

# MAS-SOC: A Social Simulation Platform based on Agent-Oriented Programming

Rafael H. Bordini<sup>1,4</sup>, Antônio Carlos da Rocha Costa<sup>2,4</sup>, Jomi F. Hübner<sup>3</sup>,  
Fabio Y. Okuyama<sup>4</sup>, Álvaro F. Moreira<sup>4</sup>, and Renata Vieira<sup>5</sup>

<sup>1</sup> Department of Computer Science – University of Durham  
R.Bordini@durham.ac.uk

<sup>2</sup> Escola de Informática – Universidade Católica de Pelotas (UCPel)  
rocha@atlas.ucpel.tche.br

<sup>3</sup> Departamento de Sistemas e Computação – Universidade Regional de Blumenau (FURB)  
jomi@inf.furb.br

<sup>4</sup> PPGC – Universidade Federal do Rio Grande do Sul (UFRGS)  
{afmoreira, okuyama}@inf.ufrgs.br

<sup>5</sup> PIPCA – Universidade do Vale do Rio dos Sinos (UNISINOS)  
renata@exatas.unisinos.br

## 1 Introduction

After almost a decade during which research on agent-oriented programming struggled to deliver any concrete results, the last couple of years has seen an impressive improvement in the quality of research in the area, as well as a considerable increase in the number of researchers involved in it. This led to the creation of two international workshop series on the subject (4; 16; 25; 24).

In this paper, we describe on going work that is aimed at using the recent developments in agent-oriented programming for developing a simulation platform, called MAS-SOC, which should be particularly suitable for the area of Social Simulation. This work draws on a series of papers where we contributed both to practical extensions of an agent-oriented programming language (1; 28; 3), as well as to its formal semantics (7; 27; 41). The variety of agent-programming languages and supporting tools now widely discussed in the literature are aimed particularly to support the ever increasing transfer of multi-agent systems technologies to Industry (see, e.g., the special section on the “Industrial Uptake of Agent Technology” in issue 16 of AgentLink News<sup>1</sup>). However, if such languages are to effectively improve the development of sophisticated industrial applications, it is likely they will have similar, if not greater, impact on the development of social simulations, where the notions of autonomous agents and multi-agent systems are known to have introduced major advantages.

Whilst our approach draws on the progress of such programming languages, there are still a great deal of multi-agent systems techniques that need to be incorporated. In (36), a qualitative mechanism for regulating social exchange was presented; this is one of the techniques we plan to incorporate into our social simulation approach. Further, we still lack the means for specifying social structures explicitly (e.g., groups,

---

<sup>1</sup> Available at <http://www.agentlink.org/newsletter/>.

organisations), which is very important for social simulation. To provide mechanisms for specifying such structures as part of a system of BDI agents is ongoing work; this is based particularly on the approach to organisations presented in (21). There is also ongoing work on integrating our BDI agents with Semantic Web technologies, such as ontologies (40). This is important not only for Semantic Web applications, but also for Grid-based computing (18), as Grid infrastructures are currently drawing heavily on such technologies. In the medium term, we plan to have our platform fully operational for Grid-based social simulations.

One of our long term objectives is to introduce simulation mechanisms based on the conceptual reconciliation of cognition and emergence: this is particularly inspired by Castelfranchi's (14; 13) 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. In others words, we aim (in the long term) at providing the basic conditions for MAS-SOC to help 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 (15). The BDI architecture (34; 33) and logics (35; 39; 43), on which the agent language used here (and various others among the existing agent-oriented programming platforms (4)) is based, has the advantage, as a model for cognitive agents, as being the best known and most extensively studied one; further, it has in very recent years started to attract again considerable interest in the major agent conferences.

Although MAS-SOC is a fully functional simulation platform, which was first described in (9), not many experiments have as yet been conducted with it. A simple simulation to investigate social aspects of urban growth appeared in (23); it was completely developed in accordance with the various techniques which form the MAS-SOC approach to social simulation (Section 8 brings another sample simulation). Because there is little practical experience with the platform, we do not offer, in this paper, benchmarking results as was the case of other JASSS Forum articles. Instead, we introduce the various technical aspects of a platform which follows a paradigmatic change that we think will have a major impact in the future development of large-scale social simulations.

In the next section, we give an overview of the whole MAS-SOC approach to multi-agent simulations. In the following sections, we describe each of the technologies that form the MAS-SOC platform. Towards the end, we present a simple example which shows how these various technologies are used and combined.

## 2 Overview of MAS-SOC

One of the main goals of the MAS-SOC simulation platform (MAS-SOC stands for Multi-Agent Simulations for the SOCial Sciences) is to provide a framework for the creation of agent-based simulations which does not require too much experience in programming from users, yet allowing users to use state-of-the-art agent technologies. In particular, it should allow for the design and implementation of simulations with

*cognitive agents* — a plethora of platforms for reactive agents exists, but that is certainly not the case for more elaborate agents.

In our approach, agents' reasoning is specified in a much extended version of AgentSpeak(L) (32), as interpreted by *Jason*, an *Open Source* agent platform<sup>2</sup> based on Java (6). Of particular interest is that the extensions allow for speech-act based agent communication, and there is ongoing work to allow for ontologies as part of an AgentSpeak(L) agent's belief base.

The environments where agents are to be situated are specified in ELMS, a language we have designed for the description of multi-agent environments (29; 2; 30). The development of an environment description language for our simulation platform was needed because when a multi-agent system is a (completely) computational system (i.e., not situated in the real world), this is an important level of the engineering of multi-agent systems. However, this level of agent-oriented software engineering is not normally addressed in the literature, as environments are simply considered as "given". This occurs in particular in relation to cognitive agents<sup>3</sup>.

The interactions among the simulation components (agent-agent, agent-environment) are implemented with the SACI toolkit (21). The communication and agent management infrastructure available with SACI makes it easy to run as systems distributed over the Internet.

A graphical user interface is provided, which facilitates the specification of multi-agent environments, agents (their beliefs and plans), and multi-agent simulations; it also helps the management of libraries of these simulation components. From the information given by the user, the system generates source codes for the AgentSpeak(L) and ELMS interpreters, on which MAS-SOC is based. We now discuss each of the technologies associated with MAS-SOC in turn.

### 3 AgentSpeak(L)

The AgentSpeak(L) programming language was introduced by Rao in (32). The language was quite influential in the definition of other agent-oriented programming languages. AgentSpeak(L) is particularly interesting, in comparison to other agent-oriented languages, in that it retains the most important aspects of the BDI-based planning systems on which it was based. Its relation to BDI logics (35) has been thoroughly studied (7; 8) and a working interpreter for the language (3) has been developed based on its formal operational semantics (27; 8).

AgentSpeak(L) integrates notions of the BDI architecture and logic programming, providing an elegant abstract framework for programming BDI agents. The BDI architecture is, in its turn, the predominant approach to the implementation of "intelligent" or "rational" agents (43).

Figure 1 shows the abstract syntax of AgentSpeak(L). An AgentSpeak(L) agent is created by the specification of a set *bs* of base beliefs and a set *ps* of plans. The *initial set of beliefs* is just a collection of ground first order predicates. A plan is formed by

<sup>2</sup> Available at <http://jason.sourceforge.net>.

<sup>3</sup> For this reason, a workshop series (held with AAMAS) has recently been created for discussing issues related to multi-agent environments specifically (42).

$$\begin{aligned}
ag &::= bs \ ps \\
bs &::= at_1 \dots at_n && (n \geq 0) \\
at &::= P(t_1, \dots t_n) && (n \geq 0) \\
ps &::= p_1 \dots p_n && (n \geq 1) \\
p &::= te : ct \leftarrow h \\
te &::= +at \mid -at \mid +g \mid -g \\
ct &::= at \mid \neg at \mid ct \wedge ct \mid \top \\
h &::= a \mid g \mid u \mid h; h \\
g &::= !at \mid ?at \\
u &::= +at \mid -at
\end{aligned}$$

**Fig. 1.** Syntax of AgentSpeak(L).

a *triggering event* (denoting the purpose for that plan), followed by a conjunction of belief literals representing a *context*. The context must be a logical consequence of that agent's current beliefs for the plan to be *applicable*. The remainder of the plan is a sequence of basic actions or (sub)goals that the agent has to achieve (or test) when the plan, if applicable, is chosen for execution.

