Modelling and simulation of social systems with INGENIAS

Juan Pavón*, Candelaria Sansores and Jorge J. Gómez-Sanz

Dep. Ingeniería del Software e Inteligencia Artificial
Universidad Complutense de Madrid
C/ Profesor José García Santesmases s/n, 28040 Madrid, Spain
E-mail: jpmavon@fdi.ucm.es
E-mail: csansores@fdi.ucm.es
E-mail: jjgomez@pip.ucm.es
*Corresponding author

Abstract: Most agent-based simulation toolkits are based on the Java programming language. This makes their use difficult for social scientists, who are usually not skilled in computer programming. However, agent modelling concepts are not unlike those which could be used for the modelling of social systems. This assumption is considered in proposing the use of a graphical agent-oriented language for the specification of social simulation models, and for transforming (automatically) these models to code for an agent-based simulation toolkit. In this manner, a social scientist could prepare social models in a more convenient way, and execute simulations on existing simulation toolkits, getting results back in terms of the model. This framework is built with a set of agent development tools, specifically, the INGENIAS Development Kit (IDK), which provides a customisable model editor and modules for automatic code generation.

Keywords: agent-based simulation; Agent-based Modelling; ABM; social simulation; agent development tools; model transformation; model-driven development; INGENIAS Development Kit; IDK.


Biographical notes: Juan Pavón-Mestras holds a PhD degree in Computer Science from the Universidad Politécnica de Madrid, Spain (1988) and since 1997 has been an Associate Professor at the Universidad Complutense de Madrid. He has been working in the R&D departments of Alcatel in Spain, France and Belgium, and in Bellcore, USA, involved especially in the development of software component architectures for distributed systems. Currently he is the leader of the grasia! research group at the Universidad Complutense de Madrid, with focus on agent-oriented software engineering and agent-based social modelling.

Candelaria Sansores has a degree in Software Engineering from the Instituto Tecnológico de Mérida, Yucatán, Mexico. She obtained her PhD in Computer Science from the Universidad Complutense de Madrid, Spain in 2007. She is a member of the grasia! research group on agent technologies. Her research is focused on agent-oriented software engineering and agent-based modelling and
1 Introduction

Agent-based Modelling (ABM) and simulation in the social sciences have offered researchers in this field a new methodology for the study of social phenomena characterised by a complex nature. Emergent behaviour and self-organisation produced by the collective action of social individuals in nonlinear dynamic systems are hard to understand and experiment with, but agent-based simulations have proved to be an appropriate tool for exploring these complexities (Schelling, 1971; Axtell, 2000; Macy and Skvoretz, 1998).

In this context, agents are used to model social worlds or artificial societies (Epstein and Axtell, 1996), focusing on how local interaction among agents serves to create larger and global social structures and patterns of behaviour. This model represents the individuals as autonomous social agents that perform actions in a simulated environment, the ability of an agent to interact with others being a social feature.

However, in the Agent-based Social Simulation (ABSS) domain, the specification concept is different from the computer science usage. Although the specification is also the description of the model, this description is mostly performed with a low-level programming language in the form of a source code. The model is the actual computer program, and the simulation is the running model. This is the dynamic representation of the social process under study (Gilbert, 1999). Besides, regarding these two stages, modelling and simulation of social systems, there is no consensus about an engineering process to be applied, and fundamental issues also arise, such as the facts that computational agents are rarely implemented, low-level programming tools are difficult to use for users who are not necessarily experts in computer programming, such as social scientists or economists, and that the Multiagent System (MAS) potential is not being leveraged.

Currently, there is a diversity of tools that intend to address some of these issues. For instance, Recursive Porous Agent Simulation Toolkit or RePast (Collier et al., 2003) is a set of Java libraries that allows programmers to build simulation environments, to specify properties and behaviours of agents that are possibly immersed in social networks, to collect data from simulations automatically, and to build user interfaces easily. RePast borrows many concepts from the Swarm toolkit (Minar et al., 1996), one of the earliest and most well-known toolkits for building agent-based models. Another similar library is Mason (Luke et al., 2003), a single-process discrete-event simulation core and visualisation toolkit written in Java, designed to be flexible enough to be used for a
wide range of simulations, but with a special emphasis on swarm simulations of a huge number (up to millions) of agents. The design philosophy of these toolkits is to provide a model library to which an experienced programmer can easily add features for simple simulations. For modellers, these libraries have great advantages over developing their own; however, it is usually difficult to specify social models with such tools. This is mainly because this task requires writing code in some usually object-oriented programming language (in most cases, Java). Thus, inexperienced programmers might find that a higher-level declarative language instead of a programming language would be desirable. This was the purpose of SDML (Moss et al., 1998), which was built on Smalltalk. Unlike RePast and Mason, it does not require users to be fluent in the underlying programming language, but they have to learn a complex interface that can be as difficult to master as a full programming language, which ultimately limits its usability.

Another kind of tool that has emerged for developing simulations is rapid development environments, for instance, Netlogo (Wilensky, 1999) and Python language in RePast Py. Although they are relatively easy to use because they support visual programming, agents in this kind of system are quite simple, usually with a poor or nonexistent agents’ cognitive model, and without support for modelling direct interactions between agents. In the end it is always necessary to have some programming skills to extend the basic behaviour libraries in the former or Python Scripts in the ladder.

One approach that seems to consider this requirement more formally is SeSAm (Klügl et al., 2004). This framework provides an environment for modelling that is based on UML-like activity diagrams. It attempts to facilitate model specification with graphical tools and a set of predefined behaviours as a kind of state machine. Although it goes a step forward in the high-level modelling issue, it still lacks the common functionality that is usually available in more general agent development tools, such as the ability to specify agents’ direct interactions via messages. Thus, interactions in such simulations are expressed indirectly through changes in the environment, mainly because this toolkit is based on simple reactive agents and not on sophisticated internal reasoning processes, which are prerequisites for complex negotiation abilities. Thus, it does not support complex goal-oriented agents.

