Annals of Mathematics and Artificial Intelligence **41:** 135–169, 2004.

Rapid prototyping of large multi-agent systems through logic programming

W. Vasconcelos [a,*] , D. Robertson [b] , C. Sierra [c] , M. Esteva [c] , J. Sabater [c] and M. Wooldridge [d]

[a] *Department of Computing Science, University of Aberdeen, Aberdeen AB24 3UE, United Kingdom*
E-mail: wvasconcelos@acm.org
[b] *Centre for Intelligent Systems and their Applications (CISA), Division of Informatics, University of Edinburgh, Appleton Tower, Crichton Street, Edinburgh EH8 9LE, United Kingdom*
E-mail: dr@inf.ed.ac.uk
[c] *Artificial Intelligence Research Institute (IIIA), Consejo Superior de Investigaciones Científicas (CSIC), Campus UAB, 08193, Bellaterra, Catalonia, Spain*
E-mail: {siera,marc,jsabater}@iiia.csic.es
[d] *Department of Computer Science, University of Liverpool, Liverpool L69 7ZF, United Kingdom*
E-mail: mjw@csc.liv.ac.uk

Prototyping is a valuable technique to help software engineers explore the design space while gaining insight on the dynamics of the system. In this paper, we describe a method for rapidly building prototypes of large multi-agent systems using logic programming. Our method advocates the use of a description of all permitted interactions among the components of the system, that is, the *protocol*, as the starting specification. The protocol is represented in a way that allows us to automatically check for desirable properties of the system to be built. We then employ the same specification to synthesise agents that will correctly follow the protocol. These synthesised agents are simple logic programs that engineers can further customise into more sophisticated software. Our choice of agents as logic programs allows us to provide semi-automatic support for the customisation activity. In our method, a prototype is a protocol with a set of synthesised and customised agents. Executing the prototype amounts to having these agents *enact* the protocol. We have implemented and described a distributed platform to simulate prototypes.

1. Introduction

Rapid prototyping offers a means to explore essential features of a proposed system [9,27,35], promoting early experimentation with alternative design choices and allowing engineers to pursue different solutions without efficiency concerns [9]. In [25] we find reports of many successful experiments of rapid prototyping. Multi-agent systems (MASs, for short) are harder to design than centralised systems [65] and tools and methods to support the development of MASs are in urgent need [30].

* Corresponding author.

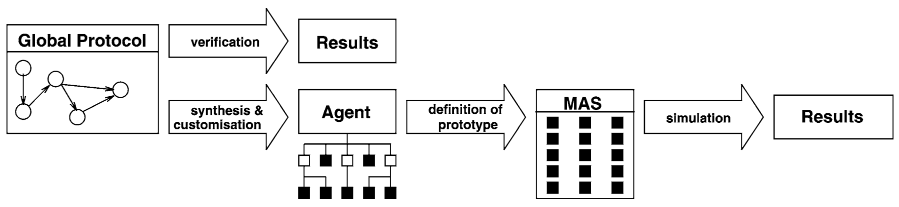


Figure 1. Overview of proposed method for rapid prototyping.

In this paper we describe a method for rapidly building prototypes of large MASs using logic programming. 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 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. Our approach to prototyping MASs reflects the modelling methodology introduced in [60] 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 form of non-deterministic finite state machines, called *electronic institutions* (or simply e-institutions) [18,39]. We explain more about this in section 2.

2. *Synthesis and Customisation of Agents* – this step addresses the automatic synthesis of agents as logic programs complying with the global protocol. Although simple, these synthesised agents are in strict accordance with the protocol from which they originate: 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, using logic program transformation techniques. 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. In section 4 we describe this step.

4. *Simulation and Monitoring of the Prototype* – the last step is the simulation of the prototype and the collection of results. We offer means for the enactment of an electronic institution: the agents are started as self-contained and asynchronous processes that communicate by means of message-passing. This step is described in section 5.

These steps are illustrated in figure 1. In the diagram, we also included a verification activity: the same specification for the global protocol can be used to check for desirable properties (or the absence of undesirable properties) – we explain this step in section 2.

We describe the steps above in sections 2–5. We compare our approach with related work in section 6 and in section 7 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 [53] where agents come to buy and sell goods. Our virtual marketplace, much like the Kasbah system depicted in [11], will be populated by agents started 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 developed in isolation, it is frequently impossible to predict the overall behaviour of the system *a priori*: its behaviour can only be understood through empirical investigation. Furthermore, to gain an insight into 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.

2. Global protocols via electronic institutions

A defining property of a MAS is the *communication* among its components: a MAS can be understood in terms of the kinds and order of messages its agents exchange [65]. We adopt the view that the design of MASs should thus start with the study of the exchange of messages, that is, the *protocols* among the agents, as explained in [60]. Such protocols are called *global* because they depict every possible interaction among all components of a MAS. The ultimate goal of our approach is to use the protocol specification to synthesise the individual components of a MAS and then run them (as explained below). The kinds and order of messages exchanged among the components of the system are all explicitly represented, and give rise to the actual agents that will ultimately enact the protocol.

Our global protocols are represented using *electronic institutions* (e-institutions, for short) [18,39]. E-institutions are a variation of *non-deterministic finite state machines* [29] (NDFSM, for short). An advantage of 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 (other than final states) which the system reaches 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 might 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 3.2 below. Again, more sophisticated notations 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 [18,46]. 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 as well as a shared ontology which allow them to interact and understand each other. We first define a scene:

Definition 1. 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 a finite, non-empty 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.

A scene is a protocol specified as a finite state machine where the states represent the different stages of the conversation and the directed edges connecting the states are labelled with messages of the communication language. A scene has a single initial state (non-reachable from any other state) and a set of final states representing the different possible endings of the conversation. There should be no edges connecting a final state to any other state. Because we aim at modelling multi-agent conversations whose set of participants may dynamically vary, scenes allow agents to join or leave at particular states during an ongoing conversation, depending on their role[1]. For this purpose, we differentiate for each role the sets of access and exit states.

[1] It is worth pointing out that roles in e-institutions are more than labels: they help us abstract from individual agents and define a pattern of behaviour that any agent that adopts a role ought to conform to. Moreover, all agents with a same role are guaranteed the same rights, duties and opportunities [18].

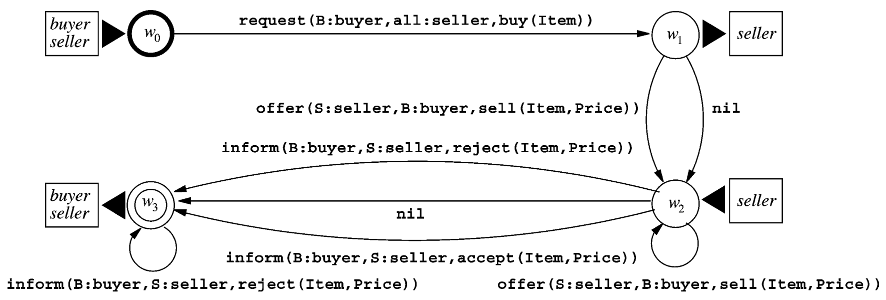

buyer
seller
request(B:buyer,all:seller,buy(Item))
seller
offer(S:seller,B:buyer,sell(Item,Price))
nil
inform(B:buyer,S:seller,reject(Item,Price))
buyer
seller
nil
seller
inform(B:buyer,S:seller,accept(Item,Price))
inform(B:buyer,S:seller,reject(Item,Price))
offer(S:seller,B:buyer,sell(Item,Price))

Figure 2. Simple Agora Room scene.

To illustrate this definition, in figure 2 we provide 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 the offers from sellers (if any) and chooses the best (cheapest) of them. The simplicity of this scene is deliberate, in order to make the ensuing discussion and examples more accessible. A more friendly visual rendition of the formal definition is employed in the figure. Two roles, buyer and seller, are defined. The initial state w_0 is denoted by a thicker circle (top left state of scene); the only final state, w_3, is represented by a pair of concentric circles (bottom left state). Access states are marked with a "▶" pointing towards the state with a box containing the roles of the agents that are allowed to enter the scene at that point. Exit states are marked with a "▶" pointing away from the state, with a box containing the roles of the agents that may leave the scene at that point. The edges are labelled with the messages to be sent/received at each stage of the scene. A special label "nil" has been used to denote edges that can be followed without any action/event.

We now provide a definition for e-institutions:

Definition 2. 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 all exit states $WE^{\mathbf{S}}$ of every scene $\mathbf{S}$ to some transition T, and $E^O \subseteq T \times WA^{\mathbf{S}}$ is a set of edges connecting all transitions to an access state $WA^{\mathbf{S}}$ of some scene $\mathbf{S}$;
- $\lambda_E : E \mapsto p(x_1, \ldots, x_k)$ 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 with other agents. We illustrate the definition above with an example comprising a complete virtual agoric market. This e-institution

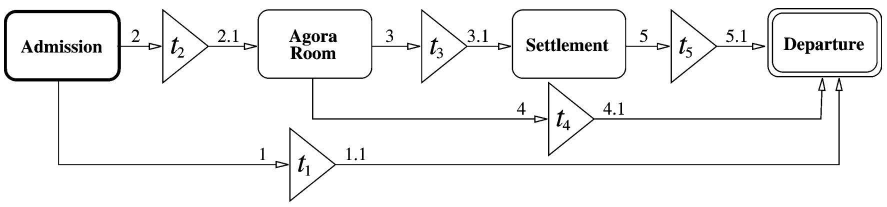


Figure 3. E-institution for simple agoric market.

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. In figure 3 we show a graphic rendition of 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 triangles. The arcs connect exit states of scenes to transitions, and transitions to access states. The labels of the arcs have been represented as numbers. The same e-institution is, of course, amenable to different visual renditions.