AgentSpeak(L) distinguishes two types of goals: *achievement goals* and *test goals*. Achievement and test goals are predicates (as for beliefs) prefixed with operators '!' and '?' respectively. Achievement goals state that the agent wants to achieve a state of the world where the associated predicate is true. (In practice, these initiate the execution of *subplans*.) A *test goal* returns a unification for the associated predicate with one of the agent's beliefs; it fails otherwise. A *triggering event* defines which events may initiate the execution of a plan. An *event* can be internal, when a subgoal needs to be achieved, or external, when generated from belief updates as a result of perceiving the environment. There are two types of triggering events: those related to the *addition* ('+') and *deletion* ('-') of mental attitudes (beliefs or goals).

Plans refer to the *basic actions* (represented by the metavariable *a* in the grammar above) that an agent is able to perform on its environment. Such actions are also defined as first-order predicates, but with special predicate symbols (called action symbols) used to distinguish them from other predicates.

```

+concert(A,V) : likes(A)
  ← !book_tickets(A,V)

+!book_tickets(A,V) : ¬busy(phone)
  ← call(V);
  ...;
  !choose_seats(A,V)

```

**Fig. 2.** Examples of AgentSpeak(L) plans.

Figure 2 shows some examples of AgentSpeak(L) plans. They tell us that, when a concert is announced for artist  $A$  at venue  $V$  (so that, from perception of the environment, a belief  $\text{concert}(A, V)$  is *added*), then if this agent in fact likes artist  $A$ , then it will have the new goal of booking tickets for that concert. The second plan tells us that whenever this agent adopts the goal of booking tickets for  $A$ 's performance at  $V$ , if it is the case that the telephone is not busy, then it can execute a plan consisting of performing the basic action  $\text{call}(V)$  (assuming that making a phone call is an atomic action that the agent can perform) followed by a certain protocol for booking tickets (indicated by '...'), which in this case ends with the execution of a plan for choosing the seats for such performance at that particular venue.

For AgentSpeak(L) to be useful in practice, various extensions to it have been proposed (28; 1; 3). In particular, the formal semantics for speech act-based communication primitives given in (28) formed the basis for the implementation of the *open source* interpreter *Jason* (6).

Through speech act-based communication, an agent can share its internal state (beliefs, desires, intentions) with other agents, as well as it can influence other agents' states. Speech-act based communication for AgentSpeak(L) agents has already been used, in a very simple way, so as to allow model checking (10; 5).

Despite the considerable improvement that has been achieved since the paradigm was first thought out (38), agent-oriented programming languages are still in their early stages of development. Industrial-strength applications usually require massive use of data, thus a belief base that is simply an unstructured collection of ground predicates is just not good enough.

Another shortfall of AgentSpeak(L), that reduces its applicability for the development of semantic web multi-agent systems, is the absence of mechanisms to indicate the ontologies that have to be considered by agents in their reasoning. This shortfall imposes the assumption (adopted in (28), for instance) that all communicating AgentSpeak(L) agents in a multi-agent application have a common understanding about terms that are used in the content of exchanged messages. This is clearly unrealistic for semantic web applications as they typically require the integration of multiple ontologies about different domains.

In order to overcome these limitations in the language we plan to improve the semantics of AgentSpeak(L) by first replacing the subset of predicate logic that is currently its subjacent logic, by more expressive Description Logics such as those of OWL DL and OWL Lite (20). The idea is to allow AgentSpeak(L) agents to reason about ontologies written in OWL. This way, applications written in AgentSpeak(L) can be deployed on the Web and interoperate with other semantic web applications based on OWL, a W3C standard.

A significant proportion of our planned future work in this area is to integrate agent-oriented programming and semantic web technologies. We believe that AgentSpeak(L) is a suitable framework to conduct this kind of research mainly for two reasons: it has formal operational semantics and there is an interpreter for AgentSpeak(L) that implements that semantics. Next, we discussed *Jason*

## 4 Jason

*Jason* is the interpreter for our extended version of AgentSpeak(L), which allows agents to be distributed over the net through the use of SACI (21). *Jason* is available *Open Source* under GNU LGPL at <http://jason.sourceforge.net> (6).

One of the important characteristics of *Jason* is that, as we mentioned above, it implements the operational semantics of an extension of AgentSpeak. Having formal semantics also allowed us to give precise definitions for practical notions of beliefs, desires, and intentions in relation to running AgentSpeak agents, which in turn underlies the work on formal verification of AgentSpeak programs. The formal semantics, using structural operational semantics (31) (a widely-used notation for giving semantics to programming languages) was given then improved and extended in a series of papers (27; 7; 8; 28; 41). However, for space limitation, we are not able to include a formal account of the semantics of AgentSpeak here, so we will just provide the main intuitions behind the interpretation of AgentSpeak programs.

Besides the belief base and the plan library, the AgentSpeak interpreter also manages a set of *events* and a set of *intentions*, and its functioning requires three *selection functions*. The event selection function ( $S_E$ ) selects a single event from the set of events; another selection function ( $S_O$ ) selects an “option” (i.e., an applicable plan) from a set of applicable plans; and a third selection function ( $S_I$ ) selects one particular intention from the set of intentions. The selection functions are supposed to be agent-specific, in the sense that they should make selections based on an agent’s characteristics (though previous work on AgentSpeak(L) did not elaborate on how designers specify such functions. Therefore, we here leave the selection functions undefined, however the choices made by them are supposed to be non-deterministic.

*Intentions* are particular courses of actions to which an agent has committed in order to handle certain events. Each intention is a stack of partially instantiated plans. *Events*, which may start off the execution of plans that have relevant triggering events, can be *external*, when originating from perception of the agent’s environment (i.e., addition and deletion of beliefs based on perception are external events); or *internal*, when generated from the agent’s own execution of a plan (i.e., a subgoal in a plan generates an event of type “addition of achievement goal”). In the latter case, the event is accompanied with the intention which generated it (as the plan chosen for that event will be pushed on top of that intention). External events create new intentions, representing separate focuses of attention for the agent’s acting on the environment. A provision exists for internal events corresponding to plan controlled update of beliefs, necessary to support limited form of plan controlled register of deductions performed.

We next give some more details on the functioning of an AgentSpeak interpreter, which is clearly depicted in Figure 3 (reproduced from (26)). In the figure, sets (of beliefs, events, plans, and intentions) are represented as rectangles. Diamonds represent selection (of one element from a set). Circles represent some of the processing involved in the interpretation of AgentSpeak programs.

At every interpretation cycle of an agent program, the interpreter updates a list of events, which may be generated from perception of the environment, or from the execution of intentions (when subgoals are specified in the body of plans). Beliefs are either updated from perception or from plan controlled operations (the latter is not shown in

3) and whenever there are changes in the agent's beliefs, this implies the insertion of an event in the set of events. This belief revision function is also supposed to be agent specific.

