An Approach to Rapid Prototyping of Large Multi-Agent Systems

Wamberto Vasconcelos
Centre for Intelligent Systems and their Applications (CISA)
Division of Informatics, University of Edinburgh,
80 South Bridge, EH1 1HN, Edinburgh, UK.
wvasconcelos@acm.org

Carles Sierra, Marc Esteva
Artificial Intelligence Research Institute (IIIA)
Consejo Superior de Investigaciones Científicas (CSIC)
Campus UAB, 08193, Bellaterra, Catalonia, Spain.
{sierra, marc}@iiia.csic.es

Abstract

Engineering individual components of a multi-agent system and their interactions is a complex and error-prone task in urgent need of methods and tools. Prototyping is a valuable technique to help software engineers explore the design space while gaining insight and a "feel" for the dynamics of the system; prototyping also allows engineers to learn more about the relationships among design features and the desired computational behaviour. In this paper we describe an approach to building prototypes of large multi-agent systems with which we can experiment and analyse results. We have implemented an environment embodying our approach. This environment is supported by a distributed platform that helps us achieve controlled simulations.

1 Introduction

In this paper we describe an approach to building prototypes for large multi-agent systems (MASs, for short). We also describe an environment to support engineers wishing to rapidly devise prototypes following our approach. Our approach is based on a global protocol depicting all interactions that take place in the MAS. The format and order of all interactions are formally specified as a richer kind of non-deterministic finite state machine. This formalism can be used to check for desirable properties in the protocol. An advantage we exploit is that this global protocol can be used to automatically synthesise the agents that will comprise the system.

Rapid prototyping offers means to explore essential features of a proposed system [19, 16], promoting early experimentation with alternative design choices and allowing engineers to pursue different solutions without efficiency concerns [5]; in [15] we find reports of many successful experiments of rapid prototyping. Our approach to prototyping MASs reflects the modelling methodology presented in [30] and consists of the following steps:

1. *Design of a Global Protocol* – in this initial step we prescribe the design of a global protocol, that is, a precise description of the kinds and order of messages that the components of the MAS can exchange. For this description we have used a richer form of non-deterministic finite state machines, the *electronic institutions* (or simply e-institutions) [10]. We explain more about this in Section 2.
2. *Synthesis and Customisation of Agents* – this step addresses the automatic synthesis of agents complying with the designed electronic institution. Although simple, these synthesised agents are in strict accordance with the e-institution they originate from: their behaviours conform to the specification of the global protocol. To allow for the variability of the components of a MAS and to help engineers explore the design space of individual agents, we offer means to customise the synthesised agents into more sophisticated pieces of software. We explain this step in Section 3.
3. *Definition of Prototype* – a prototype consists of an e-institution and a set of corresponding customised agents. Designers may deliberately leave empty slots in the customised agents where different design possibilities may be pursued. These slots can be completed differently giving rise to distinct prototypes. A visual interface is automatically generated to enable these slots to be filled in by the designers. This interface can be seen as a console to change parameters and monitor the simulation of the prototype. In Section 4 we give more details on this step.
4. *Simulation and Monitoring of the Prototype* – the last step is the simulation of the prototype and the collection of results. Our environment offers means for the enactment of an electronic institution: the agents are started up as self-contained and asynchronous processes that communicate by means of message-passing. This step is described in Section 5.

In the remainder of this section we address issues related to our proposal. We then describe the steps above in sections 2 to 5. In Section 6 we provide an example to illustrate our approach and we compare our approach with related work in Section 7. In Section 8 we discuss the ideas presented, draw conclusions and give directions for future work.

1.1 A Typical Scenario for MAS Prototyping

MASs consist of many components which interact dynamically, each with its own thread of control, and engaging in complex coordination protocols. MASs are more complex to correctly and efficiently engineer than stand-alone systems which use a single thread of control. They are becoming, in these days of cheap, fast and reliable interconnections amongst computers, a common way of carrying out computations.

Let us consider a scenario in which we want to design a virtual marketplace [27] where agents come to buy and sell goods. Our virtual marketplace, much like the Kasbah system depicted in [7], will be populated by agents started up by users (humans or their software agents) who wish to sell or buy goods. The agents that buy and sell are designed to be personalised to the needs of the user. Parameters such as which goods to trade, the highest price a buyer agent is prepared to pay, the lowest price a seller agent will accept to sell the goods, time constraints, negotiation strategies, and so on, should be fixed by the users prior to the agent joining the marketplace.

In order to explore the design space when building such a system, rapid prototyping is essential. Even though individual agents may be correctly developed, their interactions and the scalability of the whole system have to be empirically examined. Furthermore, to gain an insight on how the interplay among the internal features of the individual agents influences the overall dynamics of the system, the prototypes ought to offer convenient ways to change these features and to examine any resulting changes in the collective behaviour of the agents.

1.2 Prototyping Concurrent Applications

MASs can be regarded as a special kind of concurrent system, one in which the processes are agents with the generally agreed properties of proactiveness, persistence, reactiveness, awareness of the environment, autonomy and interactivity with other agents [12, 26]. An important distinction between MASs and ordinary concurrent applications concerns the nature of the interactions among the components. In MASs, the interactions are at a higher level of sophistication, as in an electronic auction, a negotiation or an argumentation [23]. Furthermore, the components of MASs are endowed with reasoning capabilities which use the interactions and any information on the environment to guide and adjust their behaviour.

We find in [16] a survey of programming languages and systems for prototyping concurrent applications. The surveyed languages and systems are not directly applicable to MASs because of the nature of interactions among components: these are too simple and mostly exchange information rather than more complex knowledge representations, normally associated with MASs. One can also notice an altogether lower level of abstraction for the intended systems with concerns on underlying protocols and message-passing services. Notwithstanding, our approach bears resemblance with the work of [4], in which simple skeletons of programs are generated from special kinds of Petri Nets. However, in that work only simple interactions are addressed.

2 Electronic Institutions

A key concept of our approach is the global protocol which describes the interactions of all components of a MAS. We use the global protocol as a sufficient description of the MAS to guide the construction of our prototypes. An advantage of using global protocols to describe MASs is the ease with which one can readily modify the interactions among components of the MAS. If, for instance, we had instead a collection of partial protocols among the different components, the modification of one of the partial protocols could cause changes in other partial protocols.

Our global protocols are represented using *electronic institutions* (e-institutions, for short) [10]. E-institutions are a richer variation of non-deterministic finite state machines. An advantage in using a finite-state machine formalism to represent protocols is that we can use automated techniques to check for properties (or their absence). For instance, protocols should not have "sinks", that is, states where the systems gets to and is never able to leave; there should not be unreachable states in a protocol; and so on. Such properties can be checked with standard graph algorithms. If our protocols were described in a more sophisticated formalism with a more operational semantics, *e.g*, in a programming language, such checks may not be easily done. Another advantage is that we can use the representation of our protocols to *synthesise* the agents that will comprise the MAS. We exploit this advantage in our approach. This is explained in detail in Section 3 below. Again, more sophisticated notations with more operational semantics would make this synthesis process a lot more complex, if not impossible.