The predicates $p(x_1, \ldots, x_k)$ labelling the arcs, shown above as numbers, typically represent constraints on roles that agents ought to have to move into a transition, how the role changes as the agent moves out of the transition, as well as the number of agents that are allowed to move through the transition and whether they should synchronise their moving through it. In the agoric market in figure 3, the arc label 3 is:

$$p_3(x, y) \leftarrow id(x) \wedge role(y) \wedge y \in \{seller, buyer\} \wedge \langle x, y\rangle \in Ags \tag{3}$$

that is, transition t_3 is restricted to those agents x whose role y is either *seller* or *buyer* – information on such agents is recorded in the set Ags. The complementary arc label 3.1 leaving transition t_3 is:

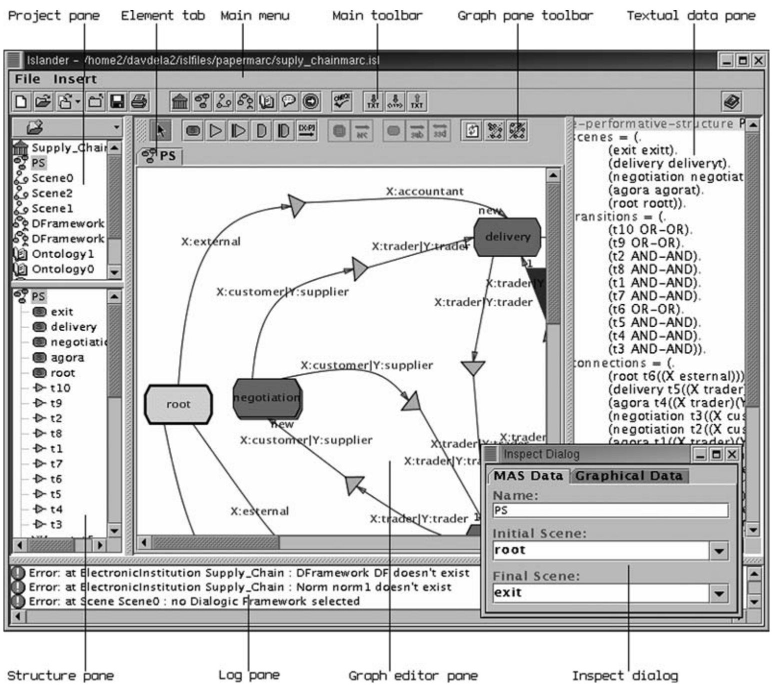
$$p_{3.1}(x, z) \leftarrow \langle x, y\rangle \in Ags \wedge y/z \in \{seller/payee, buyer/payer\} \tag{3.1}$$

that is, those agents $\langle x, y\rangle \in Ags$ that moved into t_3 may move out of the transition provided they change their roles: *seller* agents in the **Agora Room** scene should become *payee* agents in the **Settlement** scene, *buyer* agents should become *payer* agents.

2.1. Designing and representing e-institutions

Those wishing to design their own e-institutions can make use of a graphical editor, Islander [16,37]. Users can prepare their e-institutions by drawing diagrams as figures 2 and 3 using a selection of icons and a repertoire of drawing operations. The graphical notation is a means to present the formal definitions above and allow their more ergonomic manipulation. We show in figure 4 a screenshot of Islander.

Graphically represented e-institutions are translated into a logical formalism [61] implemented in Prolog [3], making our representation computer-processable. This makes it easier to synthesise our simple agents, as we shall see below. We show in figure 5 our Prolog representation for the agora room scene graphically depicted in figure 2



```
e-performative-structure P
scenes = (.
        (exit exitt).
        (delivery deliveryt).
        (negotiation negotiat
        (agora agorat).
        (root roott)).
ransitions = (.
        (t10 OR-OR).
        (t9 OR-OR).
        (t2 AND-AND).
        (t8 AND-AND).
        (t1 AND-AND).
        (t7 AND-AND).
        (t6 OR-OR).
        (t5 AND-AND).
        (t4 AND-AND).
        (t3 AND-AND)).
connections = (.
        (root t6((X esternal)))
        (delivery t5((X trader
        (agora t4((X trader)(Y
        (negotiation t3((X cus
        (negotiation t2((X cus
        (agora t1((X trader)(Y
```

Name: PS
Initial Scene: root
Final Scene: exit

```
Error: at ElectronicInstitution Supply_Chain : DFramework DF doesn't exist
Error: at ElectronicInstitution Supply_Chain : Norm norm1 doesn't exist
Error: at Scene Scene0 : no Dialogic Framework selected
```

Figure 4. Islander graphical editor for e-institutions.

```
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 5. Representation of agora room scene.

above. Each component of the formal definition has its corresponding representation. Since many scenes may coexist 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 `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

```
scenes([admission,agora,settlement,departure]).
transitions([t1,t2,t3,t4,t5]).
root_scene(admission).            output_scene(departure).
arc([admission,w3],$p_1$,t1).       arc(t1,$p_{1.1}$,[departure,w0]).
arc([admission,w3],$p_2$,t2).       arc(t2,$p_{2.1}$,[agora,w0]).
arc([agora,w3],$p_3$,t3).           arc(t3,$p_{3.1}$,[settlement,w0]).
arc([agora,w3],$p_4$,t4).           arc(t4,$p_{4.1}$,[departure,w0]).
arc([settlement,w3],$p_5$,t5).      arc(t5,$p_{5.1}$,[departure,w0]).
```

Figure 6. Representation of agoric market e-institution.

tuple comprising its formal definition. In figure 6 we present a Prolog representation for the agora market e-institution. Of particular importance are the arcs connecting scenes to transitions and vice-versa. In definition 2 arcs E are defined as the union of two sets $E = E^I \cup E^O$, E^I connecting (exit states of) scenes to transitions, and E^O connecting transitions to (access states of) scenes. We represent the E^I arcs as `arc`/3 facts the first argument of which holds (as a list) a scene and one of its exit states, the second argument holds the predicate (constraint) p_i which enables the arc, and the third argument is the destination transition. For simplicity, we chose to represent the arcs of E^O also as `arc`/3 facts, but with different arguments: the first argument holds the transition, the second argument holds the constraint that enables the arc, and the third argument holds (as a list) a scene and one of its access states.

2.2. *Checking properties of e-institutions*

Scenes and transitions are means for breaking up complex interactions of a MAS in a natural way. They can be seen as modules that can be combined together, provided some conditions hold. Complex interactions should be split into smaller parts with a coherent meaning: for instance, the part relating to admission, the part relating to the actual selling and buying, and so on. An immediate benefit in breaking up the interactions of a complex MAS is that its design becomes more manageable. Additionally, modules encourage reuse.

Checking properties automatically is an integral part of the formal specification of computer systems [14,21]. The modular description of a complex MAS as scenes and transitions allows useful checks to be performed with lower associated costs. The decomposition of a complex protocol into sub-protocols helps deter the multiplication of combinations of possible outcomes: a scene that has been checked for some property will not be affected by the properties of any transition connected to it. Furthermore, once a scene is checked, it need not be checked again when new parts are added to the specification.

Our representation renders itself to straightforward automatic checks for well-formedness. For instance, we can check whether all `theta`/2 terms are indeed defined with elements of `states`/3, whether all `arc`/3 are defined either for `access_states`/3 or `exit_states`/3, if all `access_states`/3 and `exit_states`/3 have their `arc`/3 definition, and so on.

```
1  connected_states(Sc,CSts):-
2    initial_state(Sc,W0), states(Sc,Sts), final_states(Sc,WFs),
3    setof(St,(member(St,Sts),path_states(Sc,W0,St,[])),RSts),
4    setof(RSt,(member(RSt,RSts),path_states(Sc,RSt,WFs,[])),CSts).