On the other hand, agent-oriented software engineering offers powerful modelling languages, at a more abstract level. These languages are usually of a higher level, based on some graphical notation, and, in some cases, easily customisable. Their capabilities make them more suitable for depicting models than programming languages. Also, the supported agent model is richer, both at micro (agent) and macro (organisation) levels, than in the agent-based simulation toolkits. Furthermore, agent modelling concepts are closer to the abstractions that a sociologist or economist could use to model social systems. For instance, for modelling individual decision making and social behaviour, including social interaction, collaboration, group behaviour and the emergence of higher-order social structure, agent concepts such as the different computer science views of agency (which range from reactive decision rules to complex adaptive intelligence in the form of agents’ internal architecture), organisational issues, agents’ autonomy, relationships, communication languages and the environment (in which they interact) are well suited to represent these social aspects. For these reasons, we believe that these languages should be considered for modelling social systems rather than plain program code.
Accordingly, we propose to perform the ABSS in two main activities. First, the description of the model in a high-level language will be done. Describing social phenomena with a highly visual language eliminates implementation complexity for social scientists and is generally easier to use than a programming language. We are aware that having a universal modelling and simulation language for the social domain is not feasible, so our aim is to provide a visual language that can be customised to particular social domains, by specialisation or the addition of new elements, which can be defined by the modellers of such domains.

Once the model specification is done, the second activity is to perform the execution of the model. Execution means animating the specification, and being able to get some information on how the system evolves over time. This is not normally supported by agent-oriented software engineering tools. But agent-based simulation toolkits do this very well. Consequently, we are considering the use of an agent-oriented modelling language to specify MAS models representing complex social systems, and automatic code generation for existing simulation platforms by applying transformations from the MAS models. This idea of transformation is in line with Model Driven Engineering (MDE) practices.

To implement this approach we require a methodology mainly because the specification can become very large, considering the many aspects of the complex social systems, and unmanageable, driving us to the same problem as using a programming language. A methodology guides the creation of a complex specification and makes its translation to implementation code easier. The source of complexity in modelling social systems is the large number of interacting individuals, their dynamic nonlinear interactions (societies emerge from constant individual change), heterogeneity (people in societies are heterogeneous, with different capabilities, desires, knowledge, etc.), individuals’ reflexivity (Woolgar, 1988) and others. Hence, the methodology should be an agent-oriented one with (1) a suitable language to specify complex social systems as MASs and (2) tools to support the transformation between MAS models and simulation code. Both requirements are satisfied by INGENIAS (Pavón et al., 2005), and this is the main reason for selecting it for this work. This methodology is supported by a set of tools, the INGENIAS Development Kit (IDK), which facilitates the edition of models and the definition of transformations for automatic code generation. The foundation of INGENIAS is the definition of MAS metamodels, which can be customised to particular application domains. This is interesting if we need to extend the existing notation to cope with new issues that may be required for the specification of a simulation model for a social system.

The paper presents how the INGENIAS modelling language can be used to specify social systems and how transformations can be built to generate code on RePast. The translation of INGENIAS concepts to RePast has been a real challenge owing to conceptual differences between the two approaches and missing concepts in most of the cases. To overcome this difficulty these differences have had to be addressed with the definition of agent concepts in the implementation platform. This platform has been chosen as it is one of the most well-known and well-supported agent-based simulation toolkits and, as its principles are similar to others, such as Swarm and Mason, the solution could be easily extended to those (in fact, some work in this sense has been already done (Sansores and Pavón, 2005)). The next section discusses what elements and concepts should be considered in a language for the modelling of social systems. This is followed
by a section where the INGENIAS modelling language is reviewed to see how it supports
the requirements for social systems modelling, and how it can be extended. Then, a
section describes how to make the transformation from INGENIAS social system
specifications to RePast models. These models can be used for simulations on the RePast
framework. It has to be taken into account here that the RePast agent model has to be
enriched to cope with some of the facilities of the INGENIAS agent model. This is done
by building new facilities on top of RePast and defining the mapping of INGENIAS
MAS models to these. The conclusion summarises some of these results and discusses the
limitations and perspective of this approach.

2 Modelling social systems

Computational simulation of social phenomena implies building computer programs
that model the evolution of social processes. This involves the modelling of individuals
and groups, and the processes of social interaction. In INGENIAS, individuals are
modelled as agents, and groups and workflows in the organisation viewpoint serve to
model the structure and dynamics of the social system. Some characteristics of social
phenomena, such as emotions and social pressure, are difficult to approach; therefore
they can be modelled at some reasonable level of abstraction just for specific purposes.

Currently, social patterns under consideration in our work refer to the evolution
of individual beliefs and individual decision making in society (see, for instance, the
case study that is presented in Section 5 in this paper). This requires the ability to
represent social interactions, which give rise to the emergence of aspects of sociality,
such as cooperation, competitions, groups and the organisation. Therefore, the level of
abstraction of the language we will be using is the individual’s social action and mind
(Castelfranchi, 1998).

The language for modelling social systems can be defined in ontological concepts
of the microsociology category of the sociological perspectives on society. Under
this perspective, a human being is capable of conscious thought and self-awareness.
Human action is not simply a reaction to external stimuli, but the result of the meanings,
theories, motives and interpretations brought into a social situation by the individual.
Consequently, we need to conceptualise an individual with mental states and not just like
a behavioural entity. We agree with Castelfranchi when he states that an individual
should be modelled like a goal-oriented agent whose actions are internally regulated by
goals and whose goals, decisions and plans are based on beliefs. Both goals and beliefs
are cognitive representations that can be internally generated, manipulated and subjected
to inferences and reasoning. This is basically the approach that the INGENIAS agent
viewpoint supports.

Another fundamental concept is that of social action. As we have mentioned before,
an individual performs actions that, depending of the level of awareness, could be simple
reactive behaviour or social actions. For an action to be considered social, it needs to be
goal-directed to another entity in a common shared world and that entity should also be
an active, autonomous goal-oriented entity, both entities interacting and perceiving each
other as their similar. For our purpose, we propose to use a classification of forms of
activity and interpersonal relations in sociology to model social actions. Social actions
form the basis of social interactions, which is a dynamic changing sequence of social
actions between individuals or groups. Social interactions form the basis of social relations, a multitude of social interactions regulated by social norms between two or more people, with each having a social position and performing a social role.