We shall present e-institutions here in a "lightweight" version in which those features not essential to our investigation will be omitted. For a complete description of e-institutions readers should refer to [10, 24]. Our lightweight e-institutions are defined as sets of *scenes* related by *transitions*. We shall assume the existence of a communication language CL among the agents of an e-institution. We first define a scene:

Def. A *scene* is a tuple $\mathbf{S} = \langle R, W, w_0, W_f, WA, WE, \Theta, \lambda\rangle$ where
- $R = \{r_1, \ldots, r_n\}$ is the set of *roles*;
- $W = \{w_0, \ldots, w_m\}$ is a finite, non-empty set of *states*;
- $w_0 \in W$ is the *initial state*;
- $W_f \subseteq W$ is the non-empty set of *final states*;
- WA is a set of sets $WA = \{WA_r \subseteq W,\ r \in R\}$ where each $WA_r,\ r \in R$, is the set of *access states* for role r;
- WE is a set of sets $WE = \{WE_r \subseteq W,\ r \in R\}$ where each $WE_r,\ r \in R$, is the set of *exit states* for role r;
- $\Theta \subseteq W \times W$ is a set of *directed edges*;
- $\lambda : \Theta \mapsto CL$ is a *labelling function* associating edges to messages in the agreed language CL.

To illustrate this definition we provide in Fig. 1 a simple example of a scene for an agora room in which an agent willing to acquire goods interacts with a number of agents intending to sell such goods. This agora scene has been simplified – no auctions or negotiations are contemplated. The buyer announces the goods it wants to purchase, collects

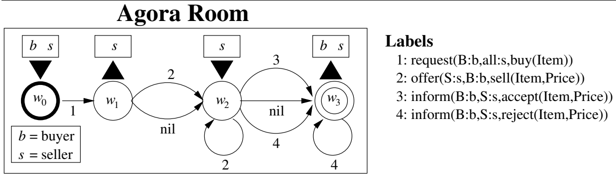

Figure 1: Simple Agora Room Scene

the offers from sellers (if any) and chooses the best (cheapest) of them. The simplicity of this scene is deliberate, so as to make the ensuing discussion and examples more concrete. A more friendly visual rendition of the formal definition is employed in the figure. Two roles b, for buyer and s, for seller, are defined. The initial state w_0 is denoted by a thicker circle (leftmost state of scene); the only final state w_3 is represented by a pair of concentric circles (rightmost state). Access states are marked with a "▼" pointing towards the state with a box containing the roles that are supposed to enter the scene at that point. Exit states are marked with a "▲" pointing away from the state, with a box containing the roles that may leave the scene at that point. For the sake of presentation, we have labelled the edges with numbers, which are defined on the right-hand side of the figure as **Labels**. A special label "nil" has been used to denote edges that can be followed without any action/event.

We can now provide a definition for e-institutions:

Def. An *e-institution* is the tuple $\mathcal{E} = \langle SC, T, \mathbf{S}_0, \mathbf{S}_\Omega, E, \lambda_E \rangle$ where

- $SC = \{\mathbf{S}_1, \ldots, \mathbf{S}_n\}$ is a finite, non-empty set of scenes;
- $T = \{t_1, \ldots, t_m\}$ is a finite, non-empty set of *transitions*;
- $\mathbf{S}_0 \in SC$ is the *root* scene;
- $\mathbf{S}_\Omega \in SC$ is the *output* scene;
- $E = E^I \cup E^O$ is a set of arcs such that $E^I \subseteq WE^{\mathbf{S}} \times T$ is a set of edges from exit states $WE^{\mathbf{S}}$ of scene $\mathbf{S}$ to transitions, and $E^O \subseteq T \times WA^{\mathbf{S}}$ is a set of edges from access states $WA^{\mathbf{S}}$ of scene $\mathbf{S}$ to transitions;
- $\lambda_E : E \mapsto p(x_1, \ldots, x_k)$ maps each arc to a predicate representing the arc's constraints.

Transitions are special connections between scenes through which agents move, possibly changing roles and synchronising their movement with other agents. We illustrate the definition above with an example comprising a complete virtual agoric market. This e-institution has more components than the above scene: before agents can take part in the agora they have to be admitted; after the agora room scene is finished, buyers and sellers must proceed to settle their debts. We show in Fig. 2 a graphic rendition of

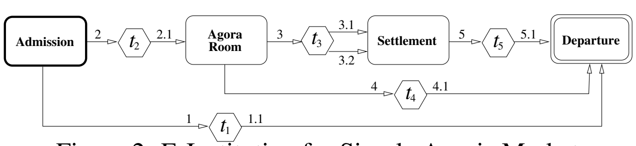

Figure 2: E-Institution for Simple Agoric Market

an e-institution for our market. The scenes are shown in the boxes with rounded edges. The root scene is represented as a thicker box and the output scene as a double box. Transitions are represented as hexagons. The arcs connect exit states of scenes to transitions, and transitions to access states. The labels of transitions have been represented as numbers. The same e-institution is, of course, amenable to different visual renditions.

2.1 Designing & Representing E-Institutions

Those wishing to design their own e-institutions can make use of a graphic editor, Islander [9, 21], incorporated into our environment. Users can prepare their e-institutions by drawing diagrams as Fig. 1 and 2 using a selection of icons and a repertoire of drawing operations. The graphic notation is a means to present the formal definitions above and allow their more ergonomic manipulation. We show in Fig. 3 a screenshot of Islander.