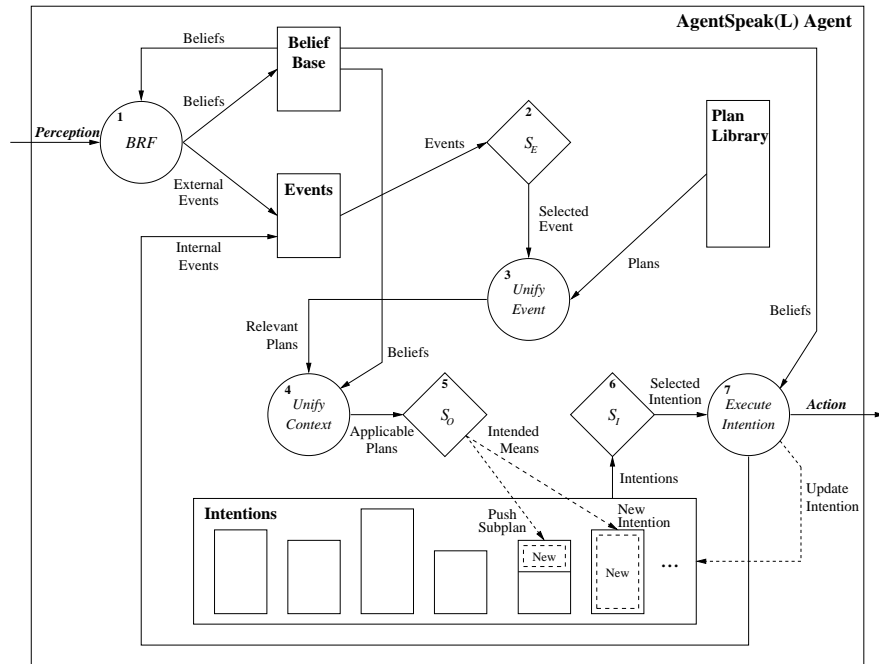


Fig. 3. An Interpretation Cycle of an AgentSpeak Program (26).

After  $S_E$  has selected an event, the interpreter has to unify that event with triggering events in the heads of plans. This generates a set of all *relevant plans*. By checking whether the context part of the plans in that set follow from the agent's beliefs, the interpreter determines a set of *applicable plans* (plans that can actually be used at that moment for handling the chosen event). Then  $S_O$  chooses a single applicable plan from that set, which becomes the *intended plan* for handling that event, and either pushes that plan on the top of an existing intention (if the event was an internal one), or creates a new intention in the set of intentions (if the event was external, i.e., generated from a perception of the environment).

All that remains to be done at this stage is to select a single intention to be executed in that cycle. The  $S_I$  function selects one of the agent's intentions (i.e., one of the independent stacks of partially instantiated plans within the set of intentions). On the top of that stack there is a plan, and the formula in the beginning of its body is taken for execution. This implies that either a basic action is performed by the agent on its environment, an internal event is generated (in the case that the selected formula is an

achievement goal), or a test goal is performed (which means that the set of beliefs has to be checked).

If the intention is to perform a basic action or a test goal, the set of intentions needs to be updated. In the case of a test goal, the belief base will be searched for a belief atom that unifies with the predicate in the test goal. If that search succeeds, further variable instantiation will occur in the partially instantiated plan which contained that test goal (and the test goal itself is removed from the intention from which it was taken). In the case where a basic action is selected, the necessary updating of the set of intentions is simply to remove that action from the intention (the interpreter informs to the agent effectors what action is required). When all formulæ in the body of a plan have been removed (i.e., have been executed), the whole plan is removed from the intention, and so is the achievement goal that generated it (if that was the case). This ends a cycle of execution, and everything is repeated all over again, initially checking the state of the environment after agents have acted upon it, then generating the relevant events, and so forth.

Some other features available in *Jason* are:

- speech-act based inter-agent communication (and belief annotation on information sources). The implemented illocutionary forces are: `tell`, `untell`, `achieve`, `unachieve`, `tellHow`, `untellHow`, `askIf`, `askOne`, `askAll`, and `askHow`;
- annotations on plan labels, which can be used by elaborate (e.g., decision theoretic) selection functions;
- the possibility to run a multi-agent system distributed over a network (using SACI);
- fully customisable (in Java) selection functions, trust functions, and overall agent structure (perception, belief-revision, inter-agent communication, and acting);
- straightforward extensibility (and use of legacy code) by means of user-defined “internal actions”;
- a clear notion of a *multi-agent environment*, which is implemented in ELMS (this can be a simulation of a real environment, e.g. for testing purposes before the system is actually deployed);

*Jason* is distributed with an Integrated Development Environment (IDE) which provides a GUI for editing AgentSpeak code for the individual agents (see Figure 4). Through the IDE, it is also possible to control the execution of a MAS. There is another tool provided as part of the IDE which allows the user to inspect the agents’ internal states when the system is running in debugging mode (see Figure 5). This is very useful for debugging MAS, as it allows “inspection of agents’ minds” across a distributed system.

Interestingly, most of the advanced features are available as optional, customisable mechanisms. Thus, because the AgentSpeak(L) core that is interpreted by *Jason* is very simple and elegant, yet having all the main elements for expressing reactive planning system with BDI notions, we think that *Jason* is also ideal for teaching AOP for under- and post-graduate studies.

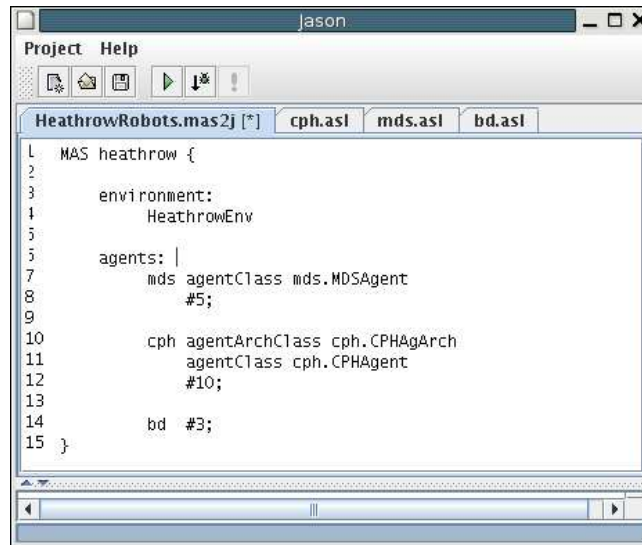


Fig. 4. Jason IDE.

## 5 The ELMS Language

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 simulations. The language is called ELMS (Environment Description Language for Multi-Agent Simulation).

Besides the basic environment properties and objects, the language provides the means for the specification of the “physical” part of simulated agents, which we refer to as the “bodies” of the agents, as well as the various kinds of physical interactions, through actions and perceptions that may happen between agents and objects, including other agents in the environment.

We have developed a prototype of an interpreter for an environment definition language, presented in detail in (29) and also in (2; 30).

### 5.1 Modelling Environments with ELMS

An environment description is a specification of the properties and behaviour of the environment objects. 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

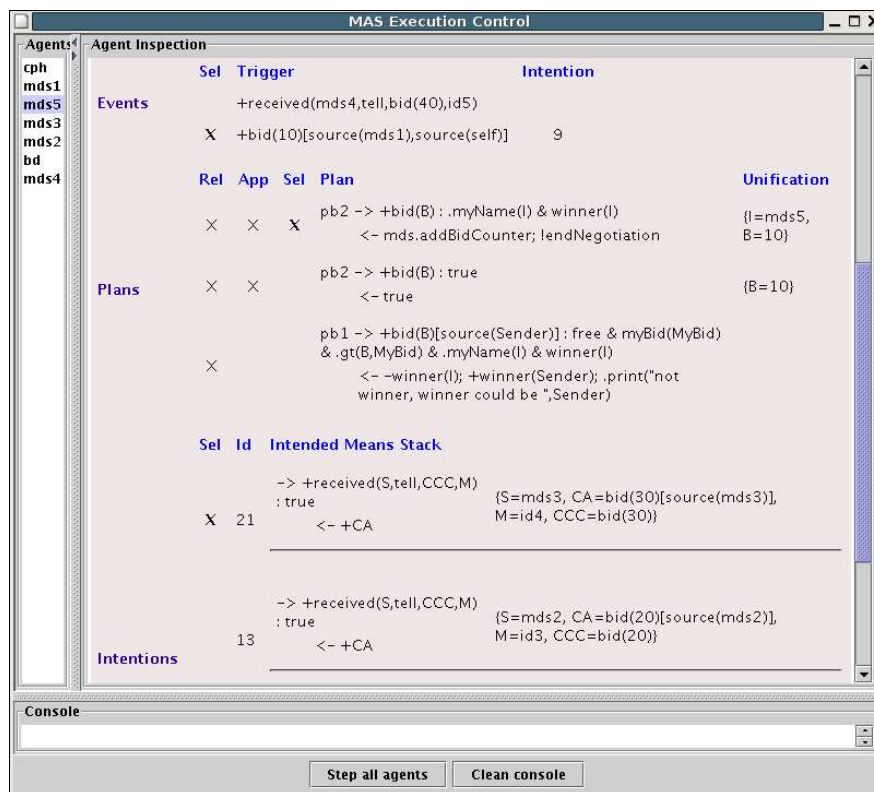


Fig. 5. Jason “Mind Inspector” Tool.