5  connected_scenes(CScs):-
6    root_scene(RSc), scenes(Scs), output_scene(FSc),
7    setof(Sc,(member(Sc,Scs),path_scenes(RSc,Sc,[])),RScs),
8    setof(Sc,(member(Sc,RScs),path_scenes(Sc,FSc,[])),CScs).
```

Figure 7. Fragment of program to check properties.

However, the representation is also amenable for checking important graph-related properties using standard algorithms [12]. It is useful to check, for instance, if from the `initial_state/2` we can reach all other `states/2`, whether there are `states/2` from which it is not possible to reach an `exit_state/3` (absence of *sinks*), and so on. We show in figure 7 a portion of a Prolog program to check for properties in e-institutions. Predicate `connected_states/2` (lines 1–4) obtains a list `CSts` of connected states in scene `Sc`, that is, those `states/2` that can be reached from the `initial_state/2` of the scene and from which a `path_states/4` to one of the `final_states` exists. This predicate works by finding all `states/2` `St` to which there is a `path_states/4` from `initial_state/2` `W0` (line 3) and then it tests (line 4) among these states, those from which a path to one of the `final_states/2` exists. Predicate `connected_scenes/1` (lines 5–8) returns a list `CScs` comprising all the scenes that can be reached from the `root_scene/1` and from which there is a path to the `output_scene/1`. Its operation is similar to the predicate `connected_states/2` just described.

Predicates `connected_states/2` and `connected_scenes/1` rely, respectively, on predicates `path_states/4` and `path_scenes/3`. The goal `path_states(Sc, St, FSt, Path)` holds if `Path` is a list of states representing a path between `St` and `FSt`, in scene `Sc`. Likewise, predicate `path_scenes(Sc, FSc, Path)` holds if `Path` is a list of scenes representing a path between scene `Sc` and scene `FSc`. Both `path_states/4` and `path_scenes/3` can cope with lists of final states/scenes, that is, they may also take as a parameter a list of final (destination) states/scenes and they hold if there is a path to one of the elements of this list. These predicates incorporate the usual transitive formulation [7,50] to find a next state/scene and then recursively find a path from this new state/scene to the destination, using the path built so far to avoid loops.

3. Synthesis and customisation 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 [61] we introduced a simple way to synthesise agents from our e-institutions. We devised a means to use the logical representation of the e-institution in order to obtain a set of Horn clauses which capture the behaviours for the agents participating in the e-institution. The synthesis obtains, for the roles of each scene, a set of Horn clauses which represent the connections among the states and the events, i.e., sending or receiving messages, associated with these edges.

The basic idea in this step is to automatically extract from an e-institution an account of the behaviours agents ought to have. This simplified account is called a *skeleton*: it provides the essence of the agents to be developed. Our skeletons are devised from the interaction specification (at the e-institution level) being much simpler to read than full agent descriptions, thus encouraging their use as the initial design for sophisticated reasoning agents. Engineers willing to develop agents to perform in e-institutions could then be offered a skeleton which would be gradually augmented into a complete program. Depending on the way skeletons are represented, semi-automatic support can be offered when augmenting them into more complex programs.

Skeletons should ideally exist in a computer-processable format, by which we mean that the behaviours represented by them should be reproducible by a computer. This way we do not have to perform further transformations from an abstract format onto more computationally-oriented representations – skeletons, after all, should guide designers in the development of their agents. Our skeletons are simple logic programs: the terse syntax, the precise declarative and procedural meanings and the ease with which one can write meta-programs to obtain alternative executions are some of the advantages of our approach. We explain more about the connection between skeletons and e-institutions in section 3.1, and in section 3.2 we show how they can be synthesised.

A skeleton should define all the basic behaviours agents should possess to successfully perform in the e-institution they are designed for. Our skeletons are simple logic programs with very limited functionality: they store the current state of the computation, and are able to move on to a next state, given certain conditions. However, e-institutions are non-deterministic and there might be states of the computation from which more than one next state is possible. When a rational agent follows an e-institution, any non-determinism should be resolved by formal reasoning and decision-making procedures. The augmenting process which skeletons undergo is aimed at "filling in" such capabilities. Reasoning and/or decision-making procedures have to be appropriately added to the initial skeleton, yielding more sophisticated agents that conform to the e-institution from which they were extracted. Furthermore, any variation to be performed by the components (such as the customisation of messages) is not specified in the e-institution. If, for instance, a message offering an item is to be sent, the actual item which is offered is to be defined by whichever agent actually participates in the e-institution. This variability is another capability that ought to be added to the initial skeleton.

Our choice of logic programs to represent skeletons is also supported by the wealth of research and results on automatic support and environments for logic programming development [5,15,23,58]. Of particular importance to our proposal is the work on the systematic approach to logic program development using *skeletons* and *programming techniques* [5,33,45,49]. With this approach, an initial simple program which defines the flow of execution (a *skeleton*) is augmented with more features (the *programming techniques*). These are extra computations to be performed as the flow of execution, defined by the initial skeleton, is followed. Since e-institutions prescribe the high-level flow of execution of a MAS there is a natural affinity between e-institutions and skeletons.

The activity of customisation of the skeleton is thus given support: programming techniques are added at the designer's will, conferring on the program additional capabilities. Program editing environments can be offered for this purpose, by which designers shift the focus of their attention: rather than seeing programs as sequences of characters (and adding or deleting them), programs are seen as "chunks" of constructs and operations over them. The addition of a parameter to a predicate, for instance, rather than requiring the appropriate editing of lines and characters to include the new parameter (with the likely risk of missing out on recursive calls or predicates with multiple definitions), becomes one single command which adequately alters all relevant parts of the program. We explain in more detail the customisation activity in section 3.3.

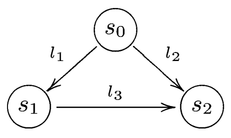
3.1. Skeletons as logic programs

Given an e-institution, we want to automatically extract essential information determining the behaviour of individual agents that will join in and interact with each other for some specific purpose(s). We shall call the representation for this essential information a *skeleton* of an agent.

The information obtained is to be used to restrict or define the possible behaviours of agents joining an e-institution. The same e-institution can be employed for this purpose, but we want simpler and more specialised versions aimed at the individuals that will populate the enactment of the e-institution. The simplification and/or specialisation of an e-institution, however, is in the sense of obtaining parts of the original NDFSM that are relevant for specific agents. This process hopefully yields a smaller NDFSM.

The information of a NDFSM can be efficiently represented with any of the classic data structures employed with graphs [12]. However, we need to add the dynamics of a *flow of execution* to the static information of states and transitions. This flow of execution captures the informal mechanism we use when we try to follow a NDFSM. NDFSMs are abstract models that can be given different computational interpretations [29]: the same automaton can be understood as a generator of correct output strings or as a device that accepts or rejects input strings. We also want to add to our representation some form of operational "meaning" of what happens when an edge is followed or triggered.

We propose logic programming for this purpose. The Horn clauses of logic programs are a compact formalism with precise declarative and procedural meanings. It provides a simple and natural means to represent our NDFSM as well as the flow of execution of such devices. Our proposal is exemplified in figure 8: a non-deterministic state transition diagram is shown with its associated clauses. The meaning associated to NDFSMs is the following: if $s \xrightarrow{l} t$ is an edge, then when the flow of the execution is in s, it should make l happen and move to state t; alternatively, the flow of execution



$$\begin{array}{lll}
state(s_0) & \leftarrow & trans(l_1) \wedge state(s_1) \\
state(s_0) & \leftarrow & trans(l_2) \wedge state(s_2) \\
state(s_1) & \leftarrow & trans(l_3) \wedge state(s_2)
\end{array}$$

Figure 8. State transitions and Horn clauses.

should *wait* until l happens and then it should move to state t. Both these possibilities can be captured with our Horn clause proposal: the appropriate definition of predicate *trans* that checks if a transition is enabled can give rise to the different meanings.

A more intuitive way to represent an edge $s \xrightarrow{l} t$ is via the clause $state(t) \leftarrow trans(l) \wedge state(s)$. This representation, however, fails to capture the temporal ordering between s and t, that is, the fact that the flow of execution can move from state s to state t if it can prove that $trans(l)$ holds. Clause $state(s) \leftarrow trans(l) \wedge state(t)$, on the other hand, capture this important relationship: if we use it to prove $\leftarrow state(s)$ applying SLDNF resolution [54], as implemented in Prolog [3], then the execution proceeds by (i) matching $\leftarrow state(s)$ with the head goal of the clause (i.e., flow of execution enters state s); (ii) an attempt to prove $trans(l)$ is made; and (iii) if $trans(l)$ is successfully proved, then the flow of execution tries to prove $state(t)$ – this step amounts to moving to state t. It is possible, though, to write a meta-interpreter [28,50] to use the more intuitive (and logically sound) representation and still capture the temporal ordering, but this would add complexity to our proposal.

Other forms of representation for NDFSMs such as adjacency matrices and adjacency (linked) lists [12] address separately the static information, i.e., the states and transitions, and the dynamic aspects of the model, i.e., how the static information is employed during computations with the NDFSM. Although such representations may be equivalent in terms of expressiveness or even more efficient in terms of storage space and retrieval speed, they are not so appropriate for our needs, that is, a minimal representation for a NDFSM which should be used as an initial design for a program.

There are other advantages in using clauses as a representation. The simplicity of this notation is complemented by the procedural meaning given by sound and complete proof procedures such as SLDNF resolution [54], efficiently implemented in different logic programming systems. If we assume this procedural interpretation, then it is enough to show the clauses that comprise our NDFSM. Our representation is thus an actual, albeit simple, logic program with a precise semantics. Given a NDFSM $M = (S, \Sigma, \delta, s_0, T)$, where $S = \{s_0, \ldots, s_n\}$ is the set of states (or vertices), $\Sigma = \{l_1, \ldots, l_m\}$ is the set of labels of transitions (we have used the more generic term "label of transitions" instead of an alphabet, as is the case in automata [29]), $\delta : S \times \Sigma \mapsto S$ is a (partial) transition function, $s_0 \in S$ is a special state, the *initial state* and $T \subseteq S$ is the set of terminal (or acceptance) states, then we can provide an automatic translation to our clause representation. For any $s, t \in S$, $l \in \Sigma$, such that $\delta(s, l) = t$, then we have $state(s) \leftarrow trans(l) \wedge state(t)$ in our clause representation. Additionally, we can include clauses to record the initial and final states, completely defining a NDFSM.

```
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 9. Fragment of program to synthesise agents.

3.2. Synthesis of skeletons from e-institutions

In [61] we introduced 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 figure 9 part 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`*$_2$`)`. Depending on which kind of arc `cl_arc`/3 uses (cf. figure 6 and its discussion), that is, whether it uses an E^I arc (connecting a scene to a transition) or E^O (connecting a transition to a scene) then an appropriate clause is synthesised. The first clause of `cl_arc`/3 defines the format of clauses for E^I arcs; the second clause defines the format of clauses the E^O. 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.

We have adopted the general format `s(`*`LInfo`*$_1$`):-`*`Cond`*, `s(`*`LInfo`*$_2$`)` for the clauses of our synthesised agents. In *`LInfo`*$_i$ we keep a list with information on the agent's current state of computation: we aim at the minimum required information to uniquely define it. In *`Cond`* we represent the condition to be fulfilled in order for the agent to move from `s(`*`LInfo`*$_1$`)` to `s(`*`LInfo`*$_2$`)`. We also aim at simplicity, so we do not devise different clauses for distinct situations the agent is (within scenes, leaving a scene and moving into a transition, or leaving a transition and entering a scene). We could have devised different clauses for each such situation, but this would require different means to cope with them during their execution.

```
s([agora,w0,buyer]):-
  send(request(B:buyer,all:seller,buy(Item))),
  s([agora,w1,buyer]).
s([agora,w0,seller]):-
  rec(request(B:buyer,all:seller,buy(Item))),
  s([agora,w1,seller]).
...
s([agora,w3,seller]):-
  rec(inform(B:buyer,S:seller,reject(Item,Price))),
  s([agora,w3,seller]).
s([admission,w3,seller]):- holds(p1),s([t1,seller]).
...
s([t5,buyer]):- holds(p5.1),s([departure,w0,buyer]).
```

Figure 10. Synthesised agent from e-institution.