Table 1 summarises this classification and illustrates the concepts required to define social actions in our language.

Table 1 Social actions for modelling social systems

<table>
<thead>
<tr>
<th>Social actions</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Communication</td>
<td>Aimed at giving beliefs to the addressee</td>
</tr>
<tr>
<td>Rational</td>
<td>Planned action, taken after considering costs and consequences</td>
</tr>
<tr>
<td>Instrumental</td>
<td>Action that is taken because it leads to a valued goal, but with no thought of its consequences and often without consideration of the appropriateness of the means chosen to achieve it</td>
</tr>
<tr>
<td>Emotional</td>
<td>Action that is taken owing to one’s emotions, to express personal feelings</td>
</tr>
<tr>
<td>Traditional</td>
<td>Action that is carried out owing to traditions, because they are always carried out in such situations</td>
</tr>
</tbody>
</table>

The different instances of an individual type will give rise to different individuals in a society with different roles. It will also be possible to define groups as aggregates of the individual type. The individual attributes may include emotional or character attributes; for example, a numeric stress-level variable that determines the agent’s behaviour in a given time. There are many possibilities of attributes to define an individual, and depending on their values, the individual could perform a variety of predefined actions associated with these values. Society and the environment it imposes could include resources which the individual may count on. Also, the individual may have a set of actions to perform on the environment, such as the action of using a resource. In INGENIAS, resources are described in the environment viewpoint by the set of actions that agents can perform on them, and in the organisation viewpoint by their properties and the way they are shared.

Regarding the agent paradigm to model all the aspects of human cognition and social abilities we mentioned before, we make two considerations. First, it would be unrealistic to expect multiagent models to be able to simulate human cognition and social phenomena to any level of detail. The key is to concentrate on those features of the system which are of the most theoretical significance and leave out others which make our model more realistic but too complex to be implemented. Actually, this is an open debate in the social domain which concerns modelling individuals with cognitive models or modelling individuals with simple internal models. Proponents of the simplicity of agents (Axelrod, 1997) point out that the most interesting analytical results are obtained when complexity at the macro level is produced by simple micro-level dynamics. Proponents of the complexity of agents (Conte, 1999) obtain their arguments especially from the fields of Sociology and Cognitive Psychology, and emphasise the idea that agents should be kept as simple as is suitable. There are uses of computer simulation in which the faithful reproduction of a particular setting is important, such as the prediction of interest rates in an economy. For this purpose, the assumptions that go into the model may need to be quite realistic.
Second, the main concepts of MASs used to model human behaviour and social abilities are agents’ architectures to model human rationality. These architectures are modelled using artificial intelligence techniques (e.g., the Belief, Desire, Intention (BDI) (Bratman, 1987)) and include aspects like knowledge representation, planning, goal directness and social models. Unlike microeconomic theory, which considers agents completely rational, in ABSS a more interesting concern is bounded rationality (Simon, 1982), with agents having limited information and limited capacity to process that information which is considered more realistic to human societies. Another important concept in MASs is that of adaptability, used to represent the ability of human actors to learn and adapt their behaviour over time based on past experiences.

Aside from the main function of the environment concept in a MAS, which is like an entity across which agents can move, perceive and affect, in ABSS another function is assigned that is like a simulated environment which provides a spatial context to actors and which is controlled by the modeller to run controlled experiments. Although spatial environments like grids are very common, there are simulations which consider agents in spaceless environments, or network environments, etc.

The agent communication language is fundamental for the social aspect of human actors. Communication in ABSS is normally performed through the environment, but direct messages between actors are also required, so the agent communication language allows a way to send and receive messages in an agreed-upon manner.

The internal social models of agents allow the modelling of human actors with knowledge of interrelationships with other actors in the world or in their immediate local situation. One last concept not considered in MASs but which is necessary for the dynamic aspects of the system observed is the passage of time in the simulation. In ABSS simulated time is modelled in a time-stepped round base, a degenerated form of discrete-event models.

3 INGENIAS modelling language

INGENIAS is a methodology for the development of MASs that integrates different results in this area. This integration is made by the validation of methods and tools through experimentation in several applications made during the last few years. For this reason, INGENIAS assumes from the beginning the need to evolve the modelling language, the methods and the tools. And to facilitate this, it uses metamodelling. The different tools of the IDK are generated from MAS metamodel specifications. These are specifications of what elements the modelling language has and which constraints should apply in their use. With this framework, changes in the metamodel specification are quickly reflected in the tools. This approach facilitates the evolution of INGENIAS and its adaptation to specific application domains, as its adaptation for social simulation, subject of this paper.

The INGENIAS modelling language is structured in five packages, which represent the viewpoints from which a MAS can be regarded (see Figure 1): Organisation, Agent, Goals-Tasks, Interactions and Environment.
The organisation of a MAS establishes the framework where agents, resources, tasks and goals coexist. It defines structural relationships (groups, hierarchies), social norms (constraints and forms in the behaviour of agents and their interactions) and workflows (how agents collaborate when performing tasks in the organisation).

Groups may contain agents, roles, resources or applications. There may be several ways to structure an organisation, for instance, according to its functional needs. Or at the same time it can be structured by a geographical distribution. An agent, therefore, can belong to several groups at the same time. Assignment of elements to a group obeys some organisational purpose, e.g., because the grouping facilitates the definition of workflows or because its members have some common characteristics.

In general, the concept of role is used to provide more flexibility in the definition of organisations. A role represents functionality or services in an organisational structure. Agents play roles in the organisation. And several agents may play the same role, each one according to its abilities and strategies.