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 the perceptible aspects of their bodies, 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, agent bodies, 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.

## 5.2 Language Constructs in ELMS

The ELMS language uses an XML syntax, which can be somewhat cumbersome to be handled 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.

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 within that environment. A specification of an agent-body class contains its name, a list of attributes, a list of actions, a list of perception types, and an optional “constructor” procedure that is executed when the “agent body” instance is created. 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).

The code sample below<sup>4</sup> define an agent-body class named `consumer` which has five integer attributes, one of them initialised with a random value from 60 to 100. Agents defined in this class are able to execute three types of perception, which will be explained later, and six types of actions. The constructor bellow places the agent in a random cell where the preconditions are true, the cell must not be a `shop` and may not have a `owner` already. After the placement, some values are set so that the agent is defined as the owner of that cell.

```
<AGENT_BODY NAME="consumer">
  <INTEGER NAME = "money"> 1000 </INTEGER>
  <INTEGER NAME = "id"> "SELF" </INTEGER>
  <INTEGER NAME = "home_x">0 </INTEGER>
  <INTEGER NAME = "home_y">0</INTEGER>
  <INTEGER NAME = "stamina">
    <RANDOM MIN = "60" MAX = "100"/>
  </INTEGER>
  <PERCEPTIONS>
    <ITEM NAME = "see_products"/>
    <ITEM NAME = "self_info"/>
    <ITEM NAME = "verify_products"/>
  </PERCEPTIONS>
  <ACTIONS>
    <ITEM NAME = "walk_right"/>
    <ITEM NAME = "walk_left"/>
    <ITEM NAME = "walk_down"/>
    <ITEM NAME = "walk_up"/>
    <ITEM NAME = "buy"/>
    <ITEM NAME = "rest"/>
  </ACTIONS>
  <CONSTRUCTOR>
    <IN_RAND>
      <PRECONDITION>
        <EQUAL>
          <OPERAND>
            <CELL_ATT ATTRIBUTE = "shop">
              <X>"X"</X>
            </OPERAND>
          </OPERAND>
        </EQUAL>
      </PRECONDITION>
    </IN_RAND>
  </CONSTRUCTOR>
</AGENT_BODY>
```

---

<sup>4</sup> Part of the environment definition from the example shown in Section 8.

```

        <Y>"Y"</Y>
      </CELL_ATT>
    </OPERAND>
  <OPERAND> "FALSE" </OPERAND>
</EQUAL>
<EQUAL>
  <OPERAND>
    <CELL_ATT ATTRIBUTE = "owner">
      <X>"X"</X>
      <Y>"Y"</Y>
    </CELL_ATT>
  </OPERAND>
  <OPERAND> -1 </OPERAND>
</EQUAL>
</PRECONDITION>
<ELEMENT NAME = "SELFCLASS">
  <INDEX>"SELF"</INDEX>
</ELEMENT>
</IN_RANGE>
<ASSIGN>
  <ELEMENT_ATT NAME = "SELFCLASS" ATTRIBUTE = "home_x" >
    <INDEX> "SELF" </INDEX>
  </ELEMENT_ATT>
  <EXPRESSION> "X"</EXPRESSION>
</ASSIGN>
<ASSIGN>
  <ELEMENT_ATT NAME = "SELFCLASS" ATTRIBUTE = "home_y" >
    <INDEX> "SELF" </INDEX>
  </ELEMENT_ATT>
  <EXPRESSION> "Y"</EXPRESSION>
</ASSIGN>
<ASSIGN>
  <CELL_ATT ATTRIBUTE = "owner" >
    <X>+0</X>
    <Y>+0</Y>
  </CELL_ATT>
  <EXPRESSION> "SELF"</EXPRESSION>
</ASSIGN>
</CONSTRUCTOR>
</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 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.

**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, chang-

ing 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 the agents' internal reasoning.

```
<ACTION NAME="rest">
  <PRECONDITION>
    <EQUAL>
      <OPERATOR>
        <CELL_ATT ATTRIBUTE = "owner" >
          <X> +0</X>
          <Y> +0</Y>
          <Z> +0</Z>
        </CELL_ATT>
      </OPERATOR>
      <OPERATOR>
        <ELEMENT_ATT NAME = "SELFCLASS" ATTRIBUTE = "id" >
          <INDEX> "SELF" </INDEX>
        </ELEMENT_ATT>
      </OPERATOR>
    </EQUAL>
  </PRECONDITION>
  <ASSIGN>
    <ELEMENT_ATT NAME = "consumer" ATTRIBUTE = "stamina" >
      <INDEX> "SELF" </INDEX>
    </ELEMENT_ATT>
    <EXPRESSION>
      <RANDOM MIN = "60" MAX = "100"/>
    </EXPRESSION>
  </ASSIGN>
</ACTION>
```

In the example above, an action named `rest` is defined. It has no parameters, but the agent must be on his home cell as defined as a precondition for the action be executed, otherwise the action will fail. If the precondition is satisfied the `stamina` attribute will receive a random value between 60 and 100 units.

## 2. Defining the Environment Objects and Space:

**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<sup>5</sup>. The code below exemplifies a definition of a two-dimensional grid that has ten columns and ten lines, where each cell has an integer that represents its owner (where `-1` means no owner) and a boolean variable that keeps the information about whether the cell is a shop or not. In this example, grid cells do not have reactions.