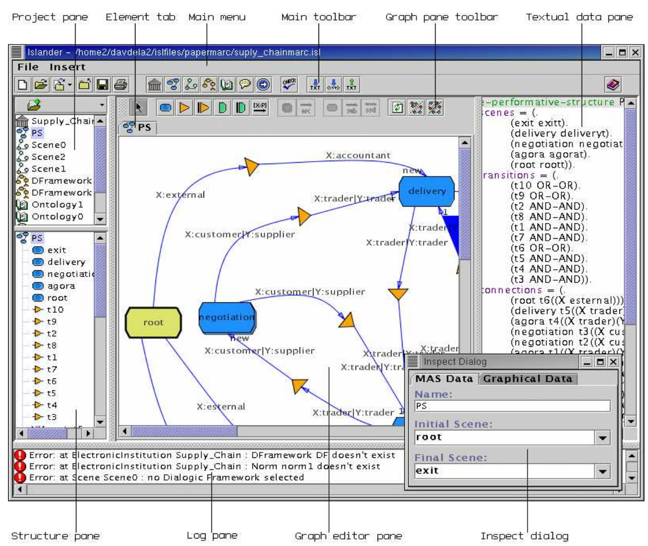

Figure 3: Islander Graphic Editor for E-Institutions

Graphically represented e-institutions are translated into a logical formalism [31] implemented in Prolog [2], making our representation computer-processable. This makes it easier to synthesise our simple agents, as we shall see below. We show in Fig. 4 our Prolog representation for the agora

```
roles(agora,[buyer,seller]).  states(agora,[w0,w1,w2,w3]).
initial_state(agora,w0).  final_states(agora,[w3]).
access_states(agora,buyer,[w0]).
access_states(agora,seller,[w0,w2]).
exit_states(agora,buyer,[w3]).
exit_states(agora,seller,[w1,w3]).
theta(agora,[w0,request(B:buyer,all:seller,buy(I)),w1]).
theta(agora,[w1,offer(S:seller,B:buyer,sell(I,P)),w2]).
theta(agora,[w1,nil,w2]).
theta(agora,[w2,offer(S:seller,B:buyer,sell(I,P)),w2]).
theta(agora,[w2,inform(B:buyer,S:seller,accept(I,P)),w3]).
theta(agora,[w2,inform(B:buyer,S:seller,reject(I,P)),w3]).
theta(agora,[w2,nil,w3]).
theta(agora,[w3,inform(B:buyer,S:seller,reject(I,P)),w3]).
```

Figure 4: Representation of Agora Room Scene

room scene graphically depicted in Fig. 1 above. Each component of the formal definition has its corresponding representation. Since many scenes may co-exist within one e-institution, the components are parameterised by a scene name (first parameter). The Θ and λ components of the definition are represented together in $\texttt{theta}/2$, where the second argument holds a list containing the directed edge as the first and third elements of the list and the label as the second element.

Any scene can be conveniently and economically described in this fashion. E-institutions are collections of scenes in this format, plus the extra components of the tuple comprising its formal definition. In Fig. 5 we present a

```
scenes([admission,agora,settlement,departure]).
transitions([t1,t2,t3,t4,t5]).
root_scene(admission).            output_scene(departure).
arc([admission,w3],p[1],t1).      arc([admission,w3],p[2],t2).
arc(t1,p[1.1],[departure,w0]).    arc(t2,p[2.1],[agora,w0]).
arc([agora,w3],p[3],t3).          arc([agora,w3],p[4],t4).
arc(t3,p[3.1],[settlement,w0]).
arc(t3,p[3.2],[settlement,w0]).
arc(t4,p[4.1],[departure,w0]).
arc(t5,p[5.1],[departure,w0]).
arc([settlement,w3],p[5],t5).
```

Figure 5: Representation of Agoric Market

Prolog representation for the agora market e-institution. Of particular importance are the arcs connecting scenes: these are represented as `arc`/3 facts storing the first argument of which holds (as a sublist) an exit state of a scene, the second argument holds the predicate (constraint) $p_{[i]}$ which enables the arc, and the third argument is the destination transition. Another `arc`/3 shows arcs leaving a transition and entering an access state of a scene.

Our graphic editor offers users means to check for desirable properties in their e-institutions via model-checking. It does so by using the underlying Prolog representation and standard model-checking techniques [17].

3 Synthesis of Agents

Our choice of a global protocol has the advantage that we can use it to synthesise the agents that will comprise our MAS. This feature allows designers to experiment with different variations of a specific global protocol, knowing that the corresponding prototype will be automatically generated.

In [31] we introduce a simple way to synthesise agents from our e-institutions. We devised a means to use the logic representation of the e-institution in order to obtain a simple set of Horn clauses which capture the behaviours for the agents partaking the e-institution. The synthesis obtains, for the roles of each scene, a set of Horn clauses which represent the connection among the states and the events, *i.e.*, sending or receiving messages, associated with these connections (edges).

We show in Fig. 6 portions of a Prolog program to synthesise agents from an e-institution represented in the above logical form. Predicate `cl_arc`/3 uses the role (first argument) and an arc (second argument) to assemble an agent clause (third argument) of the form `s(`$LInfo_1$`):-holds(`*Pred*`),s(`$LInfo_2$`)` where $LInfo_i$ are lists storing information on the scene, transition, state and role the agent currently is ($i = 1$) and will be next ($i = 2$). States of computation in an e-institution are uniquely defined by their label and the scene they are part of. The role that a component plays within an e-institution is also employed when we

```
cl_arc(R,arc([Sc,St],P,T),Clause):-
   satisfy(R,P),
   Clause = (s([Sc,St,R]):-holds(P),s([T,R])).
cl_arc(R,arc(T,P,[Sc,St]),Clause):-
   satisfy(R,P),
   Clause = (s([T,R]):-holds(P),s([Sc,St,R])).

cl_theta(R,theta(Sc,[St,L,NSt]),Clause):-
   L =..  [_,_:R,_,_],
   Clause = (s([Sc,St,R]):-send(L),s([Sc,NSt,R])).
cl_theta(R,theta(Sc,[St,L,NSt]),Clause):-
   L =..  [_,_,_:R,_],
   Clause = (s([Sc,St,R]):-rec(L),s([Sc,NSt,R])).
cl_theta(R,theta(Sc,[St,_,NSt]),Clause):-
   Clause = (s([Sc,St,R]):-s([Sc,NSt,R])).
```

Figure 6: Portion of Program to Synthesise Agents

want to follow an e-institution. A list was used to store all relevant components – this way only one argument is required in the agent. The `satisfy`/2 predicate ensures that agents with that role are allowed to follow the arc – this depends on the predicate labelling the edge to/from a transition between scenes. Predicate `cl_theta`/3, similarly, uses the role (first argument) and an intra-scene Θ edge (second argument) to obtain an agent clause (third argument) of the form `s(`$LInfo_1$`):-`P`(`*Label*`),s(`$LInfo_2$`)`, P being either `send`/1 or `rec`/1. These clauses are obtained depending on whether the sender is of the same role as the first argument (predicate `send`/1 is employed in this case) or whether the receiver is of the same role (predicate `rec`/1 is used instead). If the role is not the sender nor the receiver then the assembled clause is of the form `s(`$LInfo_1$`):-s(`$LInfo_2$`)`. Auxiliary predicates are required to exhaustively combine the roles of every scene with all appropriate edges and arcs to obtain the complete agent.

We show in Fig. 7 some of the clauses synthesised from the e-institution of Fig. 2. The top clauses depict the agora

```
s([agora,w0,buyer]):-
  send(request(B:buyer,all:seller,buy(Item))),
  s([agora,w1,buyer]).
...
s([agora,w3,seller]):-
  rec(inform(B:buyer,S:seller,reject(Item,Price))),
  s([agora,w3,seller]).
s([admission,w3,seller]):- holds(p[1]),s([t1,seller]).
...
s([t5,buyer]):- holds(p[5.1]),s([departure,w0,buyer]).
```

Figure 7: Synthesised Agent from E-Institution

scene. The bottom clauses are the transitions among scenes. Additional predicate definitions are required for message exchange and these are inserted at a later stage. An agent whose predicates are all defined is a completely operational and executable Prolog program which captures the behaviours within an e-institution.