The functionality of the organisation is defined by its purpose and tasks. An organisation has one or more goals, and depends upon its agents to perform the necessary tasks to achieve them. How these tasks are related, and who is responsible for their execution, is defined in workflows. Workflows show the dynamics of the organisation. They define associations among tasks and general information about their execution. For each task, a workflow defines what should be its results, the agent or role responsible for their execution and which resources are required. This is useful to gain knowledge on the relationships between agents through tasks, and the assignment and availability of resources in an organisation.

Both aspects, structural and dynamic, define the macro view of the MAS. This perspective facilitates the management of complex systems as it allows determining the context and norms for the behaviour of agents, similarly to what happens in human organisations.

Agent behaviour is described in the agent viewpoint. It is determined by the agent mental state, a set of goals and beliefs. Also, an agent has a mental state processor, which allows the agent to decide which task to perform, and a mental state manager to create,
modify and delete mental state entities. INGENIAS does not state specifically how to define the mental state processor as it considers that there may be many ways to implement it. For instance, it could be a rule-based engine, a case-based reasoning system, or a neural network. It depends on the needs of the application or the mechanism that best fits according to the developer.

Agents are intentional entities; this means that they act as they pursue some goals. As they are also social entities, they collaborate to satisfy organisational goals. When designing a MAS, it is possible to start with the identification of organisational (system) goals. These goals can be refined into simpler goals up to a level where it is possible to identify specific tasks that satisfy them. Another possibility is to identify individual goals of agents, which could be refined in a similar way. In both cases, there will be a relationship of goals and tasks, which is described in the goals-tasks viewpoint.

As social entities, agents interact. Their interactions can be produced in several ways, the most common being message passing, which is normally asynchronous, and shared spaces, where agents can act (produce modifications) and perceive (the modifications), as is the case in shared tuple spaces. This is described in the interaction viewpoint. In INGENIAS, apart from indicating the types of messages and protocols in an interaction, what is important is to show the intentionality of the interaction: which goals are pursued by the parts in the interaction, and how these contribute to their satisfaction.

Finally, the environment is where agents perceive and act. Depending on the application, perception and acting can have very different meanings. The environment consists of a set of resources, applications and other agents. In many situations, the environment can be specified as a set of application programming interfaces, which would be the classes that wrap it to allow interaction with it.

For social simulation, agents usually need to consider their location in the environment and the evolution of time. These two aspects have required a refinement of the environment viewpoint in INGENIAS.

The temporal perspective deals with the progress of time in the model when executing the simulation. In this case we are assuming that simulations are time driven rather than event driven, as most agent-based simulation toolkits work with this schema. This means that there is a need to model constant time steps to simulate the perception-reaction cycle of agents. A reason for choosing this schema is that an event-driven one would require a central coordinator for events or a complex synchronisation among agents. Traditional discrete-event simulations (Misra, 1986) work with a data structure called the event-list, which maintains a set of time-stamped events to be scheduled for further transmission. At each step, the event with the smallest associated stamped time is removed from the list and the simulated time is set to this value. Then the effect of this event is simulated in the system. This event may cause other events to be added to the list or cause earlier events to be cancelled. Consequently, an event-driven mechanism requires a coordinator to synchronise all events and maintain the list.

The spatial perspective describes how agents are situated in the environment. In general, simulation toolkits provide two- and three-dimensional spaces with diverse configurations.

These extensions have required modifications in the original INGENIAS MAS metamodel and the regeneration of the IDK tools. In this way, it has been possible to get a new personalised IDK for agent-based simulation.
4 From INGENIAS to RePast

In the following, we describe the RePast platform infrastructure and the main components to define a simulation model with it. Then, we list the INGENIAS concepts used to specify a social simulation model and how they are correspondingly mapped to RePast (at both ends of the mapping it has been necessary to make adaptations to concepts). Finally, we describe how the generic transformation process of specifications with INGENIAS is performed, and the correspondence of INGENIAS elements with RePast elements in more detail.

4.1 Social simulation target platform: an introduction to RePast

RePast, originally developed by Collier et al. (2003), is a free open-source programming framework created by the Social Science Research Computing at the University of Chicago for the development of agent-based simulation using Java language. It provides a class library for creating, running, displaying and collecting data from the simulations.

A simulation in RePast usually consists of a collection of agents of any type and a model that sets up and controls the execution of these agents’ behaviours according to a schedule. The execution of the simulation is divided into time steps or ‘ticks’ in which the agents may play an action. The specification of the agents and the environment in RePast is carried out by a set of variables and methods according to the object-oriented paradigm. It does not have intelligent agent concepts or agent architectures, so an agent is an object without any restriction in relation to its internal architecture.

However, it does provide specific libraries to implement functionalities like neural networks or genetic algorithms. Also, it provides facilities for easily developing graphical user interfaces, and for execution, monitoring and presentation of results. The monitoring facilities are capable of visualising and modifying the agent’s internal state dynamically as well as model properties at run-time. The execution and presentation facilities allow the visualisation of an animated representation of the simulation with a range of two-dimensional agent environments and visualisations, and provide tools to record snapshots of the display, social networks support tools, and built-in simulation results logging and graphing tools.

In essence, the main components proposed by RePast to construct a simulation are:

- **The agent.** This is an ordinary Java object, ranging from having no communication at all to communicating via the environment, although it is possible to use libraries to implement an agent communication mechanism.

- **The scheduling mechanism.** The definition of a model period is related to the timing of interactions, although the RePast scheduling approach looks more like a ‘social’ planner that decides who performs what actions. This is due to the way actions are scheduled. The order of agents’ activation is systematically randomised from period to period (each step) in order to avoid the reproduction of biases. Any change we want to occur in a simulation must be scheduled with a scheduler object.

- **The spaces.** The space object provides the environment where agents are immersed. Spaces in RePast function as a collection of agents that defines the spatial relationship of agents relative to each other.
• The model. This is the concentrating object that gathers all the components and sets up the simulation. Its function is fundamental since it is in charge not only of the components but also of the context of the ‘experiment’, i.e., initial parameters, simulation results and visualisation updates.

