial societies. In Rosenschein, J.S., Sandholm, T., Michael, W., Yokoo, M., eds.: Proceedings of the Second International Joint Conference on Autonomous Agents and Multi-Agent Systems (AAMAS-2003), Melbourne, Australia, 14–18 July, New York, NY, ACM Press (2003). 81–88.
14. Bordini, R.H., Fisher, M., Visser, W., Wooldridge, M.: Verifiable multi-agent programs. In Dastani, M., Dix, J., El Fallah-Seghrouchni, A., eds.: Programming Multi-Agent Systems, Proceedings of the First International Workshop (ProMAS-03), held with AAMAS-03, 15 July, 2003, Melbourne, Australia (Selected Revised and Invited Papers). Number 3067 in the LNAI Series, Berlin, Springer-Verlag (2004), 72–89.

Appendix A Example of an ELMS Specification

We provide below a very simple example so as to illustrate the use of the ELMS language for specifying an environment. A robot (simulated by an AgentSpeak(L) agent) must find garbage in a territory that is modelled as a 10×10 grid. When a piece of garbage is found, the robot collects it and takes it to an incinerator located at the centre of the territory that is to be kept clean. In the environment used in simulations carried out to observe the behaviour of the AgentSpeak(L) agent, garbage randomly "appears" on the grid. We have included some redundant attributes in the example just so that we could show how to use various ELMS constructs. Due to the lack of space, only a few excerpts of the code are explained with accompanying text.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE ENVIRONMENT SYSTEM "elms.dtd">
<ENVIRONMENT NAME = "TERRITORY">

<!-- AGENTS SECTION  -->

    <AGENT_BODY NAME="robot">
      <BOOLEAN NAME = "loaded"> "FALSE" </BOOLEAN>
      <PERCEPTIONS>
          <ITEM NAME = "self_info"/>
          <ITEM NAME = "cur_position"/>
      </PERCEPTIONS>
      <ACTIONS>
          <ITEM NAME = "load"/>
          <ITEM NAME = "unload"/>
          <ITEM NAME = "move_north"/>
          <ITEM NAME = "move_south"/>
          <ITEM NAME = "move_east"/>
          <ITEM NAME = "move_west"/>
      </ACTIONS>
    </AGENT_BODY>
```

This excerpt defines a class of agent bodies named `robot`. This class has as attribute a boolean value named `loaded` which is true whenever the robot is carrying a piece of garbage. The robot is able to perform two types of perceptions: `self_info` and `cur_position`, which will be defined in the perception section below. Also, it is able to perform six different actions, as listed above and defined later in the action section.

```
<!-- PERCEPTIONS SECTION  -->

    <PERCEPTION NAME="cur_position">
      <CELL_ATT ELEMENT = "garbage" ATTRIBUTE ="size" > // SIZE OF THE GARBAGE
            <X> +0 </X> <Y> +0 </Y>                     // PRESENT IN CURRENT CELL
      </CELL_ATT>
      <CELL_ATT ATTRIBUTE = "colour">
            <X> +0 </X> <Y> +0 </Y>
      </CELL_ATT>
    </PERCEPTION>
```

This perception allows the agent to have an explicit representation of information about the cell where it is currently positioned: the size of the piece of garbage in that cell (if there is any) and the cell's colour, which is represented by an integer. No information about neighbouring cells is perceived.

```
    <PERCEPTION NAME="self_info">
        <ELEMENT_ATT NAME = "SELFCLASS" ATTRIBUTE = "loaded">
            <INDEX>"SELF"</INDEX>
        </ELEMENT_ATT>
    </PERCEPTION>



<!-- ACTIONS SECTION  -->

    <ACTION NAME="move_east">
      <MOVE>
        <ELEMENT NAME = "SELFCLASS">
            <INDEX>"SELF"</INDEX>
        </ELEMENT>
        <FROM>
            <CELL>
                <X>+0</X>
                <Y>+0</Y>
            </CELL>
        </FROM>
        <TO>
            <CELL>
                <X>+1</X>
                <Y>+0</Y>
            </CELL>
        </TO>
      </MOVE>
    </ACTION>

    <ACTION NAME="move_north">                    // SUMMARISED
    <ACTION NAME="move_south">
    <ACTION NAME="move_west">
    <ACTION NAME="load">
        <PARAMETER NAME="G1" TYPE="INTEGER" />
        <PRECONDITION>
            <UNEQUAL>                            // FAIL CHANCE = 1/20
                <OPERAND>
                    <RANDOM MIN="0" MAX="20"/>
                </OPERAND>
                <OPERAND> "10" </OPERAND>
            </UNEQUAL>
            </PRECONDITION>
            <OUT>
                <ELEMENT NAME = "garbage">
                    <INDEX> "G1" </INDEX>
                </ELEMENT>
                <CELL>
                    <X>+0</X>
                    <Y>+0</Y>
                </CELL>
            </OUT>
            <ASSIGN>
                <ELEMENT_ATT NAME = "SELFCLASS" ATTRIBUTE = "loaded">
                    <INDEX>"SELF"</INDEX>
                </ELEMENT_ATT>
                <EXPRESSION> "TRUE" </EXPRESSION>
            </ASSIGN>
    </ACTION>
```

The action above removes the garbage from the cell and changes the `loaded` attribute of the agent. This action can fail a random number of times, as it has as precondition that a random number between 0 to 20 must not be equals to 10 or else the action will fail. This nondeterminism models possible failures of the robot's grabbing mechanism. The action has as parameter, referred as `G1`, the index of the garbage that will be loaded.