We store in $\mathtt{LInfo}_i$ a means to represent an agent's state of computation within an e-institution. Within a scene, the triple $\langle S, w, r\rangle$ where S is the scene, w is a state within S and r is the role the agent has adopted, is sufficient to uniquely identify an agent's state of computation – we encode this as the list [Sc, St, R]. Likewise, within a transition, the pair $\langle t, r\rangle$, where t is the transition and r is the role of the agent in the transition uniquely depicts when an agent is at a transition – we encode this as the list [T, R]. By using lists we can accommodate both cases using the same representation for a clause.

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($\mathtt{LInfo}_1$):-P($\mathtt{Label}$), s($\mathtt{LInfo}_2$), 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 (first clause of cl_theta/3 – predicate send/1 is employed in this case) or whether the receiver is of the same role (second clause of cl_theta/3 – predicate rec/1 is used instead). If the role is not the sender nor the receiver (clause 3 of cl_theta/3 – the exception of clause 1 and clause 2) then the assembled clause is of the form s($\mathtt{LInfo}_1$):-s($\mathtt{LInfo}_2$). 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 figure 10 some of the clauses synthesised from the e-institution of figure 3, represented as in figures 5 and 6. The top clauses depict the agora 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 of computation of an agent. As explained above, this list can either be of the form [Sc, St, R] or [T, R] where Sc is the name of the scene, St is the identification of a state within Sc, R is a role and T is a transition. Depending on the role of the agent, a suitable action send/1 or rec/1, to send and receive a message, respectively, is chosen

```
s([]).
s([X|Xs]):-
  test(X),
  s(Xs).
s([_|Xs]):-
  s(Xs).
```

•

```
t([]).
t([X|Ys]):-
  test(X),
  t(Ys).
t(Ys):-
  t(Ys).
```

=

```
p([],[]).
p([X|Xs],[X|Ys]):-
  test(X),
  p(Xs,Ys).
p([_|Xs],Ys):-
  p(Xs,Ys).
```

Figure 11. Skeleton • technique = program.

for the clause within a scene. By using the clauses with the standard SLDNF resolution mechanism [3] we get all possible behaviours of the agents in the e-institution.

3.3. Customising synthesised agents

The clauses synthesised from the e-institution describe all possible 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.

3.3.1. Programming with skeletons and techniques

Automatic programming [4] has been a long-term goal of computer science in general [31] and, in particular, software engineering: programs being obtained via the rigorous manipulation (i.e., by a computer) of intermediate formalisms. Programming is an activity that can be given different degrees of support: we can see a spectrum of possibilities, ranging from completely automatic programming environments through to the completely manual and unsupported text-editing scenario. Somewhere in between these two extremes lie the *programming assistants*. These are tools that support human programmers developing, reusing, documenting and maintaining their code [24,44].

Logic programming, with its terse syntax, concise semantics and formal underpinnings, is particularly suitable for such support tools. One particular approach incorporates the classic methodology proposed by Wirth [64] by means of which an initial simple program is gradually refined and customised to the user's needs. The initial program is a *skeleton* and the refinements are *techniques* added to it [5]. Logic program development can thus be seen as a *transformation* activity [58] in which legal operations on a program (adding techniques) must preserve desirable properties (e.g., termination) [43].

To illustrate the skeletons and techniques approach, using a particular form of logic programming, viz. Prolog [3], we present an example in figure 11. We show, on the leftmost box a skeleton `s/1`, to traverse a list and test for specific components – skeletons define the flow of execution to be followed [5,33] by the programs that incorporate it. The skeleton of the figure is shown being *augmented* with a technique `t/1` (middle box) which collects those components that satisfy a test. A technique augments the functionalities of a skeleton (or program): additional computations are performed as the flow of execution is followed. In our case, the resulting program `p/2` (right box) builds a list (second argument) with selected elements from the traversed list (first argument). The "•" operator appropriately joins the two fragments, making sure that the base-case (non-recursive) clauses appear together, and that the recursive clauses get "blended" correctly.

In order to match the respective recursive clauses together, the `X` variable appearing in both skeleton and technique ought to be the same – although a variable `X` appears both in the skeleton and in the technique, the scope of a variable in Horn clauses is the clause in which it appears [3]. The resulting program `p/2` joins the functionalities of the skeleton (a list is traversed and its items are tested for some property) and the technique (those items that fulfil the test for some property are assembled together as a list). The flow of control, that is, the program's execution, is defined by the skeleton, whereas the computations to be performed as the execution proceeds are added by the technique.

The above "•" operator stands for the low-level operations on the programming constructs that take place for a complete program to be built. Although substantial support can be offered [5,58] human intervention is still needed at points. For instance, in the example above it is required that the `test/1` predicate be defined in order to have a complete program. Even though a tool could offer a library of likely tests, users may still want to develop their own routines. Again, help could be offered when auxiliary predicates are being developed, and so on, until the program is complete. The programming activity is thus redefined: an initial skeleton is chosen from an existing collection and techniques are applied to gradually obtain a program with the desired flow of execution and that performs the expected computations.

3.3.2. An extensible techniques-based programming environment

It is possible to develop a programming environment incorporating the skeletons and techniques approach. In the case of Prolog, a high-level symbolic language that allows programs to be conveniently manipulated by a program written in the same language, this has been successfully done [5,58].

Techniques are represented in a declarative way, free of implementational detail or particular usage in mind. The process of applying a technique is independently defined. An immediate benefit of this approach is that techniques have a clean and concise presentation format that would enable both engineers of the tool and its future users to quickly recognise and understand them. An environment that makes such a distinction is defined in [58] – we have adapted that proposal for our purposes here. Techniques are represented as simple program transformation schemata [57,59]. These are rewrite rules with program "templates", that is, abstract constructs that stand for *classes of programs*. We show in figure 12 an example of a technique represented as a transformation. The $\vec{A}_i$ constructs stand for *vectors of arguments*, that is, a possibly empty sequence of terms. P and Q are meta-variables that abstract the actual predicate names. Construct c stands for a generic constant name and $f(x, y)$ for a generic functor with two arguments, the second one of which is recursively defined. A program transformation is defined: if a

$$\boxed{\begin{array}{l} P(\vec{A}_1, c, \vec{A}_2)\texttt{.} \\ P(\vec{A}_3, f(x,y), \vec{A}_4)\texttt{:-} \\ \quad Q(x), \\ \quad P(\vec{A}_5, y, \vec{A}_6) \\ P(\vec{A}_7, f(x,y), \vec{A}_8)\texttt{:-} \\ \quad P(\vec{A}_9, y, \vec{A}_{10}) \end{array}} \Rrightarrow \boxed{\begin{array}{l} P(\vec{A}_1, c, \texttt{[]}, \vec{A}_2)\texttt{.} \\ P(\vec{A}_3, f(x,y), \texttt{[}x\texttt{|Ys]}, \vec{A}_4)\texttt{:-} \\ \quad Q(x), \\ \quad P(\vec{A}_5, y, \texttt{Ys}, \vec{A}_6) \\ P(\vec{A}_7, f(x,y), \texttt{Ys}, \vec{A}_8)\texttt{:-} \\ \quad P(\vec{A}_9, y, \texttt{Ys}, \vec{A}_{10}) \end{array}}$$

Figure 12. Programming technique represented as a rewrite rule.

program matches the left-hand side schema, then it can be rewritten as the right-hand side. On the right-hand side schema, arguments are appropriately added to the program, with the same effect of the programming technique of figure 11 above. The application of transformations such as the one above is precisely defined by a semi-unification algorithm [59]. Actual programming constructs are matched against the schematic constructs. This match will yield the new program with the prescribed added parts. Additional constraints on the schematic constructs can be defined, so as to narrow the possible matches.

We have developed an extensible programming environment using the above proposal. The environment is given an e-institution from which an initial skeleton is synthesised. This initial skeleton is then customised in different ways by the user. We are able to represent a comprehensive repertoire of program manipulation operations organised in the following three categories (in increasing order of complexity of captured programming expertise):

– *Program editing* – operations such as insert/delete an argument in a predicate, insert/delete goal in a clause, insert/delete clause in a program, and so on.
– *E-institution editing* – operations to *restrict* the clauses to specific scenes, states of a scene, transitions and roles. Such operations take into account the inter-dependence of concepts within the e-institution; for instance, if an agent has access to scene $\mathbf{S}_1$ then it may also need to have access to scene $\mathbf{S}_2$; if the user tried to restrict the clauses to scene $\mathbf{S}_1$, a message would be issued.
– *Program techniques* – insertion of extra functionalities with a coherent meaning/purpose, such as pairs of accumulators to carry values around, building recursive data structures, and so on [50].

These operations require user intervention in order to be properly applied. Users must determine where an argument is to be inserted, which transition, scene, or role is to be removed from the program being built, and so on. Our environments also offer the means to perform manual editing: the users are presented with the code for the program in a text editor and they can alter the program in whichever way wanted; when the users are finished, they select to save the changes and the program is stored with all the performed manual changes.