A normal question that could arise when talking about scheduling components in simulations is, if the global behaviour of the system is said to emerge from agents and their interactions, why is the model not just instantiated, letting the agents interact and monitoring what happens without the scheduler functions? Doing so certainly provides an emergent behaviour, but social scientists would not be able to understand under what circumstances or what local interaction among agents give rise to the global emergent structure. Besides, to discover the consequences of their theories in the artificial society, modellers need to have control over the simulation to observe what is happening at every moment and what parameters are influencing the results.

Finally, social abstractions such as social actors, social actions, organisations and groups are missing in RePast. These abstractions are relevant for our purpose since we intend to model social systems in a high-level modelling language and map these models into RePast computational models, accomplishing in this way the code generation task. In a model-driven development approach, if modelling concepts in the platform-independent models do not exist in the platform-specific models, we have two options to solve this issue: one is to transform or make adequate the high-level language concepts or design concepts into existing platform-specific concepts; or if they do not exist at all, as in the case of the RePast toolkit, we have the second option, which is to create them. If we do not create them in the specific implementation platform, then we are going to use agents and social concepts for modelling social systems but not for implementing them. A negative side effect is that there is absolutely no guarantee that what is being designed corresponds to what is being implemented and that the computational model is not leveraging the MAS paradigm. For this reason, we include a set of social and MAS concepts, such as a computational INGENIAS-SIM model built on top of RePast toolkit facilities, as we will see further on.

4.2 INGENIAS representation of social simulations

A simulation model with INGENIAS is designed with the language elements presented in Section 3 through a set of diagrams representing the relationships between them. The visual language allows social scientists to concentrate their efforts on model design and abstract them from implementation issues. However, detailed implementation should be carried out at a given moment and design model entities should be mapped to a source code to run in a simulation environment. This task is achieved by defining methods to map INGENIAS language entities to the source code for RePast. The following table shows both the main INGENIAS elements or entities, and its corresponding mappings to RePast.
Table 2  INGENIAS’ main concepts for an agent-based simulation model

<table>
<thead>
<tr>
<th>INGENIAS elements</th>
<th>Implementation elements</th>
</tr>
</thead>
<tbody>
<tr>
<td>Agent</td>
<td>Each agent is defined by its purpose (what goals an agent is committed to pursue), responsibilities (what tasks it has to execute) and capabilities (what roles it can play). This concept of agent in INGENIAS has a richer representation of agents than in RePast and is a good abstraction to use in the specification of ABSS. However, the corresponding mapping in RePast does not really exist and needs to be created. One of the Drawables interfaces of RePast will be included in the mapping if the agent is to be graphically displayed.</td>
</tr>
<tr>
<td>Organisation</td>
<td>The organisation allows managing complexity by structuring the system and determining global behavioural rules in a MAS with INGENIAS. For simulation purposes an organisation could be used to model institutions, companies and organisational entities. However, in ABSS most of these entities are simply simulated as agents to avoid complex models. For now, this entity will be used only like a form to structure the model, like the RePast model does.</td>
</tr>
<tr>
<td>Environment</td>
<td>The environment concept in INGENIAS is basically what agents can perceive or actuate, such as other agents, resources, etc. This concept has been extended to include space and scheduling considerations. The space aspects of INGENIAS will map one of the RePast space objects used to represent spatial agents situated in the environment. The scheduling aspects will map the corresponding scheduler object and scheduling set-ups responsible for altering the state of the simulation.</td>
</tr>
<tr>
<td>Tasks</td>
<td>These are actions to be performed by the agents to accomplish their goals. These concepts will map RePast methods encapsulating agent actions, which will be scheduled for execution during simulation steps.</td>
</tr>
<tr>
<td>Goals</td>
<td>We are considering individual goals of the agents in INGENIAS to represent intentional agents in RePast. These goals will map information within data structures of agents, which will act as they pursue some goals.</td>
</tr>
<tr>
<td>Role</td>
<td>Basically, the functionality associated with a role in INGENIAS will be directly migrated to each agent playing this role in RePast.</td>
</tr>
<tr>
<td>Groups</td>
<td>This will map information within data structures of agents.</td>
</tr>
<tr>
<td>Interaction</td>
<td>This is a representation of agents’ communication mechanism in INGENIAS, which includes message passing and protocols. This concept does not have a corresponding mapping in RePast since communications are performed through the environment only. To include more sophisticated communication mechanisms in RePast, it will be required to include data structures associated with the agent with state machines implementing protocols.</td>
</tr>
</tbody>
</table>

4.3 Transforming the specification

The IDK supports the transformation of specifications with IDK modules. Modules (or plug-ins) in INGENIAS are programs that process specifications and produce some output:

- **Source code.** There is an infrastructure that facilitates the transformation of the specification into source code. This is based on the definition of code templates for each target platform and information extraction procedures from the current models.
• **Reports.** Specifications can also be analysed to check, for instance, whether they have certain properties or whether special semantics (defined by the developer) are being respected, or to collect statistics of usage of different elements.

• **Modifications on current specification diagrams.** Though this feature is in the beta stage, a module could insert and/or modify entities in the diagrams, or insert/delete diagrams. This feature is useful for defining personal assistants that interact with the tool.

• **Other models.** A model specification can also be transformed into other models. For example, we can produce a new model of the system from a different viewpoint based on different metamodels.

The mapping from INGENIAS models to RePast is therefore implemented by an IDK module. The module has to traverse specifications, extract information from specifications, and put the extracted information into templates of RePast agents. Environmental constraints (such as spatial and temporal aspects) can also map to the main program.

The IDK module for RePast code generation is developed as an iterative process through several steps. This process can be applied similarly for other agent-based simulation toolkits. The basis for code generation is the availability of code templates for the target platform, RePast in this case. This is usually the most difficult to obtain as it requires a good knowledge of how to implement agents in the target platform. Our experience has shown that this can be accomplished through an iterative process, in which the developer progressively defines the architecture of the code for the target platform and the transformations from specification to code templates. This process could be sketched in several steps, which have been applied to develop the module for RePast but could be used for modules that generate code for other platforms:

**Step 1 Small initial prototype.** The process starts with a simple prototype of the simulation model. Initially, the developer would centre in on one or more features of the specification, those that are easy to implement, if possible; for instance, how to make an agent process an event. As a result, the developer gains knowledge of the target platform and has a prototype of an application on the target platform that realises a small part of the specification with a selected set of features.