The clauses define predicate `s`/1 which uses a list to represent the current state. The list consists of the name of the scene, the name of the state in the graph and the role of the agent. Depending on the role of the agent, a suitable action `send`/1 or `rec`/1, to send and receive a message, respectively, is chosen for the clause. By using the clauses with the standard SLDNF resolution mechanism [2] we get all possible correct behaviours of the agents in the e-institution.

3.1 Customising Synthesised Agents

The clauses synthesised from the e-institution describe all correct behaviours an agent may have. Because it is an exhaustive process, all scenes, edges, transitions and roles are considered. However, if we were to use the same clauses to define agents which would enact an e-institution, they would all have precisely the same behaviours. Although this might be desirable at times, we also want to offer means for designers to add *variability* to the agents synthesised and use them in our prototypes.

We have detected three different ways designers can customise their synthesised agents. A straightforward first manner is by *restricting* the agent's possible behaviours, that is, designers may choose from the existing scenes, edges, roles and transitions, those parts of the e-institution that the agent can partake. This can be done by editing the definition of the e-institution to leave out unwanted parts or by altering the synthesised set of Horn clauses to remove those parts describing the unwanted behaviours.

A second way is by *augmenting* the set of Horn clauses to add computational capabilities. The synthesised clauses supply us with all correct behaviours but capabilities such as keeping a record of states the agent has been at or messages that have been sent or received are not available. We have developed [31] a semi-automatic tool with which designers can augment the synthesised clauses with such extra capabilities. Designers manipulate the clauses (complete or restricted ones) with our tool conferring extra functionalities on them. This tool is a special kind of *program editor*, along the lines of [14], and supported by standard transformation techniques for logic programs [29]. The tool is an extensible programming environment to which new operations can be added. The sequence of operations applied to an agent is recorded and users may backtrack to previous points in order to change their design decisions. Users employ the set of synthesised clauses as an initial design template to which different capabilities are added. Parts of the designed agent can be left deliberately unspecified, for instance, the lowest price a seller agent may take or the highest offer a buyer agent may make. These parameters and their possible values are used to automatically create the Lab Console shown in Fig. 12.

We show in Fig. 8 the first two clauses of the synthesised

```
s([agora,w0,buyer],Stock,Msgs):-
  chooseItem(Stock,Item),
  send(request(B:buyer,all:seller,buy(Item))),
  updateMsgs(send,Msgs,buy(Item),NewMsgs),
  s([agora,w1,buyer],Stock,NewMsgs).
s([agora,w0,seller],Stock,Msgs):-
  rec(request(B:buyer,all:seller,buy(I))),
  updateMsgs(rec,Msgs,buy(Item),NewMsgs),
  s([agora,w1,seller],Stock,NewMsgs).
...
```

Figure 8: Augmented Agent

agent with an example of the kinds of customisation via augmenting we allow users to perform within our environment. In order to save space, we focus only on the two first clauses. Starting with the synthesised clauses of Fig. 7 the user gradually adds features to the agent's capabilities. We show the added portions underlined. The first modification inserts a programming technique which carries a `Stock` data structure around as the execution proceeds; this data structure is employed to obtain via predicate `chooseItem`/2 the value of `Item` in the first clause. The definition for `chooseItem`/2 has to be supplied. The second modification concerns the addition of another technique to assemble a data structure `Msgs`. This data structure stores the messages sent and received and is updated by means of calls to predicate `update`/3 (which should also be supplied). The environment ensures that arguments are consistently inserted, and the user must provide suitable definitions for any auxiliary predicates. The original set of behaviours of the synthesised agent is preserved in our extended program above. Ideally this should always happen, making sure that agents will perform correctly and efficiently/intelligently.

A third manner of adding variability to the synthesised agents concerns *changing the policy* of non-deterministic choices. The clauses, when used with the standard way of implementing SLDNF resolution [2], offer a default behaviour that will be followed by the agent. Any non-determinism involving two or more clauses depicting the edges leaving one same state would be deterministically solved. In such scenario the clauses are assumed to comprise an ordered sequence and the first clause obtained in this sequence which is successfully proven will be the one chosen. This feature, however, may not always be desired. Logic programming has long been praised as a useful tool for meta-programming – a meta-program is a program whose data denotes another (object) program, both of which are in the same language. We provide designers with a library of meta-interpreters which allow the meta-reasoning about non-deterministic choices. We show in Fig. 9 a simple meta-interpreter for our needs. Its two initial clauses

```
meta((G,Gs)):- meta(G), meta(Gs).
      meta(G):- system(G), call(G).
      meta(G):- setof(B,clause((G:-B)),Bs),
                chooseBody(Bs,ChB),
                meta(ChB).
```

Figure 9: Meta-Interpreter for Agents

are the usual meta-interpreter definitions for conjuncts (first clause) and system built-ins (second clause) [2]. The third clause generalises the usual meta-interpreter definition to handle user-created predicates: those clauses `(G:-B)` the head of which unifies with goal `G` are all collected and one of their bodies `Bs` is chosen via predicate `chooseBody`/2 as the `ChB` that will be further used in the meta-interpretation. Predicate `chooseBody`/2 ought to be defined by the agent's designer and should reflect the policies and attitudes of the agent regarding non-deterministic choices.

4 Building MAS Prototypes

A prototype of a MAS consists of an e-institution and agents to enact it. These agents have been synthesised from the e-institution (or from portions of it) and the designer has customised them by restriction, by augmenting, by meta-programming or by a combination of these. This customisation is a means to explore the design space of individual agents and by extension of the MAS as a whole. The designer selects some of these agents to make up the prototype. However, these individual agents may still have undefined parts, such as values of constants or a definition of a predicate. This is a desirable feature of our approach that allows delaying these design decisions until the simulation of the prototype.

We show in Fig. 10 the sequence of steps leading to a prototype. Arrow 1 shows the synthesis of Agent from E-Institution. Agent defines a design space (shown in the diagram as a hierarchy) that engineers can explore with our tools and techniques. Engineers choose among elements of the design space those customised versions of Agent (black boxes of the hierarchy) whose performance/behaviour they wish to examine. These may require further tuning of parameters and hence we need an adequate interface to offer these parameters to our users. This interface is the Lab Console and it is automatically generated (arrow 2). The E-Institution and the Lab Console (once the engineers supply its remaining parameters, if any) are put to use in arrow 3 when a complete prototype is defined.