Our environment allows new program manipulation operations to be added as needed. Different presentations of programs and operations can be offered to the users,

```
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 13. Augmented agent.

such as a brief explanation in English or a visual representation. Program building is supplemented with means to run the devised code, debug and/or explain them, and have their efficiency analysed and improved [58].

We show in figure 13 the first two clauses of the synthesised 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 figure 10 the user gradually adds features to the agent's capabilities. We show the added parts underlined. The first modification inserts a programming technique which carries a `Stock` data structure around as the program 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 must 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, ensuring that agents will perform correctly and efficiently/intelligently.

3.3.3. Exploring and organising the design space

Given an e-institution $\mathcal{E}$ represented in the Prolog format explained in section 2.1 above, we can synthesise an initial skeleton Π, a fully operational, albeit simple, Prolog program. Our program editing environment offers an extensible repertoire of program manipulation operations (represented generically as "$\Rrightarrow$"), mapping a program to another program. The environment records the sequence of operations performed: this comprises the history of the preparation of a program. Users may backtrack to previous points in order to change their design decisions. We show in figure 14 a diagrammatic representation of this process: an e-institution $\mathcal{E}$ yields an initial skeleton Π which is operated upon via the $\Rrightarrow$ program manipulation operations, yielding programs Π_i. The design space is infinite: the program manipulating operations are means to explore this design space for the agents that will perform in the e-institution. The histories of different agents are means to organise the explored design space.

$$\begin{tikzcd}
\mathcal{E} \arrow[r, "synthesis"] & \Pi \arrow[r, Rrightarrow] \arrow[dr, Rrightarrow] & \Pi_{[1]} \arrow[r, Rrightarrow] & \Pi_{[1,1]} \arrow[r, Rrightarrow] & \Pi_{[1,1,1]} \arrow[r, Rrightarrow] & \cdots \\
 & & \Pi_{[2]} \arrow[r, Rrightarrow] & \Pi_{[2,1]} \arrow[r, Rrightarrow] & \Pi_{[2,1,1]} \arrow[r, Rrightarrow] & \cdots \\
 & & \cdots & & &
\end{tikzcd}$$

Figure 14. Exploring and organising a design space.

```
meta((G,Gs)):-
  meta(G), meta(Gs).
meta(G):-
  system(G), call(G).
meta(G):-
  setof(Body,clause((G:-Body)),Bodies),
  chooseBody(Bodies,ChosenBody),
  meta(ChosenBody).
```

Figure 15. Meta-interpreter for agents in Prolog.

3.3.4. Coping with non-determinism using meta-interpreters

Besides the customisation described above, another 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 [3], 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 a 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 [28,51] – 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 figure 15 a simple meta-interpreter for our needs. Its two initial clauses are the usual meta-interpreter definitions for conjuncts (first clause) and system built-ins (second clause) [3]. The third clause generalises the usual meta-interpreter definition to handle user-created predicates: those clauses `(G:-Body)` the head of which unifies with goal `G` are all collected and one of their bodies `Bodies` is chosen via predicate `chooseBody/2` as the `ChosenBody` that will be further used in the meta-interpretation. Predicate `chooseBody/2` must be defined by the agent's designer, and should reflect the policies and attitudes of the agent regarding non-deterministic choices.

Interestingly, the use of meta-interpreters to control the execution of agents may provide us with another way to approach agent development. In this approach, it is the meta-interpreter that is gradually augmented: the synthesised skeleton is left unchanged and techniques are instead applied to the meta-interpreter, augmenting its capabilities. One advantage of applying techniques to meta-interpreters is that these are normally compact pieces of code and hence are easier to alter and maintain. Another advantage lies in their potential for reuse: the same (augmented) meta-interpreter can be used with distinct skeletons from disparate e-institutions [61].

3.4. Working example: Agents for the agoric market

We now further develop our Agoric Market example to illustrate our approach and introduce other features of our environment. 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 only customisation in these scenes concerns the value of variables in sent messages: the synthesised skeleton may contain variables whose values await definition. The possible values for variables in messages can be determined using the underlying assumed ontology, but programmers must ultimately define how particular values will be assigned to such variables, via suitable predicate calls.

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* [19]. In our environment, the undefined parts are annotated with their possible values, as we explain below, and are used to prompt 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: these are associated with alternatives with descriptive labels which will be used as an interface to define a prototype.

We notice that the **Agora Room** scene 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 desire to sell each item (label 2 of figure 3). 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 in order to simplify our analysis.

We have employed the augmented clauses shown in figure 13 to prepare two kinds of agents, the seller and the buyer, by restricting their roles in the **Agora Room** scene – these are the entries `Seller` and `Buyer`, respectively, shown in figure 18. The alterations performed in both cases are very similar and are explained in detail below. Two arguments `Msgs` and `Stock`, storing, respectively, the messages exchanged and the

```
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).

designOption(predicate:greed/1,[greedy:greed(40),
                                 considerate:greed(10)]).
```

Figure 16. Fragment of `GenericSeller` agent.

stock of item, were consistently inserted in the clauses with predicate calls to manage them; these calls were further customised and provided with definitions.

3.4.1. A generic seller agent

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 to be used by the seller agents when assigning prices to items. 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 postpone the particular choices to a later stage. We show in figure 16 the clause of the `GenericSeller` agent, where the pricing is established as well as the definition of one of the auxiliary predicates and design options. 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 `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 w_2. 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` (marked with a “`$`” explained below) 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 `designOption/2` predicate highlights that `greed/1` is yet to be defined. When the user marks a programming construct with “`$`” our programming tool prompts

```
s([agora,w2,buyer],Stock,Msgs):-
  minimalOffers($minMsgs,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).