**Step 2 Marking up the prototype code.** Looking at both the prototype and the specification, it is possible to identify parts of the prototype that match parts of the specification. As a result, the developer identifies possible mappings from the specification to the prototype code. This is reflected in a prototype code marked up with tags. The marked-up pieces of source code are called templates.

**Step 3 Generating/Modifying a module.** A module has to traverse the specification in order to obtain the information required to instantiate and fill in the prototype templates. The IDK provides an API for traversing specifications and Java packages for building modules. In concrete, the module engineer has to extend the class `BasicCodeGeneratorImp` for the code generation module. Other support classes may be created as well.
Step 4  *Deploying the module.* The resulting Java classes and templates of the module are put together into a jar file. This jar file is deployed in a specific folder where the IDK Editor can load it dynamically.

Step 5  *Testing the module.* Testing is performed with the IDK Editor. By executing the module over the specification, the developer can check if the diagram is traversed properly and if all templates have been filled in as they should have. Also, templates demand concrete information, and it may be possible that this is not present or that it is not expressed as it should be. Therefore, it may turn out that the specification was not correct or was incomplete. In this sense, any module can be useful to validate the specification against some completeness criteria. As a result, several kinds of problems may appear: with the code generated by the module, with the traversal of the specification, or with the specification itself.

Step 6  *Debugging.* If something goes wrong, debug the prototype and go to:

a  Step 2 – if there is a new code that was not marked up before

b  Step 3 – if the failure was in the module and the data traversal

c  Step 4 – if there was a failure in the prototype that could be solved without marking up the code again.

Step 7  *Refinement and extension.* When the module is finished, it can translate diagram specifications into code or perform verification of some properties. However, the module performs these tasks with a reduced set of the diagram specifications. The next step would be to take the code generated by the module and extend it so that it can satisfy other parts of the specification. Therefore, we would go back to Step 1.

In this way, modules produce code using a template-based approach. For RePast, there are several templates:

- A RePast agent template. This template implements a prototype of an INGENIAS agent, which implements a drawable RePast interface for being displayed in a simulation and a `RunnableAgent` interface.

- A model template. This template implements the main class which sets up the simulation components, like the collection of agents and the scheduler. For complicated scheduling mechanisms we could have a separate scheduler template.

- An environment template. This template implements the spaceless environment for agents without physical references as well as the RePast space environment, where spatial agents interact.

- Role, Goal and Task templates, which are usually embedded in the agent template.

The way the module works is shown in Figure 2. A developer defines a template of a RePast agent (Steps 1 and 2) and extracts data from the MAS specification (Step 3) to generate the code. A RePast agent needs a goal, a role and a name. As a result, we get a Java RePast agent, which is instantiated from the RePast agent template.
These elements are configured within a module and deployed in the IDK (Step 4), and tested over the specification of a MAS (Step 5). As a result of the testing, we would obtain the generated code presented in Figure 2.

As we have said before, RePast lacks many concepts necessary to model social actors, although it gives users complete flexibility as to how they specify the properties and behaviours of agents. This flexibility allows RePast to be applied to a wide range of domains. However, it also means the burden of agent design and implementation is passed on to the simulation developer. While this may not be much of a problem where simple agents will suffice, in many applications it is useful to provide agents with more sophisticated capabilities; goal-directed agents that adapt to changing circumstances are desirable in many contexts.

Figures 3, 4 and 5 depict (1) the Java module developed to provide goal-directed behaviour specifically for agents in RePast simulations, this is, the computational INGENIAS simulation model, and (2) the main RePast elements used to develop an agent-based simulation. The Java module is the basis of RePast templates and relaxes the knowledge requirements to build agent-based simulations with RePast.

A typical simulation written with RePast will have at least three classes: the agent class, the model class and the space class. The RePast agent class will be largely simulation specific. If the agent is to be displayed, it should implement one of the Drawable interfaces. The RePast model class sets up and controls both the representational (variables such as initial parameters) and infrastructure (variables such as a schedule) parts of a RePast simulation. All RePast simulation models must implement the SimModel interface. RePast provides an abstract class SimModelImpl that partially implements this interface and it is expected that most if not all models will extend this class. Figure 3 shows these RePast classes in the package uchicago.src.sim, which represents the main elements to be mapped by INGENIAS simulation model elements.

Figure 3 also illustrates INGENIAS simulation model elements in the package social.sim.model. In this package the model class implements required SimModel interface methods as well as other methods that divide the infrastructure and representation set-up process into coherent groups. It also provides features for creating and setting up the components of the simulation, such as agents and model’s parameters.
It also contains all the agents of a simulation and stores them in a list, referencing them via the AgentType abstract class. This class represents an agent type and provides a standard access point for referencing agents in the model. It implements the SteppableAgent interface, which contains the step() method, which will be executed in each simulation step on all agents in a simulation. The implementation of this method, which conforms to the design pattern TemplateMethod (Gamma et al., 1995), involves the execution of all methods defining the agent’s behaviour (perceives, deliberates and acts). The AgentType class also includes sophisticated features like goal-dynamics, roles, actions, tasks and interactions through relations with corresponding classes. The Agent class represents the social actors in a simulation and it is a concrete implementation of the AgentType abstract class.

Figure 3  RePast and INGENIAS model and agent simulation elements
The RePast class space controls the environment in which the action takes place. In RePast it is theoretically possible to omit the explicit space object, but this is rarely done. This means that agents are always immersed in a space relationship relative to each other, represented with one of the space classes of the package `uchicago.src.sim` depicted in Figure 4, which also shows the corresponding INGENIAS simulation model elements.