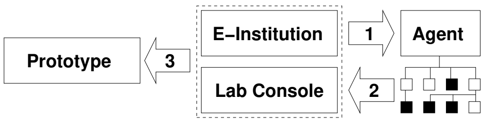

Figure 10: Building Prototypes for MASs

In its simplest form, the Lab Console allows engineers to set the kinds and numbers of agents that will make up the prototype MAS. Other typical parameters of agents are values of constants, description of functions (for instance, threshold of negotiation protocols [23]), definition of predicates (for instance, the belief revision of an agent's BDI model [32]), and so on. Engineers may decide to leave these features unspecified and allow their customisation via the console. These features will thus be parameters for different experiments in distinct prototypes.

A screenshot of our implementation of the laboratory console is shown in Fig. 11, at the stage where parameters can be tuned. We have employed HTML [22] to dynamically generate our front-end which is then viewed via conventional web browsers. Depending on the parts of the customised agents that the designers left unspecified and their associated choices our environment offers an appropriate presentation using the Prolog P*i*LLoW library [6]. The parameter tuning stage simply collects the user's design choices, postponing the actual synthesis of the prototype until the very last moment. By using a web-based front-end we can offer our environment simultaneously to many (remote) users. Our environment is supported by Prolog CGI scripts, again using the P*i*LLoW library, which allows a seamless coupling between the interface and the simulation platform.

We aim to build prototypes with hundreds of agents to simulate. Rather than customising individual agents, we customise groups of agents instead. Clearly, this approach allows for the customisation of individual agents – these comprise groups of only one element. We customise a group of agents by creating a *profile* for the group. Fig. 11 illustrates the creation of such a profile. To create a profile, the user selects a name for the profile, the number of agents that will follow the profile, and the roles for these agents inside the e-institution. The actual parameters associated with the profile are customised in a subsequent screen (Fig. 15).

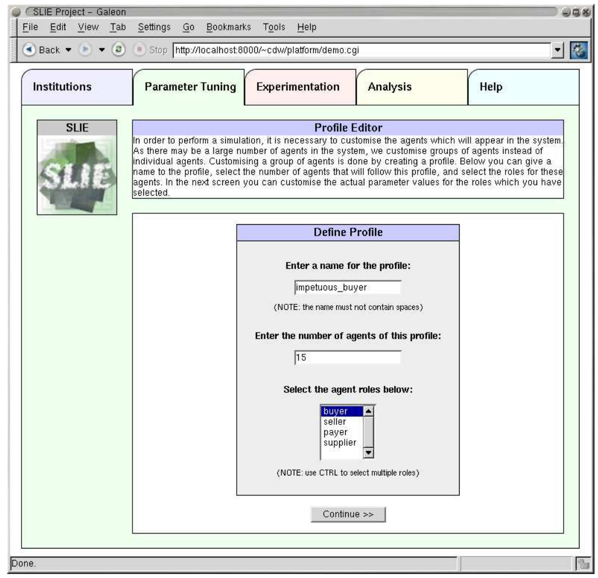

Figure 11: Definition of Profile

The use of profiles provides a means to customise large numbers of agents at a time, thus rapidly generating large simulations. Profiles also hide the details of individual agents behind a set of parameters, meaning that users need not be concerned with the underlying logic at work inside the agents, though our environment also offers the ability to alter agent definitions. Furthermore, profiles can be constructed and stored by different users, meaning that a simulation can be quickly assembled from pre-defined profiles. The simulation occurs only when all of the required profiles have been defined and included in the prototype. All of these advantages mean that the user can concentrate on tuning the simulation, rather than spending time simply building the model.

5 Simulation of Prototypes

The synthesised console provides an interface to an e-institution and associated agents which will enact it. When the user fully tunes the controls of the console then a prototype is generated. This prototype can be simulated and monitored. We show in Fig. 12 a diagrammatic account of this stage of our approach. Once these controls and pa-

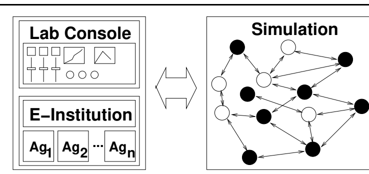

Figure 12: Simulation & Monitoring of MASs Prototypes

rameters are set, a simulation of the MAS can take place. The console then displays results of the simulation. The controls/parameters may then be changed and another simulation takes place, its results being shown on the laboratory console. As designers get a better understanding of the MAS, they can find the adequate "tuning" for the features of the MAS, that is, the choice of controls/parameters that yield desired behaviours. These can then be incorporated to the design of the final implementation of the MAS. Alternatively, depending on the possible values of the parameters to be explored, we can apply automatic means to explore the parameter setting. The work in [25] investigates the coupling of our method/environment with genetic programming to explore the parameters of a prototype.

An e-institution is enacted by having the specified number and types of agents asynchronously following the connections among states and scenes and sending and receiving messages. Each agent is a self-contained process that communicates by asynchronous message passing. We defined a precise semantics for a class of e-institutions [28] and implemented a platform incorporating this semantics. This platform employs administrative agents to look over instances of scenes and transitions and to ensure that agents appropriately move along the states of the e-institution.

Any device aimed at carrying out experiments must offer means to *vary* some parameters while keeping *control* over other parameters. The *results* of the experiment must be output so as to provide feedback on the tuning of parameters. The Lab Console shown in Fig. 12 above offers users the controls and parameters of the experiment and shows back results of the simulation. There are simple services for the plotting of results obtained during the enactment of an e-institution. Engineers can, depending on the changes they need/want to perform, go back to any of the previous steps and try different alternatives reusing previous designs.

6 Prototyping Agoric Markets

In this section we provide an example to illustrate our approach. We shall use the e-institution for the simple agoric market shown in Fig. 2. Let us assume, for the sake of simplicity, that the **Admission**, **Settlement** and **Departure** scenes are completely deterministic, that is, they do not have more than one edge leaving a state. This feature will influence the parts of the synthesised agents responsible for their enactment of those scenes: those parts will also be deterministic and hence will not require customisation. The **Agora Room** scene, on the other hand, has non-determinism which has to be explicitly dealt with by the agents performing in it.

We shall assume there is a finite set of items with their corresponding suggested retail prices. Buyer agents will try to buy all these items from the seller agents. Seller agents, as specified in the **Agora Room** scene, must determine the price at which they want to sell each item (label 2 of Fig. 2). This feature allows for the study of design choices of the seller agents: these can be either *greedy*, when their pricing policy maximises profit, or *considerate*, when their pricing is low. The greedy/considerate design choice is, in fact, a continuum, but we have chosen to make it discrete to simplify our analysis. When we customise our seller agents to deal with their pricing policy, we define the functions which implement the respective policies and leave a slot with the possible choices *greedy* or *considerate*. Depending on the choice taken, the distinct policies are incorporated. We can also pursue the continuum alternative and have a slot for the profit margin which will be a numeric value between 0 and 100 which will be used by the seller agents when assigning prices to items.