designOption(constant:minMsgs,[integer:(1,20)]).
```

Figure 17. Fragment of `GenericBuyer` agent.

her to specify what the construct is expected to be and what values it may have. This is then represented in the program itself via predicate `designOption`/2: its first argument states that a predicate `greed/1` awaits definition and its possible definitions are represented as a list (second argument of `designOption`/2) of pairs *`Label:Definition`*. A more informative label, such as `greedy` and `considerate`, can thus be associated to a definition. The labels are used to automatically synthesise an interface to the parameter-tuning of our prototypes, as explained below.

3.4.2. A generic buyer agent

We have noticed that the **Agora Room** scene also allows for the customisation of buyer agents. By examining the scene definition of figure 2, 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 part of the scene allows us to customise different kinds of buyer agents, depending on how we want them to 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 extremes define a spectrum of possibilities: if we denote by *n* 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 now proceed to customise the augmented clauses of figure 13, but this time the clauses defining the behaviour of the agent in the agora scene are restricted to the buyer role. In figure 17 we show a fragment of the buyer agent, viz., the clauses where the offers are collected (second clause of `s/3`) and one of them is accepted (first clause of `s/3`). We have engineered these clauses from those of figure 10, customised to meet our needs: two arguments, `Stock` and `Msgs` have been inserted (as in the example of figure 13) and predicates to manipulate them have been added; the remaining clauses simply pass these arguments down recursively.

Messages are received in the second clause until `minimalOffers/2` is satisfied – this predicate ensures that the agent has received the minimal number `minMsgs` offers. The programmer has, however, tagged `minMsgs` with a "`$`" to inform that the actual value for this constant is a parameter. Predicate `minimalOffers/2` counts the number of offers received and checks that it is above `minMsgs`. The programmer, having flagged `minMsgs` with "`$`" is prompted for its possible values. In the case above, the values have been specified as an integer between 1 and 20, stored in predicate `designOption/2`. Our environment uses predicate `designOption/2` to create a suitable interface to help users define the minimum number of messages a buyer agent must receive before deciding to buy.

4. Building MASs 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 parts 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.

Our prototypes are defined as collections of *populations of agents*. Designers select from among the programs Π_i obtained during the customisation stage those that will enact the e-institution and how many of each should make up the prototype. More formally,

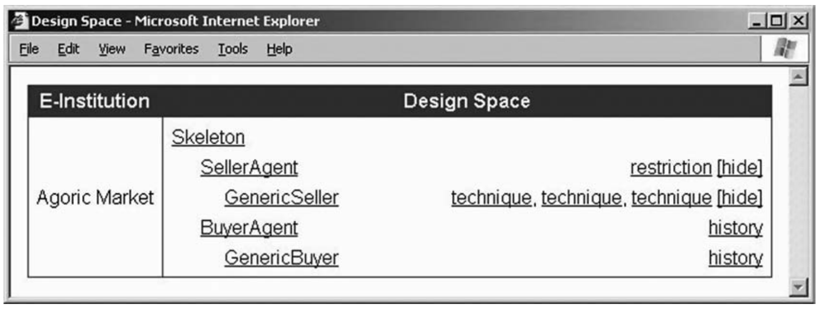
Definition 3. A population **Pop** is the pair $\mathbf{Pop} = \langle \Pi, n \rangle$ where Π is a program and $n \in \mathbb{N}$ is the number of individual copies of Π that will comprise the population.

This definition also caters for the case when a combination of a meta-interpreter Π and a synthesised/customised agent Π' is employed. In this case, we simply consider the union $\Pi \cup \Pi'$ of the clauses of the meta-interpreter and the clauses that make up the agent to be run. We now define a prototype for a MAS:

Definition 4. A prototype **P** is a pair $\mathbf{P} = \langle \mathcal{E}, \{\mathbf{Pop}_1, \ldots, \mathbf{Pop}_n\} \rangle$ where $\mathcal{E}$ is an e-institution and $\mathbf{Pop}_i$, $1 \leqslant i \leqslant n$, comprise a set of populations of agents.

In a prototype, the agents of the populations must all stem from the same e-institution $\mathcal{E}$. We explain below how this restriction is enforced when a prototype is assembled.

We have embedded the above concepts into an integrated environment for defining prototypes of MASs. We have developed different versions [62,63] of this environment


Design Space - Microsoft Internet Explorer
File Edit View Favorites Tools Help

E-Institution	Design Space	
Agoric Market	Skeleton	
	SellerAgent	restriction [hide]
	GenericSeller	technique, technique, technique [hide]
	BuyerAgent	history
	GenericBuyer	history

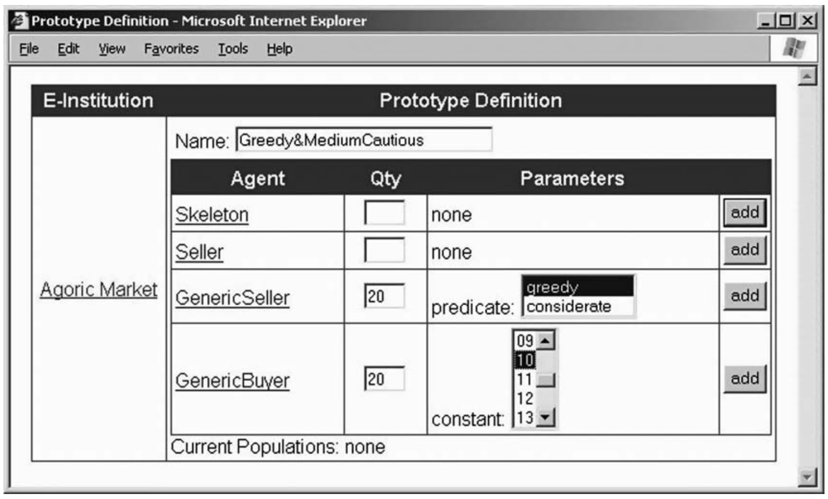
Figure 18. Screen shot of environment showing a design space.

which uses a simple HTML [38] interface to guide its users in the activity of building and running prototypes. Our environment receives as input e-institutions in the format of figure 6. The e-institutions have a unique identification which our environment uses to index the users' activities. The environment provides support for checking properties, for the synthesis and customisation of agents and the definition and execution of prototypes, as explained below. All these activities are dependent on an input e-institution. A number of distinct e-institutions can be input and exploited simultaneously without any risk of mixing them up or their agents and prototypes.

Users select one of the input e-institutions which is shown in a simple textual form with colours to differentiate its components. Users then proceed to check the e-institution for desirable properties: within our environment users may check e-institutions for their well-formedness, absence of unreachable scenes, unreachable states and sinks within scenes and unreachable transitions. The environment issues a report listing the problems detected to help designers fixing the offending parts of the e-institution.

When an error-free e-institution is obtained, users may proceed to the synthesis and customisation stage of the environment. The environment uses the representation of the e-institution to synthesise an initial skeleton for the agents which will enact the protocol, as explained in section 3.2. The skeleton is synthesised only once and stored with the e-institution, and is displayed in a simple text form with distinct colours to differentiate the information on the names of the e-institution, scene or transition, state and roles.

This initial skeleton is input to the customisation services, which aid users explore the design space, as explained in section 3.3. The kinds and order of operations applied to the skeleton and its ensuing versions are recorded to provide information on the history of the agent's construction. We present the explored design space in a simple textual form, showing a hierarchy of more specialised programs. Because the design space may grow considerably, we need to offer ways to hide details until they are needed: the history information is hidden until the user clicks on it. We show a screen shot of our environment in figure 18 below. The screen shot shows the Agoric Market e-institution with the currently explored design space for its agents. We use hyperlinks [38] to hide details – if users click on these, they are expanded and a "`[hide]`" option is offered to "tuck away" the information. We can see a hierarchy of agents rooted in `Skeleton`, the skeleton synthesised from the e-institution. Users give names to the partially developed


Prototype Definition - Microsoft Internet Explorer
File Edit View Favorites Tools Help

E-Institution	Prototype Definition			
Agoric Market	Name: Greedy&MediumCautious			
	Agent	**Qty**	**Parameters**	
	Skeleton		none	add
	Seller		none	add
	GenericSeller	20	predicate: greedy / considerate	add
	GenericBuyer	20	constant: 09 / 10 / 11 / 12 / 13	add
	Current Populations: none			

Figure 19. Console for Agoric Market prototypes.

programs: `SellerAgent`, `GenericSeller`, `BuyerAgent` and `GenericBuyer`. The development history of the programs is shown as a `history` hyperlink: upon clicking on it, users are presented with a sequence of names describing the operations applied to obtain the program on the left. These operation names, upon clicked, are further expanded. In the picture above, the `SellerAgent` has been obtained from `Skeleton` via a `restriction` operation. This operation allows parts of the original skeleton to be appropriately removed. In this case, we have restricted the skeleton to incorporate only the role of a seller in the Agora Room scene.

4.1. Prototyping agoric markets

We show in figure 19 a screen shot of our environment presenting the design space for a prototype definition. The options represented as `designOption`/2 predicates further refine the presentation of the agents. Designers assemble a prototype by defining and adding populations of agents: the field of column `Qty` represents how many individuals of that kind of agent should be in the population. Any extra parameters which require user customisation are offered in the `Parameters` column. The `add` buttons insert the respective population into the prototype definition. The `Parameters` column displays any parameters that must be set and their options. In our example, we have the greedy and considerate seller agent and the levels (1–20) of caution that a buyer agent can have.

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. All messages exchanged are recorded during the simulation, 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 family 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.

5. Simulation of prototypes

Our environment provides a means to rapidly build a MAS consisting of hundreds of agents. These agents can offer controlled levels of variability which allows the systematic exploration of the dynamics of the system. Prototypes can be reused: once it is defined, a prototype is stored and can be later retrieved and altered and put to use again. After a prototype is defined (or altered), we can simulate it, recording information about all messages exchanged among its agents. Agents may also leave "traces" of their execution behind, recording information on the tuple space. This information comprises the result of the simulation and can be displayed in different formats.

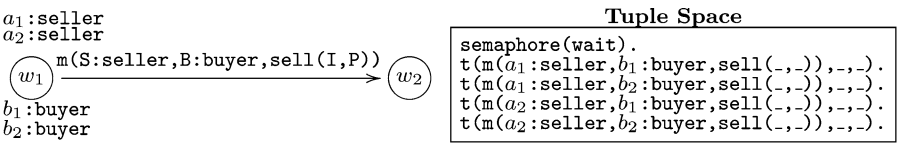
After a simulation, the parameters may then be changed and another simulation takes place, its results being recorded and adequately shown to the users. As designers get a better understanding of the MAS, they can find the adequate "tuning" for the features of the MAS, that is, the choice of parameters 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 [48] investigates the coupling of our method/environment with genetic programming to explore the parameters of a prototype.

5.1. Enacting e-institutions

We have incorporated the concepts above into a distributed simulation platform for e-institutions. This proof-of-concept platform, developed in SICStus Prolog [47], simulates an e-institution using a number of administrative agents, implemented as independent processes, to oversee the simulation. These administrative agents look after the agents taking part in the enactment which interact via a blackboard architecture, using the SICStus Linda tuple space [10,47].

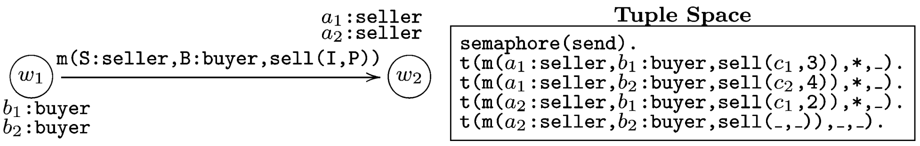
In order to develop the simulation platform, a suitable operational meaning for e-institutions, which is not part of the original proposal [18,39], had to be defined. The meaning of a scene is captured by those messages that are sent at each edge, as the agents participating in the scene move between states. A certain degree of synchronisation among those agents participating in the e-institution must be ensured.

An administrative agent *admScene* oversees the enactment of a scene, ensuring that the participating agents synchronise their states. This synchronisation is achieved via a semaphore in the tuple space which is updated by the *admScene* agent. We show



```
semaphore(wait).
t(m(a1:seller,b1:buyer,sell(_,_)),_,_).
t(m(a1:seller,b2:buyer,sell(_,_)),_,_).
t(m(a2:seller,b1:buyer,sell(_,_)),_,_).
t(m(a2:seller,b2:buyer,sell(_,_)),_,_).
```

Figure 20. Enactment of Agora Room scene (wait stage).



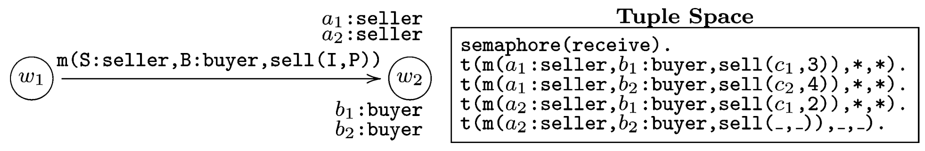
```
semaphore(send).
t(m(a1:seller,b1:buyer,sell(c1,3)),*,_).
t(m(a1:seller,b2:buyer,sell(c2,4)),*,_).
t(m(a2:seller,b1:buyer,sell(c1,2)),*,_).
t(m(a2:seller,b2:buyer,sell(_,_)),_,_).
```

Figure 21. Enactment of Agora Room scene (send stage).

in figure 20 a diagrammatic representation of edge (w_1, w_2) in the Agora Room scene previously shown in figure 2, its label and the participating agents a_1, a_2 (sellers) and b_1, b_2 (buyers) which have previously joined the scene – these agents are shown in state w_1. Our diagrammatic representation is purely for explanation purposes: when we write and show that an agent is in a state s it means that its internal representation for the current state is set to s.

On the right-hand side of the diagram above, we also show a fragment of the tuple space, managed by the *admScene* agent. The tuple space consists on the semaphore in the wait stage, and terms of the form t(*MsgTemplate*, *SendStatus*, *RecStatus*), which the *admScene* agent has recorded. These terms represent the messages *MsgTemplate* that can be sent by the agents. *SendStatus* and *RecStatus* are flags which indicate, respectively, when a message has been successfully sent and received; in such cases, the flag is set to "*". To ensure that an agent does not try to receive a message that has not yet been marked as sent but that may still be sent by some agent, the *admScene* agent synchronises the agents in the scene: it first lets the sending agents change state by moving along the corresponding edge, marking their messages as sent. When all sending agents have moved, then the *admScene* agent lets the receiving agents receive their messages and move to the next state of the scene.

The synchronisation among the agents of a scene is achieved via the semaphore term in the tuple space. The participating agents trying to send a message must wait until this semaphore has been set to "send" by the *admScene* agent. A participating agent sends a message by first checking if there is a matching *MsgTemplate* in the tuple space. The message is sent by assigning a value to any uninstantiated variables of the term in the tuple space (represented above as anonymous "_" variables) and assigning a "*" to the *SendStatus*. We show in figure 21 the same portion of the Agora Room scene after the semaphore has been set to "send": the seller agents a_1 and a_2 have sent their messages and thus moved to state w_2; the messages consist of goods c_1 and c_2 offered to the buyer agents at different prices; the *SendStatus* was adequately set to "*". Agent *admScene* creates templates for *all* messages that can be sent, but not all of them may in fact be sent: in our example, the last tuple was not used.