**Figure 4** RePast and INGENIAS environment simulation elements

We provide a broader view of the environment. Instead of stating that the agents are physically in the environment, we also give the possibility to define spaceless agents. In this sense, the environment will be more notional than physical. The space environments we support are more flexible with respect to different representations, achieving this flexibility through an `EnvironmentInterface` interface, which offers a standardised access point for all environment implementations through a common set of functions. As illustrated in Figure 4, the package `social.sim.model` contains this interface and a concrete `Environment` class which implements it; the `Environment` class uses a space RePast object.
Object2DGrid to implement the interface. This object represents a two-dimensional grid space. The concrete class Environment conforms to the design pattern Adapter (Gamma et al., 1995), allowing one to easily replace the space object by any other RePast spaces. The spaceless environments are supported with the aid of a space object not depicted in Figure 4, but which provides methods for manipulating the environment by the agents, for example, to locate other agents or to use a resource.

Within a RePast simulation all state changes are the responsibility of a scheduling mechanism implemented with the Schedule class. The mechanism consists of setting up method calls on objects to occur at certain times. These method calls must be wrapped by subclasses of the BasicAction class. A BasicAction consists of some variables used by the scheduler and an abstract execute() method. Any class that extends BasicAction class must implement this method. It is in this method that the actual method call or calls to be scheduled should occur. These two RePast classes represent the main elements of RePast scheduling and can be observed in Figure 5 in the package uchicago.src.sim.engine.

As explained in Sansores and Pavón (2006), this scheduling mechanism is contrary to some of the ideas that have motivated a multiagent-based approach, since creating a list of actions to be taken at each time step looks more like a central coordinator approach. For this reason, we do not schedule method calls on agents, rather we schedule the agents themselves. In the social.sim.model package illustrated in Figure 5, we can observe how we will achieve this, and the INGENIAS simulation model elements and their relationships with the corresponding RePast elements.

In the social.sim.model package, the Model class contains an inner class named ModelRunner, which is itself a subclass of the RePast BasicAction class. Therefore, an instance of ModelRunner can be scheduled to be executed each step of a simulation as a BasicAction. ModelRunner object contains an execute() method which will be executed each time step, and is in this method where it is called the step() method of each agent in the simulation model, that is in agentList list. The step() method of each agent does not represent an action but the skeleton of an algorithm implementing its behaviour. This means that it is in the agent’s internal structure where we have to include a decision mechanism of which actions to execute in each time step, in contrast to RePast, which suggests calling actions in agents according to predefined listed actions in the model. Of course, although our point of view is better aligned with the MAS paradigm, it is certainly a more difficult way of timing agents’ actions. In the case of simulations where modellers need to have control over the artificial society to observe what is happening at every moment and what parameters are influencing the results, it is always possible to combine both scheduling approaches.

The Java module implementing the INGENIAS simulation model also provides a Goal class which represents goals pursued by an agent and a Role class which may be played by the agent. The role also pursues goals. An individual agent can adapt its capabilities and observe specialisation of roles based on a reinforcement mechanism, in this way dynamically changing the roles played. The Task class allows defining a set of tasks to achieve the different Goals an agent is pursuing; thus an agent is related with a set of tasks to perform. A task may be composed of one or more actions defined with the Action class; the deliberating mechanism has to decide what action to execute depending on the goal an agent is pursuing at a given time. An example of actions could be sending a message, moving in the environment, or application domain actions such as moving an object.
5 Case study

The way a user, an expert in a social domain, exploits the MDE approach for developing social simulations is through a set of tools provided by the IDK. The user edits and defines some model of a social system with the IDK Editor. From this editor it is possible to invoke several types of modules to work with the model. Normally, before running a simulation of the model, the user should verify that the model satisfies certain properties; for instance, that all the elements that are required for the simulation have been defined, or that there are no isolated agents in the system. Other types of properties that can be considered useful can be verified by creating the appropriate IDK plug-in.
Once all models satisfy the required properties, the user can invoke the code generator module for running a simulation under a particular toolkit. Sometimes it can be useful to run the same experiment on different platforms, to get more confidence in the results (like what was done with RePast and Mason toolkits, as reported in this case study). From this moment, the simulation toolkit can be used to obtain results. These have to be interpreted in terms of the model. This interpretation could be done directly by the user or, better, supported by the IDK module in order to be presented directly in terms of the model entities.

This proposal has been tested on different models with different complexities. Here we show the results of the work on one such model, which has been implemented in two different simulation toolkits, RePast and Mason. The case study described here is a replication of a model originally described by Di Tosto et al. (2003).

We tried to reproduce this model starting from the text description provided by the authors of the original model. The model examines which type of mental construct corresponds to the norm of reciprocity within the simulation-based study of altruism: a simple altruistic algorithm, or a cognitive one. In the former, agents apply routines under given conditions. In the latter, agents execute actions to achieve their goals.

In our experiment we reproduced the dynamic variant of smart agents in Di Tosto et al. (2003) because it demands concepts to model rational agents and we intended to test the expressive capacity of the INGENIAS modelling language to specify agent-based simulation models with cognitive agents. The system studies altruism among smart entities in a society. In this model the society consists of a population of bats (the agents) that live in roosts, where they return after hunting and where they perform social activities like grooming and sharing food. The bats are modelled as agents. Roosts are modelled as aggregates of bats or groups in INGENIAS. In roosts, bats are allowed to share food and to groom one another. Help-giving allows bats to achieve credits, which will be extinguished if and when help is returned. Bats are endowed with social knowledge, consisting of a memory of past grooming and food-sharing interactions, and of consequent credits. The smart algorithm implemented a small number of rules modifying the value of agents’ motivational force for pursuing goals. This force was supposed to increase or decrease as an effect of others’ impact on one’s own conditions. If one receives help, the force of the altruistic motivation increases, whereas it decreases if one is denied help. This effect is even stronger if one’s actions are not reciprocated and one’s credits are therefore not extinguished. In this way agents play different strategies depending on their motivational force for giving help: cheater, prudent, fair, generous and martyr. The main question originally raised was, what are the effects of goal-dynamics on altruism?