```
    <ACTION NAME="unload">
        <PARAMETER NAME="G1" TYPE="INTEGER" />
        <IN>
            <ELEMENT NAME = "garbage">
                <INDEX>"G1"</INDEX>
            </ELEMENT>
            <CELL>
                <X>+0</X>
                <Y>+0</Y>
            </CELL>
        </IN>
        <ASSIGN>
            <ELEMENT_ATT NAME = "incinerator" ATTRIBUTE = "empty">
                <INDEX>
                    <CELL_ATT ELEMENT = "incinerator" ATTRIBUTE ="id" >
                        <X>+0</X>
                        <Y>+0</Y>
                    </CELL_ATT>
                </INDEX>
            </ELEMENT_ATT>
            <EXPRESSION> "TRUE" </EXPRESSION>
        </ASSIGN>
        <ASSIGN>
            <ELEMENT_ATT NAME = "SELFCLASS" ATTRIBUTE = "loaded">
                <INDEX>"SELF"</INDEX>
            </ELEMENT_ATT>
            <EXPRESSION> "FALSE" </EXPRESSION>
        </ASSIGN>
    </ACTION>


<!-- GRID DEFINITIONS SECTION  -->
```

```
    <DEFGRID SIZEX="10" SIZEY="10">
        <INTEGER NAME = "colour">
            <RANDOM MIN="0" MAX="16"/>
        </INTEGER>
        <REACTIONS>
            <ITEM NAME ="sprout_trash"/>
        </REACTIONS>
    </DEFGRID>



<!-- RESOURCES SECTION  -->

    <RESOURCE NAME="garbage">
        <INTEGER NAME="size">  5  </INTEGER>
    </RESOURCE>

    <RESOURCE NAME="incinerator">
        <BOOLEAN NAME="empty"> "TRUE" </BOOLEAN>
        <INTEGER NAME="id">    "SELF" </INTEGER>
        <REACTIONS>
            <ITEM NAME ="burn"/>
        </REACTIONS>
    </RESOURCE>


<!-- REACTIONS SECTION  -->

    <REACTION NAME="burn">
      <PRECONDITION>
        <EQUAL>
            <OPERAND>
                <ELEMENT_ATT NAME = "SELFCLASS" ATTRIBUTE = "empty">
                    <INDEX>"SELF"</INDEX>
                </ELEMENT_ATT>
            </OPERAND>
            <OPERAND> "FALSE" </OPERAND>
        </EQUAL>
      </PRECONDITION>
        <DELETE NAME = "garbage">
            <INDEX>
                <CELL_ATT NAME = "garbage" ATTRIBUTE = "id">
                    <X>+0</X>
                    <Y>+0</Y>
                </CELL_ATT>
            </INDEX>
        </DELETE>
    <ASSIGN>
        <ELEMENT_ATT NAME = "SELFCLASS" ATTRIBUTE = "empty">
        <INDEX>"SELF"</INDEX>
        </ELEMENT_ATT>
        <EXPRESSION>
            "TRUE"
        </EXPRESSION>
    </ASSIGN>
    </REACTION>

    <REACTION NAME="sprout_trash">
      <PRECONDITION>
        <EQUAL>
            <OPERAND>
                <RANDOM MIN="0" MAX="100"/>
            </OPERAND>
            <OPERAND> 10 </OPERAND>
        </EQUAL>
      </PRECONDITION>
        <NEW NAME = "incinerator">
```

```
            <N>1</N>
        <CELL>
             <X>+0</X>
             <Y>+0</Y>
        </CELL>
        </NEW>
    </REACTION>



<!-- OBSERVABLES SECTION  -->

    <OBSERVABLE>
            <CELL_ATT ATTRIBUTE = "colour">
                <X> "ALL"</X>
                <Y> "ALL" </Y>
            </CELL_ATT>
            <CELL_ATT ATTRIBUTE = "CONTENTS">
                <X> "ALL"</X>
                <Y> "ALL" </Y>
            </CELL_ATT>
    </OBSERVABLE>



<!-- INITIALIZATION SECTION  -->

    <INITIALIZATION>
        <NEW NAME = "incinerator">
            <N>1</N>                        //ONE INSTANCE
        <CELL>
             <X>4</X>
             <Y>4</Y>
        </CELL>
        </NEW>
    </INITIALIZATION>
</ENVIRONMENT>
```

The simple AgentSpeak(L) code that could be used for the robot's reasoning has not been included, as the focus here in on modelling environments, but such code can be found in [14] and is one of the examples distributed with ***Jason*** [4].