```
semaphore(receive).
t(m(a1:seller,b1:buyer,sell(c1,3)),*,*).
t(m(a1:seller,b2:buyer,sell(c2,4)),*,*).
t(m(a2:seller,b1:buyer,sell(c1,2)),*,*).
t(m(a2:seller,b2:buyer,sell(_,_)),_,_).
```

Figure 22. Enactment of Agora Room scene (`receive` stage).

Finally, the participating agents may receive messages, as the *admScene* agent sets the semaphore value to "`receive`". The participating agents receive a message by checking for a term in the tuple space that has been marked as sent and whose *MsgTemplate* matches the agents' message format. We show in figure 22 the Agora Room after agents b_1 and b_2 have received their messages and have moved to state w_2. The tuples have been marked as received, with *RecStatus* being set to "`*`".

The participating agents are the agents previously synthesised and customised. They are implemented as independent processes with their own thread of control. They access the tuple space in a distributed fashion, assigning values to variables (that is, sending messages) and retrieving values (that is, receiving messages), making use of predicates defined via SICStus Prolog Linda built-ins [47]. The participating agents inform the *admScene* agent, also via the tuple space, of the state of the scene they are currently at. With this information the *admScene* agent is able to "herd" agents from one state to another, as it creates messages templates, lets the sending agents mark them as sent and then lets the receiving agents mark them as received (also retrieving their contents). Those agents that do not send nor receive can move between states without having to wait for the semaphore. All agents though synchronise at every state of the scene, that is, there is a moment in the enactment when all agents are at state w_i, then after sending and receiving (or just moving) they are all at state w_{i+1}.

An enactment of an e-institution begins with the enactment of the root scene and terminates when all agents leave the output scene. Engineers may specify whether a scene can have many instances enacted simultaneously, depending on the number and order of agents willing to enter it. We did not include this feature in our formal presentation because instances of a scene can be understood as different scenes: they are enacted independently from each other, although they all conform to the same specification. When a scene terminates, there is a record in the tuple space of all messages that were exchanged as a result of the scene's enactment. This is useful for following the dynamics of the e-institution.

Transitions are enacted in a similar fashion. The platform assigns an agent *admTrans* to look after each transition. However, transitions differ from scenes in two ways. Firstly, we do not allow instances of transitions. This is just a methodological restriction, rather than a technical one: we want transitions to work as "meeting points" for agents moving between scenes and instances of transitions could prevent this. Secondly, transitions are *permanent*, that is, their enactment never terminates. Scenes (or their instances), once enacted (i.e., all the agents have left it at an exit state), cease to exist, that is, the *admScene* agent looking after it stops.

6. Related work

Our approach has parallels with [36], where logic programming is also exploited to specify and simulate MASs. The approach of [36] 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. The work in [13] extends that of [36], allowing the design of heterogeneous and open MASs by incorporating a mediator system and a generic agent execution platform.

Computational logic has been widely advocated as a means to specify software, offering distinct ways to analyse the specification and, when needed and appropriate, to execute it [21]. Such features, exploited in the days of stand-alone, centralised systems, have also proved to be applicable to MASs [2,6,34]. 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 or appeal to everyone. In contrast, 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.

Logics have been used as a unifying formalism to represent, reason about, model and implement protocols in MAS. For instance, the work in [55,56] propose an alternative notation to specify many-to-many interactions among agents – the labels of edges of a variation of e-institutions are formulae of a special-purpose first-order logics; they are used to create models that can be checked for properties and also used in simulations. Another related work is [1] where protocols are represented as *social integrity constraints*, a restricted class of first-order logics. Special social integrity constraints which refer to events in the past (backward expectations) give rise to interaction protocols.

Our work has strong connections with agent-oriented software engineering [30] as we cater for prototyping, an important stage in the design of complex systems. There are similarities between ours and the work of [66] in that both try to bridge the gap between formal models and implementations of multi-agent systems. Moreover, both underlying formal models are finite state machines: the work in [66] uses G-nets, a type of high-level Petri net [41].

6.1. *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 [20,52]. 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 [40]. 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 [27] a survey of programming languages and systems for prototyping concurrent applications. The languages and systems surveyed 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 resembles that of [8], in which simple skeletons of programs are generated from special kinds of Petri Nets. However, in that work only simple interactions are addressed, whereby information is passed around distributed algorithms.

7. Conclusions and future work

In this paper we have described an approach for rapidly prototyping large multi-agent systems. 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 populations of the previously synthesised/customised agents to enact a global protocol.
4. Simulation and monitoring of the prototype.

After analysing 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 [61]: 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. Our approach also offers a means to check for global properties of the MASs to be built.

Our environment offers a means to carry out simulations of our prototypes. Each agent becomes a self-contained asynchronous process which communicates via a tuple space. The environment incorporates a platform to enact electronic institutions [55] with administrative agents which ensure 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 features 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 a means to automatically transform a prototype into an efficient implementation [27,35]. 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. However, the customisation stage is more difficult to achieve using the syntax of C or Java, as it involves the manipulation and alteration of a program. 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 their proxy.

We are currently investigating alternative formalisms for global protocols, combining NDFSMs with logics. There are a number of likely candidates both for the NDFSM and logics to label edges with. For the NDFSM, we are currently investigating Petri nets [41] and conceptual graphs [42], as well as a specially engineered electronic institutions. Likewise, for the logics we will investigate typed first-order logics and special-purpose logics with explicit sets [55]. The aim is a combination which is expressive and natural and with a clear and concise semantics that also allows properties to be proved as well as the synthesis (and customisation) of components and the enactment of the system described. We also want to bring into our initial specification and prototypes more features of e-institutions left out of this current work, such as normative rules and more general dialogic frameworks [17,18,39].

We have used our method to create prototypes for auction rooms [22,39] and negotiations [32,40], exploiting different organisational schemes such as supply networks [26]. We have built experiments with up to 250 individually customised agents each of which executed as a stand-alone Prolog process communicating via the shared tuple space. Our customisation process and support tool scales up naturally: the hierarchical relationships among agents will help designers concentrate on "families" or "species" of agents with similar behaviours and their differences. The number of individual agents that will define an instance of the MAS is a simple parameter that can be changed at will.

Once designers define a MAS, they can simulate its execution using our platform. During this simulation engineers abstract away individual agents and observe their collective behaviour. Engineers are encouraged to experiment with the parameters of their prototypes and how they affect the overall behaviour of the MAS. At the point of simulation, a MAS is a collection of independent Prolog programs that run in parallel – engineers need not and should not be looking at the actual programs comprising the individual components of the MAS.

Our method has also been put to use in an altogether different context than the one originally thought of: to create agent-based models for real-life scenarios. One of the models we developed simulates a breast cancer referral procedure: patients, general practitioners, specialists and scanning units are individual agents that interact with each other to exchange information. With this model, we are able to exploit different arrangements for patient referral, i.e., under what conditions should patients be referred to specialists, alternative protocols and procedures with the different components of the system, and alternative policies for handling waiting lists (queues, priority lists, and so on).

Acknowledgements

This work was partially sponsored by the European Union, under contract IST-1999-10208, research grant Sustainable Lifecycles in Information Ecosystems (SLIE), and by the Spanish CICYT project eINSTITUTOR (TIC2000-1414). Thanks are due to Chris Walton for proofreading earlier versions of this document and also for useful suggestions. We would also like to thank the anonymous referees for their useful comments that helped to improve this paper.