```
<DEFGRID SIZEX="10" SIZEY="10" SIZEZ="1">
  <BOOLEAN NAME = "shop"> "FALSE" </BOOLEAN>
  <INTEGER NAME = "owner"> -1 </INTEGER>
</DEFGRID>
```

<sup>5</sup> 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.

**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. The resource definition is quite similar to the agent definition, the main difference being that the resources have reactions instead of actions.

**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).

There are some operational aspects of the simulations defined through other language constructs not mentioned here. To more details see (30; 29).

### 5.3 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<sup>6</sup>;
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;

---

<sup>6</sup> Agents send a message with “true” as its content if they chose not to execute an action in that cycle.

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.

#### 5.4 Some Remarks on ELMS

In (37), Russel and Norvig define 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.

**Fully observable vs. partially observable:** 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 fully or partially observable is a designer’s decision.

**Deterministic vs. stochastic:** As ELMS environments can be only partially observable, 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 stochastic.

**Episodic vs. sequential:** 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 what 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): partially observable, stochastic, sequential, 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).

There are many possibilities to evolve the work of environmental description with the ELMS language. We are still developing it through the aggregation of more complex language constructs, to ease the description of environments. Another possibility on development, is the replacement of the input language by a restricted set from the OWL language, what will open many other possibilities to use the structure that we developed for the environment description and execution, as will be presented on Section 6.

Although the ELMS interpreter was been tailored for social simulation implemented according to the MAS-SOC approach, it could potentially be useful for other applications as well.

## 6 Ontologies

An extension of our work above is a top level ontology for specifying environments, through which a project-level, executable definition of a multi-agent environment can be derived. Ontologies are known to be useful in many communication situations: between people, between people and systems, and also between systems themselves. In this work, the use of environment ontologies adds three important features to the existing platform/approach.

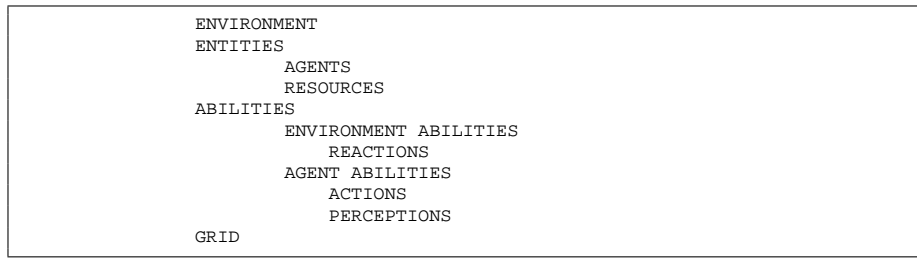
First, ontologies provide a common vocabulary with which simulation developers can specify environments. The developing of social simulations usually depends on the work of a team with people from different areas, as social scientists and computer scientists. It is possible that those developing the agents are not the same working on the environments, interactions, and organisational schema. In such type of work small misunderstandings may compromise the whole process. An environment ontology presents a consensual model for an environment and agents; the essential properties of the environment and agents' actions and percepts must apply in all situations as defined by the semantic relationships between the elements of the environment, which will all be explicitly represented in the ontology. For this reason, we think that defining environments through ontologies aids the development of the simulation.

Second, an environment ontology is useful for agents acting in the environment because it provides a common vocabulary for communication in and about the environment. Such explicit conceptualisation is also essential to allow interoperability of intelligent heterogeneous systems.

Third, environment ontologies are defined through an ontology editor with a graphical user interface. This makes it easier for those unfamiliar with programming to understand and even design such ontologies.

### 6.1 An Ontology for Environments in Multi-Agent Systems

We have defined a top-level ontology for environments that limits the possible constructs in an environment definition. The hierarchy of concepts related to our view of multi-agent environments can be seen in Figure 6. Table 1 shows the properties of the main concepts, and explains them briefly.



**Fig. 6.** Top-Ontology

Concept	Concept Properties	Property Description
Grid	Has_Properties Do_Reactions Size_X, Size_Y, Size_Z	a list of properties a list of cell reactions grid dimensions
Actions	Has_Conditions Has_Commands	a list of preconditions a sequence of commands
Reactions	Has_Conditions Has_Commands	a list of preconditions a sequence of commands
Perceptions	Has_Conditions Has_Properties	a list of preconditions a list of properties
Agent	Do_Actions Do_Perceptions	a list of actions a list of perceptions
Resource	Do_Reactions	a list of reactions

**Table 1.** Properties of Environment Concepts

Using an ontology editing tool (e.g., Protege<sup>7</sup>), programmers can define their project environment on the basis of the predefined top ontology. Generating the project ontology can avoid various misunderstandings in all development stages from problem analysis to the specification that will be executed. For the agent programmers it will also be clearer how the environment works, which actions each agent is able to perform, and what it will receive as percepts. The programmer does not have to handle code directly, which usually has a semantics that is subjective. Consequently, the ontology also facilitates the understanding of the simulation and the validation of environment design. This editor generates an ontology representation in various formats; we work on the basis of the W3C standard, the Ontology Web Language (OWL<sup>8</sup>).

<sup>7</sup> <http://protege.stanford.edu>.

<sup>8</sup> <http://www.w3.org/TR/owl-features/>.

## 6.2 Deriving an Environment Ontology

We present an example of an environment ontology to illustrate our approach. The environment has shops and consumer's houses; the agents are shop owners, shopkeepers, and consumers; resources are products.

To implement this example, the designer derives the `Agent` class, creating the subclasses of agents (`Shopkeeper`) and (`Consumer`); derive the `Resource` class creating classes such as `Products` instantiate the necessary resources; derive the `Action`, `Perception` and `Reaction` classes likewise as seen in the Figure 7. These are general steps to define an environment.

```
ENTITIES
  AGENTS
    SHOPKEEPER
      Do_Actions = change_price, create_product, change_sellQuality
      Do_Perceptions = seeProducts, self_info
      Has_Properties
        id (integer)
        money (integer)

    CONSUMER
      Do_Actions = walk_right, walk_left, walk_up, walk_down, buy, rest
      Do_Perceptions = see_product, verify_product, self_info
      Has_Properties
        stamina (integer)
        id (integer)
        money (integer)
        home_x (integer)
        home_y (integer)
        stamina (integer)

  RESOURCES
    PRODUCTS
      Do_Reactions = (none)
      Has_Properties
        id (integer)
        type (integer)
        retailer (integer)
        owner (integer)
        price (integer)
        created (integer)
        sellQuality (integer)
        realQuality (integer)
```

**Fig. 7.** Sample Ontology

The example in the figure shows the classes (in uppercase) and some of the properties of the classes derived from the top ontology to define the sample environment. The OWL ontology with the specific (derived) classes contains the information that is required to the ELMS interpreter for the generation of the environment controller process used in the simulation.

## 7 MAS-SOC Graphical User Interface

A graphical user interface which facilitates the creation and running of simulations is being developed. This interface gives access to what we call the MAS-SOC *manager*. Besides facilitating the creation of simulations, the MAS-SOC manager integrates the various technologies used in our approach, such as the ELMS interpreter, the AgentSpeak(L) interpreter (*Jason*, see section 4), and the SACI infrastructure.

The first aspect of the MAS-SOC manager that should be mentioned is related to the creation of libraries of plans, agents and environment (maintained in separate files), which facilitates the reuse of those definitions in different simulations. The MAS-SOC manager allows the creation and edition of each of these libraries. A file containing a plan library consists of a series of plan definitions in the AgentSpeak(L) syntax. In an agent library, each agent<sup>9</sup> is represented by a name, a set of AgentSpeak(L) base beliefs (the initial beliefs that agents of this type will have when the simulation begins), and a list of pointers to plans (in fact, plan labels) in specific plan libraries. With this, the AgentSpeak(L) source codes for the agents can be generated by the interface and sent to the running instances of the AgentSpeak(L) interpreter. The information for an environment definition is also prompted from the graphical interface, and the MAS-SOC manager automatically generates the XML-based ELMS source code, which is sent to the ELMS interpreter.

Figure 8 gives a flavour of the MAS-SOC user interface. It has the feel of a “workspace”, where one can create and edit libraries of plans, agents, and environments, which can then be used in defining a multi-agent simulation. Plans and agents follow the straightforward syntax of AgentSpeak(L), and all necessary information for an ELMS environment description is prompted via a form-like interface. Other features of the MAS-SOC manager are the creation, execution and monitoring of the simulation, which we are still improving. This part of the platform provides the integration of the several systems forming the MAS-SOC approach.

In a simulation definition window of the user interface, the user determines the set of individual agents and the particular environment that are intended for a given simulation. From the environment definition, the MAS-SOC manager checks which types of agents can participate in the simulation, and allows the user to choose, for each of those types, the number of instances of individual agents that will be created (by means of the SACI toolkit). Each of these agents runs an AgentSpeak(L) interpreter with the source code generated by the MAS-SOC manager. After the user has informed the chosen environment and the instances of agents, the simulation can be started off. The execution of agents can be aborted and new ones can be created through the MAS-SOC manager.

An execution window then provides the information about the components of a simulation (agents and resources) which are active in a simulation. There are in fact two levels of information about agents: their internal and external states. The internal state gives information on an agent’s mental attitudes (e.g., its present beliefs and intentions) at each simulation step, while its external state is related to the characteristics (proper-

---

<sup>9</sup> In fact, this refers to *types* of agents, as each of these agent definitions may have various instances in a simulation.

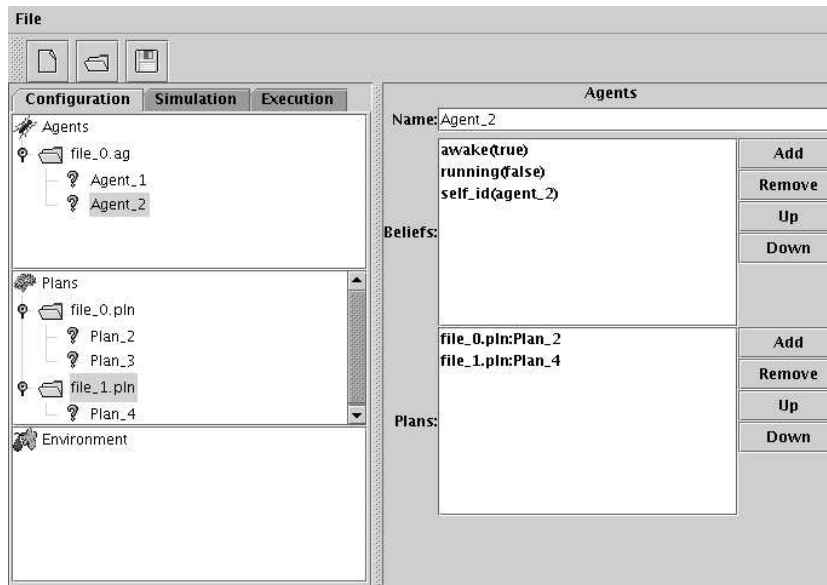


Fig. 8. The MAS-SOC Manager GUI.

ties) of the agent that are perceptible to other agents through the environment. These two levels of information on agents can be accessed separately from the execution window. For the resources, one can observe the current values associated with their properties (attributes).

## 8 Example

This is a simple example of a simulation that was built using our approach of multi-agent simulation. It will be presented briefly; see (11) for more details and some discussion on the design choices of the simulation.

The simulation is composed by consumer agents, shopkeepers, and the environment where they interact. The objective of the consumers is to buy everything in their wish-list, paying a good price for good products. The objective of the shopkeepers is to increase their profit, selling their products and sometimes cheating consumers.

### 8.1 The Environment

The environment, as mentioned in section 5.2, is represented by a  $10 \times 10$  grid, where each cell has a boolean attribute `shop` and an integer attribute `owner` which indicates the agent owner of the place. The *shop cells* are uniformly distributed in the environment, while the *consumers' houses* are randomly distributed in the free cells. For each shop there is a *shopkeeper* agent who offers for sale the products available in that cell.

When a consumer agent enters a shop its perceptions will allow him to obtain information about all the products available in the shop. The perceptions include the product number, type, price, quality (we refer to it as `sellQuality`, defined by the *shopkeeper*, which can be different from the product's actual quality).

The *consumer's house* is a cell owned by consumer agent which has the `shop` attribute set as `false`. The products bought by the consumers are stored on each consumer's house. When an agent enters on a *consumer's house* it will receive the complete information about the products present in that cell. This perception is composed of the same information about the products in the shop, but now it includes the product real quality and product retailer (*shopkeeper ID*). With this information the consumer agent will be able to evaluate which *shopkeepers* cheat and which do not.

Each *consumer* agent has an amount of *stamina* (energy) used to walk through the grid looking for the products that it wants. For each cell move, the *stamina* is automatically decreased by 3 units (this is controlled by the environment).

There is a class of resources named *products* that represents the items that a *shopkeeper* has to sell or that a *consumer* has bought. The definition of this resource is in the code sample below.