We created a specification of this model with the INGENIAS modelling language based on its original description and applied the transformation process already reviewed. The model specification consists of the different INGENIAS MAS viewpoints described with a set of diagrams. Figure 6 shows the organisation viewpoint. It simply structures the model (represented as an organisational entity) in roosts (represented as groups with the group icon) and a social (credit) network (represented as an organisational network with the network icon). There is only one type of agent, bats, which can play several roles. Bats belong to roost and social network groups. In a roost, a bat can play the role of altruist or recipient depending on whether help is given or received. Bats belonging to a social network group can exploit the functionality provided by this entity, for example by
engaging in social relationships with other agents and maintaining a history of these social relations. In this case this functionality is essential for increasing or decreasing agents’ altruistic behaviour. Figure 7 depicts how an agent type is specified in the agent viewpoint. The entity agent type is defined by its responsibilities (goals it is committed to pursue, tasks it has to execute and roles it can play). It represents a bat pursuing the survive goal, roles altruistic and recipient played by the agent, and day and night plans (depicted with a workflow icon but, instead of representing organisation functionality, it represents agents’ internal functionality like a set of tasks to be performed) the agent could perform and which are further broken down into the tasks. Thus, agents’ behaviour is specified by defining the mental state of the agent, as well as its mental state management and processing. The mental state entities consist of an agent’s goals and the information it has about the satisfaction of those goals, its knowledge about the world and facts reflecting its past experience. Tasks play a fundamental role in the evolution of agents’ mental states, since they describe how the mental states of agents change over time, the consequence of executing a task with respect to the mental state of an agent, how to achieve goals and what happens when a goal cannot be achieved. In Figure 7, we can observe all these entities. The functionality of the mental state manager and processor SimAI are implemented in the Java module created for each simulation toolkit. The environment viewpoint is illustrated in Figure 8. This viewpoint consists of two types of entities represented as environment applications. The time application provides the scheduling functionality and is a default entity in all simulation models. It represents the stepping mechanism the agents adhere to. The space application provides the functionality necessary to deploy and manage agents in a physical or notional environment. In this case, the space entity is the grid where the agents are situated and directly maps the grid space class of RePast.

Figure 6 Organisation viewpoint diagram with INGENIAS modelling language
Finally, the transformation mechanism is applied to the complete specification (not all diagrams are depicted for simplification purposes) to generate the agent-based simulation source code. The implementations to compare results were generated for RePast and Mason, both generated from the same visual model. In Figure 9, we show the execution of both applications with the same parameters: 10 000 days, a population of 150 bats, ten roosts, reproduction mechanism and inheritance activated. On the left side of the figure we can see the statistical graph generated with RePast simulation, and on the right side the one obtained from Mason.
Figure 9  Emergence of altruistic strategies with inheritance (for colour, see online version)

Notes: On the left side is the RePast statistic graph, on the right the Mason graph. Values are number of agents per strategy (y) and simulation steps (x). Population is composed of ten roosts of 15 agents each. Every agent is Prudent at the beginning.

Based on the results presented in the original model (basically different strategies emerge in the experimental condition and these strategies correspond to different patterns of relationships among agents’ goals), it is clear that Repast simulation results aligned very well with the original ones. In both conditions, the dominant strategy appears to be Martyrdom. Mason statistical results did not align with the original ones, as we can see in Figure 9. Note the difference in scale between the y axes. In the Mason chart, population decreases even though reproduction is activated. This could be due to an inadequate specification, such as missing parameters in the specification, or to an erroneous implementation, in this case an erroneous transformation of the visual model.

We cross-checked both independent simulations and even though they were implementing the same underlying process, the Mason implementation did not produce results that were sufficiently close either to the original model or to Repast results. We had a high degree of confidence in the similarity of both implementations, but one of them showed significant differences from the original published results and the other one aligned very well. We speculated that there was a possible source of inconsistency introduced by random number generators or the simulation engine itself, so we debugged the second implementation to find possible errors.

This task showed some issues (static data structures not adequate for dynamic scheduling) in the scheduling mechanism of the Mason toolkit that were introducing some biases in the results, and having a second implementation to compare with was very useful to find them and discard translation mechanism errors. Figure 10 shows the Mason statistical chart after we handled the bias issues and generated the right code.
As a main result of the experimentation we could prove the expressive capacity of the INGENIAS modelling language to model intelligent agents with cognitive properties. However, although the MAS visual modelling language is more intuitive and easier to use for a social scientist than a programming language, the specification can become very complex. This complexity can be greatly reduced by generating a domain- or theory-specific modelling language based always on INGENIAS metamodels.

6 Conclusions

The process and tools described here facilitate the specification of agent-based social systems and their simulation on existing agent-based simulation toolkits. This enables social scientists to avoid writing code in a programming language and to describe models in a more intuitive language, at a greater level of abstraction and closer to their problem domain. This is illustrated with a real example in Section 5, where a case study is modelled with INGENIAS diagrams. Also, new modules can be provided for code generation for new simulation platforms. Although building such modules requires the participation of a software engineer, who should be experienced in INGENIAS metamodels and Java programming, the final user only needs to know how to use the graphical editor and the way to interpret results from simulation, which could be more or less guided by IDK modules.

Another advantage of this approach is that it allows replication of simulations on different platforms. As modelling is performed with a graphical language, then transformed to code, by doing the transformation on different simulation platforms it is possible to compare results. Replication has been demonstrated to be very useful for isolating errors that are difficult to observe.
An interesting issue in INGENIAS to consider for further work is the ability to extend the INGENIAS MAS metamodel to create domain-oriented specification languages. This would facilitate even further the modelling activity of social scientists as they could use concrete domain concepts rather than pure agent concepts from INGENIAS. This work has to be done with domain experts and this is a task that we are currently addressing.

Acknowledgement

This work has been developed in the context of the project TIN2005-08501-C03-01, funded by the Spanish Council for Science and Technology.

References


Modelling and simulation of social systems with INGENIAS


Notes
1 repast.sourceforge.net/repastpy/
2 Available at http://ingenias.sourceforge.net.
3 http://repast.sourceforge.net