During the customisation stage users experiment with different designs, applying standard program editing commands to the synthesised agents, possibly with further manual editing. Users may choose to leave parts (*e.g.*, the value of a constant, the format of an arithmetic expression or a predicate) of the customised agent undefined, thus ending up with a kind of *open program* [11]. In our environment, these undefined parts "stick out" of the clauses of the agent, prompting engineers to consider their definition before the agent can be run. Through this approach, engineers can fully define an agent (open) program by combining it with components previously defined or with freshly devised parts. Engineers can provide alternative definitions/values for the missing parts, and associate these alternatives with descriptive labels which will be used to automatically synthesise the console for a class of prototypes.

We have employed the augmented clauses shown in Fig. 8 to prepare two kinds of agents, the seller and the buyer, by restricting their roles in the agora scene. The clauses for the remaining scenes and transitions are the same in both kinds of agents. We can then further customise the seller agent and enable it to establish a pricing policy. We can be very specific and independently carry out the alterations which will define the greedy and considerate policies, but we have noticed that these are very similar, the only distinction being the percentage of profit to be added to the price. Rather than designing the two kinds of seller agents independently, we adopt a more abstract design principle and postpone the particular choices to a later stage. We show in Fig. 13 the clause of the seller agent where the pricing is established as well as the definition of one of the auxiliary predicates. The $s/3$ definition shows the edge $w_1 \rightarrow w_2$ when the seller agent responds to a buyer request: the actual request `request:buy(Item)` is retrieved from the messages received `Msgs`, the price of

```
s([agora,w1,seller],Stock,Msgs):-
  member(request:buy(Item),Msgs),
  pricing(Item,Price),
  send(offer(S:seller,B:buyer,sell(Item,Price))),
  updateMsgs(send,Msgs,offer:sell(Item,Price),NewMsgs),
  s([agora,w2,seller],Stock,NewMsgs).

pricing(Item,Price):-
  retailPrice(Item,RPrice),
  greed(Profit),
  Price is RPrice + (RPrice * Profit).
```

Figure 13: Portion of Seller Agent

`Item` is established via predicate `pricing`/2, the offer is sent to the buyer agent, the messages sent/received are updated via `updateMsgs`/4 and finally the seller agent moves to state $w2$. Predicate `retailPrice`/2 maps each `Item` (first argument) to its suggested retail price `RPrice` (second argument).

Predicate `pricing`/2 calculates the `Price` of `Item` but it requires the definition of predicate `greed`/1 which obtains the profit margin the agent is to adopt. The distinction between a greedy and a considerate seller agent lies in the definition of `greed`/1. Both the continuum and the discrete possibilities can be exploited with suitable definitions of `greed`/1. The designer of the prototype associates the possible definitions of `greed`/1 with informative labels which will be used to automatically assemble the console of the prototype.

We have noticed that the **Agora Room** scene also allows for the customisation of buyer agents. By examining the scene definition of Fig. 1, we can see that a buyer agent has a non-deterministic choice: when it is in state w_2 it can either remain in w_2, move to w_3 via edge 3, move to w_3 via edge 4 or move to w_3 via edge nil. This portion of the scene allows us to customise different kinds of buyer agents, depending on how we want them to make their choices of how they behave. It might be useful to use a metaphor to introduce the different behaviours: when the seller agents send out their offers, buyer agents may react in an *impetuous* fashion and accept the first offer they get, that is, they follow edge 2 only once (*i.e.* they receive only one offer) and then move to w_3 via edge 3. Alternatively, the buyer agents may react in a more *cautious* way and wait for a minimal number of offers (*i.e.* loops in w_2 via edge 2) before choosing (via edge 3) the cheapest of them. Again these opposites define a spectrum of possibilities: if we use n to describe the number of offers a buyer agent must get before it decides on one of them, then we have one associated with the *impetuous* end of the spectrum and any number greater than 1 with the *cautious* end. We can customise our buyer agents to incorporate these possible design choices and leave a slot which allows them to be selected easily in order to assemble a complete prototype. We have left a slot with the choices *impetuous*/*cautious* but we could also have used natural numbers for the continuum possibility.

We now proceed to customise the augmented clauses of Fig. 8, but this time the clauses defining the behaviour of the agent in the agora scene are restricted to the buyer role. In Fig. 14 we show the clauses of the buyer agent where the offers are collected (second clause of `s`/3) and accepted (first clause of `s`/3). We have engineered these clauses from those of Fig. 8, customised to meet our needs; the remaining clauses of the agent have not been altered. Messages are received in the second clause until `minimalOffers`/1 is satisfied – this predicate ensures that the agent has received the minimal number of offers. The definition of predicate `minimalOffers`/1 employs the auxiliary predicate `minimal`/1 which returns a natural number standing for the quantity of messages the agent should wait for. Again, it is possible to use the definition of `minimal`/1 to exploit both for the continuous and discrete possibilities for the cautious/impetuous agents. Predicate `minimalOffersAux`/2 counts the number of offers received and checks that it is above `NoMsgs`. Depending on how predicate `minimal`/1 is defined, we have the impetuous or careful buyers. The designer then associates the definitions of `minimal`/1 with labels used to automatically assemble the console of the prototype.

```
s([agora,w2,buyer],Stock,Msgs):-
  minimalOffers(Msgs),
  chooseOffer(Msgs,S,Item,Price),
  send(inform(B:buyer,S:seller,accept(Item,Price))),
  updateMsgs(send,Msgs,inform:sell(Item,Price),NewMsgs),
  s([agora,w3,buyer],Stock,NewMsgs).
s([agora,w2,buyer],Stock,Msgs):-
  rec(offer(S:seller,B:buyer,sell(Item,Price))),
  updateMsgs(rec,Msgs,offer:sell(Item,Price),NewMsgs),
  s([agora,w2,buyer],Stock,NewMsgs).

minimalOffers(Msgs):-
  minimal(NoMsgs),
  minimalOffersAux(Msgs,NoMsgs).
```

Figure 14: Portion of Buyer Agent

The above agents together with the Agoric Market e-institution can be used to define a class of prototypes. Following our approach, our environment uses the definitions of predicates `greed`/1 and `minimal`/1 above and their labels to synthesise the console which will control the simulations of the different prototypes. We show in Fig. 15 the interface for the prototypes which use the discrete impetuous/cautious and greedy/considerate kinds of agents. Designers define a prototype by assembling a set of profiles together and supplying any missing parameters. In our running example, we have four previously defined profiles, *viz.*, *impetuous_buyer*, *cautious_buyer*, *greedy_seller* and *considerate_seller*, defined as collections of 15 agents each, incorporating the *buyer* (*impetuous_buyer* and *cautious_buyer*) and *seller* (*greedy_seller* and *considerate_seller*) customised clauses above, respectively. The clauses above do not uniquely define an agent since