```
<RESOURCE NAME="product">
  <INTEGER NAME = "id"> "SELF" </INTEGER>
  <INTEGER NAME = "type"> 0 </INTEGER>
  <INTEGER NAME = "retailer">-1</INTEGER>
  <INTEGER NAME = "owner">-1</INTEGER>
  <INTEGER NAME = "price"> 0 </INTEGER>
  <INTEGER NAME = "quality"> 0 </INTEGER>
  <INTEGER NAME = "realquality"> 0 </INTEGER>
  <INTEGER NAME = "created"> 0 </INTEGER>
</RESOURCE>
```

## 8.2 The Consumer agent

The consumer agent has as initial belief a list of products that it wants to buy, including information such as the expected quality and price of the products. During the simulation the agent accumulates information about where the products are available, the price and quality of desired products, and the level of confidence in each *shopkeeper*. When the consumer enters a shop, with the information received through perceptions, it will store the information about the products that are in the list of the consumer's desired products. When the consumer enters a house, it will receive the information about the products that the cell owner had bought; with this information the agent will evaluate the *shopkeepers*.

The consumer will walk around the grid examining products in shops until its *stamina* level reaches some threshold, and then the agent will start buying the products that fulfilled the expectations. The agent will go to the shops where he saw the product with the best cost and quality, and will buy the product if the product is still available. When the consumer's *stamina* level reaches another threshold level, the agent will go home to rest. At the home cell, the consumer executes the `rest` action to have its *stamina* level restored. After buying all the items in the list, the agent will return to its home cell.

In this simulation, we defined three types of agents:

**Consumer A:** this type of consumer has the lower *stamina* levels between 60 and 70.

This means that this agent will do a shorter search for the products and prices.

**Consumer B:** this type of consumer has a middle *stamina* level, between 75 and 85.

**Consumer C:** this type of consumer has the higher *stamina* level between 90 and 100.

Agents of this type will do the longest search for the products and prices.

As mentioned before, the agents reasoning is defined using the AgentSpeak(L) language. The following plan is executed when the consumer agent finds a product bought with a good price and quality. It is used to raise the evaluation of the *shopkeeper* who sold the product.