References

[1] M. Alberti, M. Gavanelli, E. Lamma, P. Mello and P. Torroni, Modeling interactions using social integrity constraints: A resource sharing case study, in: *Declarative Agents Languages and Technologies (DALT)*, Lecture Notes in Computer Science (Springer, 2004).

[2] 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) (2001).

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

[4] A.W. Biermann, Automatic programming, in: *Encyclopedia of Artificial Intelligence*, Vol. 1 (Wiley, New York, 1992).

[5] A.W. Bowles, D. Robertson, W.W. Vasconcelos, M. Vargas-Vera and D. Bental, Applying prolog programming techniques, International Journal of Human–Computer Studies 41(3) (1994) 329–350.

[6] 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* (Springer, 1999) pp. 5–32.

[7] I. Bratko, *Prolog Programming for Artificial Intelligence*, 3rd ed. (Longman Higher Education, 2000),

[8] G. Bucci and E. Vicario, Rapid prototyping through communicating Petri nets, in: *Proc. 3rd Int'l Workshop on Rapid System Prototyping*, ed. N. Kanopoulos, Los Alamitos, CA, USA (1994) pp. 58–75.

[9] R. Budde, K. Kuhlenkamp, L. Mathiassen and H. Züllighoven (eds.), *Approaches to Prototyping* (Springer, New York, 1984).

[10] N. Carriero and D. Gelernter, Linda in context, Communications of the ACM 32(4) (1989) 444–458.

[11] A. Chavez and P. Maes, Kasbah: An agent marketplace for buying and selling goods, in: *Proc. 1st Int'l Conf. on the Practical Applic. of Intell. Agents & Multi-Agent Technology (PAAM'96)*, Blackpool, UK (1996).

[12] T.H. Cormen, C.E. Leiserson and R.L. Rivest, *Introduction to Algorithms* (MIT Press, Cambridge, MA, 1990).

[13] P. Dart, E. Kazmierczak, L. Sterling, M. Martelli, V. Mascardi, F. Zini and V.S. Subrahmanian, Combining logical agents with rapid prototyping for engineering distributed applications, in: *Proc. 9th Software Tech. & Engineering Practice (STEP'99)*, Pittsburgh, PA (1999).

[14] M. d'Inverno and M. Luck, *Understanding Agent Systems* (Springer, Berlin, 2001).

[15] M. Ducassé and J. Noyé, Logic programming environments: Dynamic program analysis and debugging, Journal of Logic Programming 19, 20 (1994) 351–384.

[16] 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).

[17] M. Esteva, J. Padget and C. Sierra, Formalizing a language for institutions and norms, in: *Intelligent Agents VIII*, eds. M. Tambe and J.-J. Meyer, Lecture Notes in Artificial Intelligence, Vol. 2333 (Springer, Berlin, 2001).

[18] M. Esteva, J.-A. Rodríguez-Aguilar, C. Sierra, P. Garcia and J.L. Arcos, On the formal specification of electronic institutions, in: *Agent Mediated E-Commerce*, eds. F. Dignum and C. Sierra, Lecture Notes in Artificial Intelligence, Vol. 1991 (Springer, Berlin, 2001).

[19] P. Flener, K.-K. Lau, M. Ornaghi and J. Richardson, An abstract formalisation of correct schemas for program synthesis, Journal of Symbolic Computation 30(1) (2000).

[20] A. Franklin and A. Graesser, Is it an agent, or just a program?, in: *Intelligent Agents III* (Springer, Berlin, 1997).

[21] N.E. Fuchs, Specifications are (preferably) executable, Software Engineering Journal (1992) 323–334.

[22] P. Garcia, E. Gimenez, L. Godo and J.A. Rodriguez-Aguilar, *Bidding Strategies for Trading Agents in Auction-Based Tournaments*, Lecture Notes in Computer Science, Vol. 1571 (1999).

[23] T.S. Gegg-Harrison, Learning prolog in a schema-based environment, Instructional Science 20 (1991) 173–192.

[24] A.T. Goldberg, Knowledge-based programming: A survey of program design and construction techniques, IEEE Transactions on Software Engineering SE-12(7) (1986) 752–768.

[25] V.S. Gordon and J.M. Bieman, Rapid prototyping: Lessons learned, IEEE Software 12(1) (1995) 85–95.

[26] C.M. Harland, Supply chain management: Relationships, chains and networks, British Journal of Management 7 (1996) 63–80.

[27] W. Hasselbring, Programming languages and systems for prototyping concurrent applications, ACM Computing Surveys 32(1) (2000) 43–79.

[28] P.M. Hill and J. Gallagher, Meta-programming in logic progamming, in: *Handbook of Logic in Artificial Intelligence and Logic Programming*, Vol. 5 (1998) pp. 421–498.

[29] J.E. Hopcroft and J.D. Ullman, *Introduction to Automata Theory, Languages and Computation* (Addison-Wesley, Reading, MA, 1979).

[30] N.R. Jennings and M. Wooldridge, Agent-oriented software engineering, in: *Handbook of Agent Technology* (AAAI/MIT Press, 2000).

[31] W.H. Kautz, E.A. Voorhees and T.A. Jeeves, Automatic programming systems, Communications of the ACM 1(8) (1958) 6–8.

[32] G.E. Kersten and S. Szpakowicz, Negotiation in distributed artificial intelligence: Drawing from human experience, in: *Proc. 27th Annual Hawaii Int'l Conf. on System Sciences*, eds. J.F. Nunamaker and R.H. Sprague, Los Alamitos, CA (1994) pp. 258–270.

[33] M. Kirschenbaum, S. Michaylov and L. Sterling, Skeletons and techniques as a normative view of developing logic programs, Australian Computer Science Communications 18(1) (1996) 163–178.

[34] R.A. Kowalski and F. Sadri, From logic programming towards multi-agent systems, Annals of Mathematics and Articial Intelligence 25(3-4) (1999) 391–419.

[35] Luqi, Computer aided system prototyping, in: *Proc. 1st Int'l Workshop on Rapid System Prototyping*, Los Alamitos, CA (1992) pp. 50–57.

[36] 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, 1992).

[37] E-Institutor Project: Islander graphical editor, IIIA-CSIC, Bellaterra, Catalonia, Spain (2002), `http://e-institutor.iiia.csic.es/e-institutor/software/islander.html`.

[38] C. Musciano and B. Kennedy, *HTML & XHTML: The Definitive Guide*, 4th ed. (O'Reilly, 2000).

[39] P. Noriega, Agent-mediated auctions: The fishmarket metaphor, Ph.D. thesis, Institut d'Investigació en Intel.ligència Artificial (IIIA), Consejo Superior de Investigaciones Científicas (CSIC), Campus UAB, Bellaterra, Spain (1997).

[40] S. Parsons, C. Sierra and N.R. Jennings, Agents that reason and negotiate by arguing, Journal of Logic and Computation 8(3) (1998) 261–292.

[41] J.L. Peterson, Petri nets, Computing Surveys 9(3) (1977) 223–252.

[42] S. Polovina and J. Heaton, An introduction to conceptual graphs, AI Expert (1992) 36–43.

[43] M. Proietti and A. Pettorossi, Transformations of logic programs: Foundations and techniques, Journal of Logic Programming 19, 20 (1994) 261–320.

[44] C. Rich and Y.A. Feldman, Seven layers of knowledge representation and reasoning in support of software development, IEEE Transactions on Software Engineering 18(6) (1992) 451–469.

[45] D. Robertson, A simple Prolog techniques editor for novice users, in: *3rd Annual Conference on Logic Programming*, Edinburgh, Scotland (1991).

[46] J.A. Rodríguez-Aguilar, On the design and construction of agent-mediated electronic institutions, Ph.D. thesis, IIIA-CSIC, Spain (2001).

[47] SICS, SICStus Prolog user's manual, Swedish Institute of Computer Science (2000), `http://www.sics.se/sicstus`.

[48] 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).

[49] L. Sterling and M. Kirschenbaum, Applying techniques to skeletons, in: *Constructing Logic Programs* (Wiley, London, 1993).

[50] L. Sterling and E. Shapiro, *The Art of Prolog: Advanced Programming Techniques*, 2nd ed. (MIT Press, 1994).

[51] L.S. Sterling and R.D. Beer, Meta-interpreters for expert system construction, Journal of Logic Programming 6(1–2) (1989) 163–178.

[52] 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).

[53] M. Tsvetovatyy and M. Gini, Towards a virtual marketplace: Architecture and strategies, in: *Proc. 1st Int'l Conf. on the Practical Applic. of Intell. Agents & Multi-Agent Technology (PAAM'96)*, Blackpool, UK (1996).

[54] M.H. Van Emden and R.A. Kowalski, The semantics of predicate logic as a programming language, Journal of ACM 23(4) (1976) 733–742.

[55] W.W. Vasconcelos, Expressive global protocols via logic-based electronic institutions, in: *Proc. 2nd Int'l Joint Conf. on Autonomous Agents & Multi-Agent Systems (AAMAS 2003)*, Melbourne, Australia. A longer version appears as Tech. Report AUCS/TR0301, Department of Computing Science, University of Aberdeen, UK (2003).

[56] W.W. Vasconcelos, Logic-based electronic institutions, in: *Proc. Declarative Agent Languages and Technologies (DALT'03)*, Melbourne, Australia, Lecture Notes in Artificial Intelligence (Springer, 2003).

[57] W.W. Vasconcelos and N.E. Fuchs, An opportunistic approach for logic program analysis and optimisation using enhanced schema-based transformations, Lecture Notes in Computer Science, Vol. 1048 (Springer, 1996).

[58] W.W. Vasconcelos and N.E. Fuchs, Prolog program development via enhanced schema-based transformations, Technical report, Department of Artificial Intelligence, University of Edinburgh. Presented at the 7th Workshop on Logic Programming Environments, held in conjunction with ILPS'95, Seattle, USA (1996).

[59] W.W. Vasconcelos and E.X. Meneses, A practical approach for logic program analysis and transformation, Lecture Notes in Computer Science, Vol. 1793 (Springer, 2000).

[60] 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)*, Lecture Notes in Computer Science, Vol. 2222 (Springer, 2001).

[61] 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).

[62] W.W. Vasconcelos, C. Sierra and M. Esteva, An approach to rapid prototyping of large multi-agent systems, in: *Proc. 17th IEEE Int'l Conf. on Automated Software Engineering (ASE 2002)*, Edinburgh, UK (2002) pp. 13–22.

[63] W.W. Vasconcelos, C. Sierra and M. Esteva, An environment for rapid prototyping of large multi-agent systems, Technical report, `http://www.dai.ed.ac.uk/groups/ssp/slie`. Presented at AAMAS 2002 Demonstration Session, Bologna, Italy (2002).

[64] N. Wirth, Program development by stepwise refinement, Communications of the ACM 14(4) (1971) 221–227.

[65] M. Wooldridge, *An Introduction to MultiAgent Systems* (Wiley, London, 2002).

[66] H. Xu and S.M. Shatz, ADK: An agent development kit based on a formal design model for multi-agent systems, Automated Software Engineering 10 (2003) 337–365.