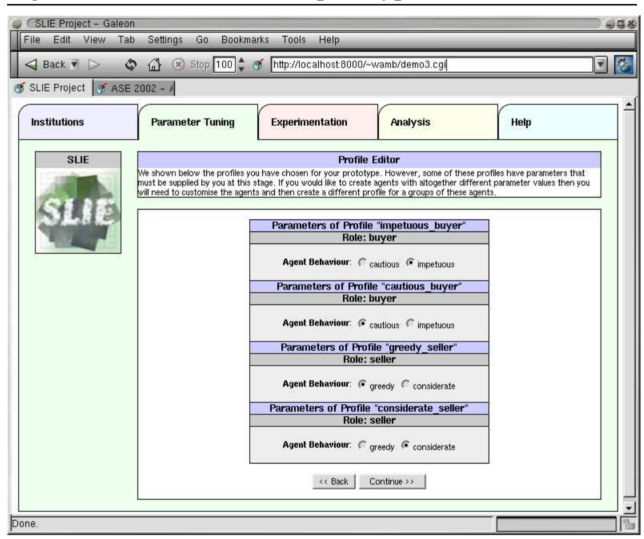

Figure 15: Console for Agoric Market Prototype

there are, for the *buyer* and *seller* agents, two possible definitions for a predicate: these options are adequately offered to the designers via our interface. In Fig. 15 we can see the offer of two possible choices for the buyers, *cautious* or *impetuous*; likewise, the *seller* agents have the options of *greedy* and *considerate* policies. We might also leave portions of the e-institution unspecified and have it defined at this stage: for instance, the items to be sold and their retail prices, the amount of money given to each buyer agent, and so on.

We can exploit scenarios where there are more items than buyers, the exact amount or fewer items than buyers, and compare the overall dynamics of the MAS in the runs of the e-institution. Designers define the values they want to follow during the simulation; the values are collected and stored in files where they can be manipulated and shown in alternative graphical formats. By experimenting with the number of each type of agent and monitoring the results obtained, engineers can explore the overall dynamics of the MAS. A whole class of prototypes can be quickly built by setting the parameters (*i.e.* providing values for the slots). In our example, there is a trade-off between being an impetuous or a cautious agent: the latter may be able to make better-informed decision by collecting offers, but they may be beaten by the quicker former agents. Similarly, the greedy and considerate seller agents have a trade-off: considerate sellers have a lower profit margin but they sell more items than greedy sellers.

7 Related Work

Our approach has parallels with [20] where logic programming is also exploited to specify and simulate MASs. Theirs is a more complex framework where a number of design features are incorporated into the MASs and their agent components: they use an object-oriented description for the overall architecture of the MAS as well as a linear logic language for describing individual agents. Although these might be appropriate choices, they may not meet the needs or the preferences of particular users. [8] extends the work of [20] allowing the design of heterogeneous and open MASs by incorporating a mediator system and a generic agent execution platform.

Computational logic has been advocated as a means to specify software, offering distinct ways to analyse the specification and, when needed and appropriate, to execute it [13]. Such features, exploited in the days of standalone systems, have also proved to be applicable to MASs [1, 3, 18]. A common problem with those advocating logic programming for MASs is that they also tend to propose their own architectures and logics which, albeit generic and expressive, may not be adequate nor appeal to everyone. Contrastingly, our work uses as building blocks very simple and standard logic programming constructs in their usual syntax and semantics. Any higher-level architectural restrictions on MASs can, however, be specified via e-institutions. Furthermore, any logic program (implementing arbitrary deductive logics) can be used to guide the synthesised agents as they make any non-deterministic choices.

8 Conclusions & Future Work

In this paper we have described an approach for rapid prototyping large multi-agent systems (MASs). We have incorporated this approach into an environment to support engineers building their prototypes. The approach follows four steps, *viz.*:

1. Design of a global protocol, formalised as an electronic institution.
2. Synthesis of agents from the global protocol and their customisation.
3. Definition of a prototype consisting of a population of the previously synthesised/customised agents to enact a global protocol.
4. Simulation and monitoring of the prototype.

After analysis of the results from the simulation, users may go back to any of the previous steps. This process gives rise to a virtuous lifecycle, as reported in [31]. The further away from the simulation the step to be re-done is the more dramatic the changes are to the design of the MAS prototype. The changes in step 3 have less of an impact in the overall dynamics of the prototype than those in step 2; the changes in the global protocol in step 1 are the most dramatic.

Our environment offers means to carry out simulations of our prototypes. Each agent becomes a self-contained asynchronous process which communicates via message-passing. The environment incorporates a platform to enact electronic institutions [28] with administrative agents which make sure the kinds and order of messages sent are those specified.

Being able to rapidly build prototypes of complex MASs allows engineers to experiment with alternative design choices and to get a “feel” for the important features of the components’ design and how these affect the overall behaviour of the system. When design features are sufficiently understood and can be related to the system dynamics, engineers can proceed towards more complete, stable and efficient versions of the MAS. Ideally, a rapid prototyping environment should offer means to automatically transform a prototype into an efficient implementation [19, 16]. Although in our environment we have not made any provisions for such transformation, it is technically possible to synthesise from the e-institution, for instance, a C or Java program for the agents. The customisation stage, however, since it involves the manipulation and alteration of a program, is more difficult to achieve using the syntax of C or Java. Ideally, the transformation should be delayed to the very last moment, but the Horn clauses of the customised agents may be too complex for a straightforward translation onto another programming language.

We might choose to view a prototype as an idealised (correct) version of the MAS to be built. Following this idea, any foreign agent willing to join in the MAS does not do so directly: they are assigned a synthesised "proxy" agent which is guaranteed to follow our e-institution. Any choice points and other customisation possibilities of the proxy agent are presented to the foreign agent which can adjust them to its needs. In this case, the prototype becomes the inner kernel of the actual MAS and the foreign agents are an outer layer. We are currently investigating how the synthesised agents could be presented to foreign agents and customised as proxy agents.

Acknowledgements: This work was sponsored by the European Union, contract IST-1999-10208, research grant **Sustainable Lifecycles in Information Ecosystems** (SLIE). Thanks are due to Chris Walton for proofreading earlier versions of this document and also for useful suggestions. We also thank Dave Robertson for his comments and improvements on earlier versions of the document. We thank the anonymous referees for their useful suggestions to improve this paper. Any remaining mistakes or imprecisions are the authors' fault only.

References

[1] J. J. Alferes, P. Dell'Acqua, E. Lamma, J. A. Leite, L. M. Pereira, and F. Riguzzi. A Logic Based Approach to Multi-Agent Systems. *ALP Newsletter*, 14(3), Aug. 2001.

[2] K. R. Apt. *From Logic Programming to Prolog*. Prentice-Hall, U.K., 1997.

[3] M. Bozzano, G. Delzanno, M. Martelli, V. Mascardi, and F. Zini. Logic Programming & Multi-Agent Systems: a Synergic Combination for Applications and Semantics. In *The Logic Programming Paradigm: a 25-Year Perspective*, pages 5–32. Springer Verlag, 1999.