```
+!refreshBeliefs (NUM)
: product(instance(INS),id(ID)) &
  product(instance(INS),realquality(RQ)) &
  product(instance(INS),sellquality(SQ)) &
  RQ >= SQ
  <- ?product(instance(INS),retailer(IDR));
    ?sellerEvaluation(IDR,Ev);
    .addB(Ev,X,NewEv);
    -sellerConcept(IDR,Ev);
    +sellerConcept(IDR,NewEv).
```

### 8.3 The Shopkeeper agent

The *shopkeeper* agent has to offer products in its shop. For each product that it creates, there is a cost proportional to the *realQuality* of the created product. The *shopkeeper* may choose to use any value as the *sellQuality* that will be presented to the consumer. If there is a big difference between the *sellQuality* and the *realQuality*, the consumers will start avoiding to buy from this *shopkeeper*. The *shopkeeper* may change the prices and the *sellQuality* of the products along the simulations, if it chooses to do so. As its products are sold, the agent needs to create new products to replace the ones that have been sold.

We defined two types of shop-keeping agents:

**Shopkeeper X:** this type of *shopkeeper* will sell its product with very little difference between the *realQuality* and the *sellQuality*. This means that this agent will not cheat the consumers, or will cheat seldom.

**Shopkeeper Y:** this type of *shopkeeper* will try to cheat the consumers unless it does not sell any good during a number of cycles.

### 8.4 Simulation Results

For this simulation, 30 *shopkeeper* agents were instantiated and uniformly distributed in a  $10 \times 10$  grid, and 15 *consumers* were randomly distributed on the free cells. The *shopkeepers* were divided equally, 15 of each type, and there were 5 consumers of each type. Each consumer had 10 items in the list of items they wanted to buy. The simulation was run for 200 cycles.

The performance of the consumer agent were measured by the following formula:  $P = \frac{Total\_cycles}{Last\_purch} + \sum(\frac{realQuality}{sellQuality})$ ; where *Last\_purch* is the cycle number when the last purchase was made by the agent; this is done, of course, for each of the products

bought. This means that the agent performance decreases as the simulation progresses (if no further purchases are made), and also decreases if the agent has been cheated by the *shopkeepers*.

The performance of the *shopkeeper* agent is measured by the formula:  $P = Final\_money + \sum(product\_cost)$ . This means that the *shopkeeper*'s performance is equal to the sum of all money it has accumulated plus the sum of the *cost* of the unsold products.

Using those formulæ, the following results were obtained. The Consumer A agents had the worst performance: they had as average 3.75 points. Consumers B had the best performance with an average of 5.56 points (48% more than consumers A) and consumers C had an average of 3.90 points. This can be explained by the fact that the consumer A agents have not searched sufficiently to find good products with good prices. Since they were the first to buy, they were cheated by the *shopkeepers*. The consumer B agents, used the experience of agents A to recognise the cheaters, and had the best performances. The consumer C agents did too much searching, and when they decided to buy, another agent might have already bought the products they wanted to buy, so they had to go after the second or third choice.

On the shopkeeper's side, shopkeepers X had as average 1934 money units, while shopkeepers Y had 1544 money units, a difference of almost 26%. The shopkeeper Y agents (the ones that tried to cheat consumers) had the worse performance, because they sold only a few products, mostly for the consumer A agents.

This example was aimed only at illustrating the use of the MAS-SOC approach. The agents' reasoning was defined using the AgentSpeak(L) language, which may be executed using the *Jason* interpreter, and the environment was defined using the ELMS language.

## 9 Final remarks

The strength of the MAS-SOC platform is its fully-fledged implementation of the operational semantics of AgentSpeak(L). Its current weakness is the lack of built-in *interaction frameworks* (protocols) which would spare the programmer to manually define, implement, and check the correctness of the forms of interactions between agents. Supporting ontologies for such interaction frameworks is also currently missing, but there is ongoing research on that subject. Introducing such indispensable elements is our current plan regarding the immediate development of the platform.

Concerning the MAS-SOC approach to social simulation, *social organisations* will be our immediate concern. We plan to adopt a role-based approach to social organisation (17; 22), where roles, norms, and values, together with negotiation protocols and strategies for solving conflicts, play a central role. That will demand the definition of which built-in *social frameworks* will be introduced in the platform to support such notions.

Work on such social frameworks is already under way, focusing on the notions of exchange values (19) and social control (12). Whatever final social elements we choose to built into the platform, we will adopt the idea that social structures can be *reified* as

concrete structures of a *general environment*, encompassing both the *physical* and the *social* structures that the agents have to deal with.

This stance allows us to introduce the above mentioned social elements (norms, roles, values, etc.) as standard constructs of the ELMS language, on equal footing with the physical elements that the language already handles. We hope that, following this track, we will be able both to refine the sound social simulation foundations underlying our approach and to further elaborate the agent-programming platform that supports MAS-SOC.

## References

1. Davide Ancona, Viviana Mascardi, Jomi F. Hübner, and Rafael H. Bordini. Co-AgentSpeak: Cooperation in AgentSpeak through plan exchange. In Nicholas R. Jennings, Carles Sierra, Liz Sonenberg, and Milind Tambe, editors, *Proceedings of the Third International Joint Conference on Autonomous Agents and Multi-Agent Systems (AAMAS-2004)*, New York, NY, 19–23 July, pages 698–705, New York, NY, 2004. ACM Press.
2. Rafael Bordini, Fabio Y. Okuyama, Denise de Oliveira, Guilherme Drehmer, and Romulo C. Krafta. The mas-soc approach to multi-agent based simulation. In Gabriela Lindemann, Daniel Moldt, and Mario Paolucci, editors, *First International Workshop on Regulated Agent-Based Social Systems (RASTA 2002)*, held with AAMAS02, Bologna, Italy, 16 July (Revised Selected and Invited Papers), number 2934 in LNAI, Berlin, 2004. Springer-Verlag.
3. Rafael H. Bordini, Ana L. C. Bazzan, Rafael O. Jannone, Daniel M. Basso, Rosa M. Vicari, and Victor R. Lesser. AgentSpeak(XL): Efficient intention selection in BDI agents via decision-theoretic task scheduling. In Cristiano Castelfranchi and W. Lewis Johnson, editors, *Proceedings of the First International Joint Conference on Autonomous Agents and Multi-Agent Systems (AAMAS-2002)*, 15–19 July, Bologna, Italy, pages 1294–1302, New York, NY, 2002. ACM Press.
4. Rafael H. Bordini, Mehdi Dastani, Jürgen Dix, and Amal El Fallah-Seghrouchni, editors. *Proceedings of the Second International Workshop on “Programming Multi-Agent Systems: Languages and Tools” (ProMAS 2004)*, held with the Third International Joint Conference on Autonomous Agents and Multi-Agent Systems (AAMAS 2004), New York City, NY, 20th of July, 2004. To appear in Springer’s LNAI Series.
5. Rafael H. Bordini, Michael Fisher, Carmen Pardavila, and Michael Wooldridge. Model checking AgentSpeak. In Jeffrey S. Rosenschein, Tuomas Sandholm, Michael Wooldridge, and Makoto Yokoo, editors, *Proceedings of the Second International Joint Conference on Autonomous Agents and Multi-Agent Systems (AAMAS-2003)*, Melbourne, Australia, 14–18 July, pages 409–416, New York, NY, 2003. ACM Press.
6. Rafael H. Bordini, Jomi F. Hübner, et al. **Jason**: A Java-based AgentSpeak interpreter used with SACI for multi-agent distribution over the net, manual, first release edition, Jan 2004. <http://jason.sourceforge.net/>.
7. Rafael H. Bordini and Álvaro F. Moreira. Proving the asymmetry thesis principles for a BDI agent-oriented programming language. In Jürgen Dix, João Alexandre Leite, and Ken Satoh, editors, *Proceedings of the Third International Workshop on Computational Logic in Multi-Agent Systems (CLIMA-02)*, 1st August, Copenhagen, Denmark, Electronic Notes in Theoretical Computer Science 70(5). Elsevier, 2002. URL: <http://www.elsevier.nl/locate/entcs/volume70.html>.
8. Rafael H. Bordini and Álvaro F. Moreira. Proving BDI properties of agent-oriented programming languages: The asymmetry thesis principles in AgentSpeak(L). *Annals of Mathematics*

- and Artificial Intelligence*, 42(1–3):197–226, September 2004. Special Issue on Computational Logic in Multi-Agent Systems.