[4] G. Bucci and E. Vicario. Rapid Prototyping through Communicating Petri Nets. In N. Kanopoulos, editor, *Procs. 3rd Int'l Workshop on Rapid System Prototyping*, pages 58–75, Los Alamitos, CA, USA, 1994. IEEE Press.

[5] Budde, R. and Kuhlenkamp, K. and Mathiassen, L. and Züllighoven, H., editor. *Approaches to Prototyping*. Springer-Verlag, New York, NY, USA, 1984.

[6] D. Cabeza, M. Hermenegildo, and S. Varma. WWW Programming using Computational Logic Systems (and the P*i*LL*o*W/CIAO Library). Technical report, Computer Science Department, Technical University of Madrid, 1996.

[7] A. Chavez and P. Maes. Kasbah: An Agent Marketplace for Buying and Selling Goods. In *Procs. 1st Int'l Conf. on the Practical Applic. of Intell. Agents & Multi-Agent Technology (PAAM'96)*, Blackpool, UK, 1996. Practical Application Co.

[8] P. Dart, E. Kazmierczak, L. Sterling, M. Martelli, V. Mascardi, F. Zini, and V. Subrahmanian. Combining Logical Agents with Rapid Prototyping for Engineering Distributed Applications. In *Procs. 9th Software Tech. & Engineering Practice (STEP'99)*, Pittsburgh, PA, USA, Sept. 1999. IEEE Press.

[9] M. Esteva, D. de La Cruz, and C. Sierra. ISLANDER: an Electronic Institution Editor. In *Proc. 1st Int'l Joint Conf. on Autonomous Agents & Multi-Agent Systems (AAMAS 2002)*, Bologna, Italy, 2002. ACM, U.S.A.

[10] M. Esteva, J.-A. Rodríguez-Aguilar, C. Sierra, P. Garcia, and J. L. Arcos. On the Formal Specification of Electronic Institutions. volume 1991 of *LNAI*. Springer-Verlag, 2001.

[11] P. Flener, K.-K. Lau, M. Ornaghi, and J. Richardson. An Abstract Formalisation of Correct Schemas for Program Synthesis. *J. Symb. Computation*, 30(1), July 2000.

[12] A. Franklin and A. Graesser. Is it an Agent, or just a Program? In *Intelligent Agents III*. Springer-Verlag, Berlin, 1997.

[13] N. E. Fuchs. Specifications are (Preferably) Executable. *Software Engineering Journal*, pages 323–334, Sept. 1992.

[14] A. T. Goldberg. Knowledge-Based Programming: A Survey of Program Design and Construction Techniques. *IEEE Trans. on Soft. Eng.*, SE-12(7):752–768, July 1986.

[15] V. S. Gordon and J. M. Bieman. Rapid Prototyping: Lessons Learned. *IEEE Software*, 12(1):85–95, 1995.

[16] Hasselbring, W. Programming Languages and Systems for Prototyping Concurrent Applications. *ACM Computing Surveys*, 32(1):43–79, 2000.

[17] G. J. Holzmann, E. Najm, and A. Serhrouchni. SPIN Model Checking: an Introduction. *Int. Journal of Software Tools for Technology Transfer*, 2(4):321–327, Mar. 2000.

[18] R. A. Kowalski and F. Sadri. From Logic Programming Towards Multi-agent Systems. *Annals of Mathematics and Articial Intelligence*, 25(3-4):391–419, 1999.

[19] Luqi. Computer Aided System Prototyping. In *Procs. 1st Int'l Workshop on Rapid System Prototyping*, pages 50–57, Los Alamitos, CA, USA, 1992. IEEE Press.

[20] M. Martelli, V. Mascardi, and F. Zini. Specification and Simulation of Multi-Agent Systems in CaseLP. In *Proc. of APPIA-GULP-PRODE*, L'Aquila, Italy, Sept. 1999.

[21] E-Institutor Project. Islander Graphical Editor. IIIA-CSIC, Bellaterra, Catalonia, Spain, `http://e-institutor.iiia.csic.es/e-institutor/software/islander.html`, 2002.

[22] C. Musciano and B. Kennedy. *HTML & XHTML: The Definitive Guide*. O'Reilly, USA, 4th edition, 2000.

[23] S. Parsons, C. Sierra, and N. R. Jennings. Agents that Reason and Negotiate by Arguing. *Journal of Logic and Computation*, 8(3):261–292, 1998.

[24] J. A. Rodriguez. *On the Design and Construction of Agent-mediated Electronic Institutions*. PhD thesis, IIIA-CSIC, Bellaterra, Catalonia, Spain, 2001.

[25] C. Sierra, J. Sabater, J. Agustí, and P. Garcia. Evolutionary Computation and Multiagent Systems Design. In *Proc. 1st Int'l Joint Conf. on Autonomous Agents & Multi-Agent Systems (AAMAS 2002)*, Bologna, Italy, 2002. ACM, U.S.A.

[26] V. S. Subrahmanian, P. Bonatti, J. Dix, T. Eiter, S. Kraus, F. Ozcan, and R. Ross. *Heterogeneous Agent Systems*. MIT Press/AAAI Press, Cambridge, MA, 2000.

[27] M. Tsvetovatyy and M. Gini. Towards a Virtual Marketplace: Architecture and Strategies. In *Procs. 1st Int'l Conf. on the Practical Applic. of Intell. Agents & Multi-Agent Technology (PAAM'96)*, Blackpool, UK, 1996. Practical Application Co.

[28] W. W. Vasconcelos. Logic-Based Electronic Institutions. SLIE Technical Paper, 2002. Available at `http://www.dai.ed.ac.uk/ groups/ssp/slie`.

[29] W. W. Vasconcelos and E. X. Meneses. A Practical Approach for Logic Program Analysis and Transformation. In *Proc. MICAI'00*, volume 1793 of *LNCS*. Springer-Verlag, 2000.

[30] W. W. Vasconcelos, D. Robertson, J. Agustí, C. Sierra, M. Wooldridge, S. Parsons, C. Walton, and J. Sabater. A Lifecycle for Models of Large Multi-Agent Systems. In *Proc. 2nd Int'l Workshop on Agent-Oriented Soft. Eng. (AOSE-2001)*, volume 2222 of *LNCS*. Springer-Verlag, 2001.

[31] W. W. Vasconcelos, J. Sabater, C. Sierra, and J. Querol. Skeleton-based Agent Development for Electronic Institutions. In *Proc. 1st Int'l Joint Conf. on Autonomous Agents & Multi-Agent Systems (AAMAS 2002)*, Bologna, Italy, 2002. ACM, U.S.A.

[32] M. Wooldridge and N. R. Jennings. Intelligent Agents: Theory and Practice. *The Knowledge Engineering Review*, 10(2):115–152, 1995.