9. Rafael H. Bordini, Fabio Y. Okuyama, Denise de Oliveira, Guilherme Drehmer, and Romulo C. Krafta. The MAS-SOC approach to multi-agent based simulation. In Gabriela Lindemann, Daniel Moldt, and Mario Paolucci, editors, *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 LNAI, pages 70–91, Berlin, 2004. Springer-Verlag.
  10. Rafael H. Bordini, Willem Visser, Michael Fisher, Carmen Pardavila, and Michael Wooldridge. Model checking multi-agent programs with CASP. In Warren A. Hunt Jr. and Fabio Somenzi, editors, *Proceedings of the Fifteenth Conference on Computer-Aided Verification (CAV-2003)*, Boulder, CO, 8–12 July, number 2725 in LNCS, pages 110–113, Berlin, 2003. Springer-Verlag. Tool description.
  11. Oscar P. Calcin, Fabio Y. Okuyama, and Aurelio M. Dias. Simulación del proceso de compra de artículos en un mercado virtual con agentes BDI. In Mauricio Solar, David Fernandez-Baca, and Ernesto Cuadros-Vargas, editors, *30ma Conferencia Latinoamericana de Informática (CLEI2004)*, pages 214–223. Sociedad Peruana de Computación, September 2004. ISBN 9972-9876-2-0.
  12. C. Castelfranchi. Engineering social order. In Zambonelli F. Omicini A., Tolksdorf R., editor, *Engineering Societies in the Agents World*, pages 1 – 18. Springer, 2000.
  13. Cristiano Castelfranchi. Simulating with cognitive agents: The importance of cognitive emergence. In Jaime S. Sichman, Rosaria Conte, and Nigel Gilbert, editors, *Multi-Agent Systems and Agent-Based Simulation*, number 1534 in LNAI, pages 26–44, Berlin, 1998. Springer-Verlag.
  14. Cristiano Castelfranchi. The theory of social functions: Challenges for computational social science and multi-agent learning. *Cognitive Systems Research*, 2(1):5–38, April 2001.
  15. Rosaria Conte and Cristiano Castelfranchi. *Cognitive and Social Action*. UCL Press, London, 1995.
  16. Mehdi Dastani, Jürgen Dix, and Amal El Fallah-Seghrouchni, editors. *Programming Multi-Agent Systems, First International Workshop, PROMAS 2003, Melbourne, Australia, July 15, 2003, Selected Revised and Invited Papers*, number 3067 in LNAI. Springer, 2004.
  17. Y. Demazeau and A. C. Rocha Costa. Populations and organizations in open multi-agent systems. In *1st National Symposium on Parallel and Distributed AI (PDAI'96)*, Hyderabad, India, 1996.
  18. Ian Foster and Carl Kesselman, editors. *The Grid 2: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann, second edition, 2003.
  19. A.C.R. Costa G.P. Dimuro and L.A.M. Palazzo. Systems of exchange values as tools for multi-agent organizations. *Journal of the Brazilian Computer Society*, 2005. (To appear, available at <http://gmc.ucpel.tche.br/valores>.)
  20. Ian Horrocks and Peter F. Patel-Schneider. Reducing OWL entailment to description logic satisfiability. In Dieter Fensel, Katia Sycara, and John Mylopoulos, editors, *Proc. of the 2003 International Semantic Web Conference (ISWC 2003)*, number 2870 in LNCS, pages 17–29. Springer, 2003.
  21. Jomi Fred Hübner. *Um Modelo de Reorganização de Sistemas Multiagentes*. PhD thesis, Universidade de São Paulo, Escola Politécnica, 2003.
  22. Jomi Fred Hübner, Jaime Simo Sichman, and Olivier Boissier. Moise+: Towards a structural, functional, and deontic model for MAS organization. In *Proceedings of the First International Joint Conference on Autonomous Agents and Multi-Agent Systems (AAMAS'2002)*, pages 501–502, Bologna, Italy, 2002. ACM Press.

23. Romula Krafta, Denise de Oliveira, and Rafael H. Bordini. The city as object of human agency. In *Fourth International Space Syntax Symposium (SSS4), London, 17–19 June*, pages 33.1–33.18, 2003.
24. João Leite, Andrea Omicini, Paolo Torroni, and Pinar Yolum, editors. *Proceedings of the Second International Workshop on Declarative Agent Languages and Technologies (DALT 2004)*, held with the Third International Joint Conference on Autonomous Agents and Multi-Agent Systems (AAMAS 2004), New York City, NY, 19th of July, 2004. To appear in Springer’s LNAI Series.
25. João Alexandre Leite, Andrea Omicini, Leon Sterling, and Paolo Torroni, editors. *Declarative Agent Languages and Technologies, First International Workshop, DALT 2003, Melbourne, Australia, July 15, 2003, Revised Selected and Invited Papers*, volume 2990 of LNCS. Springer, 2004.
26. Rodrigo Machado and Rafael H. Bordini. Running AgentSpeak(L) agents on SIM\_AGENT. In John-Jules Meyer and Milind Tambe, editors, *Intelligent Agents VIII – Proceedings of the Eighth International Workshop on Agent Theories, Architectures, and Languages (ATAL-2001), August 1–3, 2001, Seattle, WA*, number 2333 in LNAI, pages 158–174, Berlin, 2002. Springer-Verlag.
27. Álvaro F. Moreira and Rafael H. Bordini. An operational semantics for a BDI agent-oriented programming language. In John-Jule Ch. Meyer and Michael J. Wooldridge, editors, *Proceedings of the Workshop on Logics for Agent-Based Systems (LABS-02), held in conjunction with the Eighth International Conference on Principles of Knowledge Representation and Reasoning (KR2002), April 22–25, Toulouse, France*, pages 45–59, 2002.
28. Álvaro F. Moreira, Renata Vieira, and Rafael H. Bordini. Extending the operational semantics of a BDI agent-oriented programming language for introducing speech-act based communication. In João Leite, Andrea Omicini, Leon Sterling, and Paolo Torroni, editors, *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 LNAI, pages 135–154, Berlin, 2004. Springer-Verlag.
29. Fabio Y. Okuyama. Descrição e geração de ambientes para simulações com sistemas multiagentes. Dissertação de mestrado, PPGC/UFRGS, Porto Alegre, RS, Fevereiro 2003. in Portuguese.
30. Fabio Y. Okuyama, Rafael H. Bordini, and Antônio Carlos da Rocha Costa. Elms: An environment description language for multi-agent simulations. In Danny Weyns, H. van Dyke Parunak, and Fabien Michel, editors, *Proceedings of the First International Workshop on Environments for Multiagent Systems (E4MAS), held with AAMAS-04, 19th of July*, pages 67–83, number 3347 in LNAI, Berlin, 2004. Springer-Verlag.
31. Gordon D. Plotkin. A structural approach to operational semantics. Technical report, Computer Science Department, Aarhus University, Aarhus, 1981.
32. Anand S. Rao. AgentSpeak(L): BDI agents speak out in a logical computable language. In Walter Van de Velde and John Perram, editors, *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 LNAI, pages 42–55, London, 1996. Springer-Verlag.
33. Anand S. Rao and Michael P. Georgeff. Modeling rational agents within a BDI-architecture. In James Allen, Richard Fikes, and Erik Sandewall, editors, *Proceedings of the 2nd International Conference on Principles of Knowledge Representation and Reasoning (KR’91)*, pages 473–484. Morgan Kaufmann publishers Inc.: San Mateo, CA, USA, 1991.
34. Anand S. Rao and Michael P. Georgeff. BDI agents: From theory to practice. In Victor Lesser and Les Gasser, editors, *Proceedings of the First International Conference on Multi-*

- Agent Systems (ICMAS'95)*, 12–14 June, San Francisco, CA, pages 312–319, Menlo Park, CA, 1995. AAAI Press / MIT Press.
35. Anand S. Rao and Michael P. Georgeff. Decision procedures for BDI logics. *Journal of Logic and Computation*, 8(3):293–343, 1998.
  36. Máira Ribeiro Rodrigues, Antônio Carlos da Rocha Costa, and Rafael Heitor Bordini. A system of exchange values to support social interactions in artificial societies. In Jeffrey S. Rosenschein, Tuomas Sandholm, Wooldridge Michael, and Makoto Yokoo, editors, *Proceedings of the Second International Joint Conference on Autonomous Agents and Multi-Agent Systems (AAMAS-2003)*, Melbourne, Australia, 14–18 July, pages 81–88, New York, NY, 2003. ACM Press.
  37. Stuart Russel and Peter Norvig. *Artificial Intelligence - A Modern Approach*. Prentice-Hall, Englewood Cliffs, 2003.
  38. Yoav Shoham. Agent-oriented programming. *Artificial Intelligence*, 60:51–92, 1993.
  39. Munindar P. Singh, Anand S. Rao, and Michael P. Georgeff. Formal methods in DAI: Logic-based representation and reasoning. In Gerhard Weiß, editor, *Multiagent Systems—A Modern Approach to Distributed Artificial Intelligence*, chapter 8, pages 331–376. MIT Press, Cambridge, MA, 1999.
  40. Steffen Staab and Rudi Studer, editors. *Handbook on Ontologies*. International Handbooks on Information Systems. Springer, 2004.
  41. Renata Vieira, Alvaro Moreira, Michael Wooldridge, and Rafael H. Bordini. On the formal semantics of speech-act based communication in an agent-oriented programming language. *Submitted article, to appear*, 2005.
  42. Danny Weyns, H. van Dyke Parunak, and Fabien Michel, editors. *Proceedings of the First International Workshop on Environments for Multiagent Systems (E4MAS)*, held with AAMAS-04, 19th of July, New York City, NY. Number 3374 in LNAI. Springer-Verlag, 2005.
  43. Michael Wooldridge. *Reasoning about Rational Agents*. The MIT Press, Cambridge, MA, 2000.